Relational Framework for Information Extraction

Yoav Nahshon
Relational Framework for Information Extraction

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Yoav Nahshon

Submitted to the Senate of the Technion — Israel Institute of Technology
Sivan 5778 Haifa May 2018
This research was carried out under the supervision of Prof. Benny Kimelfeld, in the Faculty of Computer Science.

Some results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author’s doctoral research period, the most up-to-date versions of which being:


**Acknowledgements**

I would like to thank my advisor, my parents, and my friends for their major support.

The generous financial help of the Technion is gratefully acknowledged.
List of Figures

3.1 A run of a Spannerlog program. ........................................ 23

4.1 A document and one of its sentences .............................. 30

4.2 Extracting named entities from a document .................. 33

4.3 Extracting named entities with splitting ...................... 34

5.1 Dataflow ................................................................. 39

5.2 Screenshot web interface for Spannerlog ..................... 43
Abstract

Unstructured textual data conceals within itself structured data, and oftentimes it is accompanied by metadata. However, relational databases, which are highly suitable for storing structured data, typically treat text as a black box, as they lack the means for handling it sufficiently. In the context of text analytics, an essential component in many applications from this domain is Information Extraction (IE), the task of extracting (in a structured format) valuable knowledge from textual data. Typically, modern IE pipelines are constructed by (1) loading textual data from a database into a special-purpose application, (2) applying to the text a myriad of text-analytics functions that produce a structured relational table, and (3) storing this table in a database. However, this approach is prone to laborious development processes, complex and tangled programs, and inefficient control flows. These deficiencies have given rise to declarative solutions that automate significant parts of the manual work. However, such frameworks typically stitch together various programming components and technologies, and may lack an all-binding theory. In this thesis we embark on an effort to lay foundations of general purpose and text centric database management systems. Concretely, we introduce a novel formal framework, called Spannerlog, where we extend the relational model by incorporating into it the theory of document spanners, and define a Datalog-like query language for this model. Our main contribution is a uniform framework for textual data management w.r.t. unstructured data (text), structured data (extracted information and metadata like identifiers and timestamps), and functions that carry out transformations from the former to the latter. The formal foundations on which we built on our framework provide new capabilities and opportunities to be explored: (1) a better understanding of the system through theoretical studies; here we report on initial results concerning the expressive power of Spannerlog programs. (2) Diminished software complexity; on a single framework developers can write IE programs and query the extracted information in concise and readable manner. (3) New optimization opportunities due to static program analysis on top of Spannerlog’s formalism; to illustrate these opportunities we present the notion of split correctness, that enables the construction of parallel execution plans based on data splitting, while providing provable correctness. We believe that the formalism of Spannerlog will have a substantial impact on the way systems manage and query textual data.
Chapter 1

Introduction

In the era of big data and social networking, the high potential value within textual datasets constantly increases. Consequently, the capability to efficiently process and easily query text has become more crucial for a variety of applications and fields. As an example, security stakeholders possess a special interest in textual data sources like blogs, Facebook posts, tweets and news articles that may alert ahead of time about potential security threats [SAD+16, MDM+16, ICKM16]. Such textual resources often hold valuable information on pressing security issues including privacy abuses, terrorism and human trafficking [BDD+16, SS17, BPRA14]. Textual datasets are naturally stored in relational databases as they are typically accompanied with metadata such as timestamps and identifiers that are neatly accommodated by database tables. Notwithstanding, data manipulation capabilities of traditional database systems are in fact not suitable for extracting significant knowledge from unstructured data (e.g., text). To conduct a high quality and meaningful text analysis out-of-database solutions are typically required. These solutions are 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 and scientists. Systems that do combine databases and text analytics are usually bundles of a variety of different technologies. Such systems oftentimes comprise scripting language, an ordinary relational database, and a particular statistics/machine-learning functionality. This approach requires developers to master a collection of inherently different technologies and programming frameworks. Moreover, a bundled package limits the ability of the system to analyze and understand the entire flow as a whole, and hence, misses significant opportunities of optimization. Specifically, an important objective for optimization is reducing the use of expensive Natural Language Processing (NLP) algorithms. A wisely constructed execution plan can achieve this by filtering out irrelevant textual data at early stages of the workflow.

In this thesis we aim towards overcoming these shortcomings by turning to the fundamentals of the relational model, and extending it with respect to textual data. We
present Spannerlog, a formal framework for text analysis built upon the formalism of document spanners for information extraction (IE) [FKRV15]. The data model used by Spannerlog considers two types of atomic data values:

1. **String** - represents the unstructured data, that is, text. Metadata, such as identifiers and timestamps, is also represented by strings.

2. **Span** - represents the information extracted from the text.

In Spannerlog, every relation can have attributes of both types. On the basis of this data model, we define for Spannerlog a new query language that is similar to Datalog [AHV95]. This language incorporates concepts from spanner theory and Xlog [SDNR07]. Spanners are information extractors, applied to an input text. More formally, a spanner maps a given string to a span relation (i.e., a relation whose schema consists only of attributes of type span), over a predefined schema. One can observe that general NLP functions, such as part-of-speech (POS) taggers and dependency parsers, can be thought of as a generalization of spanners in that they too map strings to relations, but as opposed to spanners, they are not restricted solely to span relations; the target relation can have intermixed attributes of types string and span. In Spannerlog, these generalized spanners are called **IE-functions**.

We provide a proof-of-concept implementation. We shall discuss some interesting aspects regarding our implementation effort, and present several experiments that motivated us to formalize a new theory regarding data splitting as a mean of optimization using static analysis. Practically, text analytics tools, or in our context IE-functions, most likely will constitute the most considerable bottlenecks in an IE application. In the case of an in-database solution for IE, traditional query optimizers fall short in dealing with the choke points of this type. For this reason, new techniques should be considered. In this work we present the notion of **split correctness**. This gives rise to constructing a query plan that splits the data according to some split policy in order to process it in parallel fashion, while maintaining **split correctness**, that is, providing a formal guarantee that processing the program, or possibly a different one, on the split data is equivalent to running the original program on the unsplit data.

As in the cases of the relational model by Codd [Cod70], the spanner theory by Fagin et al. [FKRV15], and as well as in many other examples, the coming about of these formalisms has given rise to meaningful research, accompanied by practical implications. The Spannerlog framework is an effort in that direction, for the case of information extraction systems.

**Outline.** Chapter 2 provides general background on information extraction, and surveys related work. Chapter 3 formally defines Spannerlog, its semantics, and present several results concerning its expressive power. Chapter 4 introduces the notion of split correctness. We present an implementation of Spannerlog in chapter 5, and discuss
several case studies. Finally, in chapter 6 we conclude and discuss future research directions for Spannerlog in both the practical and the theoretical sense.
Chapter 2

Background

Data falls on a spectrum from unstructured to structured. Unstructured data refers to information that either does not have a pre-defined data model or is not organized in an consistent manner. Examples include text, images, video and more. Conversely, structured data conforms with some data model that standardizes the data elements in terms of their relevant properties, representations, and the way they relate to one another. The relational model by Codd [Cod70] is a widespread and well-studied approach to formally describing data models. Between the two sides of the spectrum there are types of data that can be classified as semi-structured data. These do not conform with the formal structure of data models, but nonetheless contains tags or other markers to separate semantic elements and enforce hierarchies of records and fields within the data. Standard and well-known formats for representing semi-structured data are XML and JSON.

In the following we shall review the various topics in the context of databases and text analytics, as we explain how they relate to this thesis.

2.1 Information Extraction

Information extraction (IE) refers to the automatic extraction of structured information from unstructured data sources. The extracted information is typically entities, particular attributes describing them, and types of relationships between them; conventionally, the format describing it conforms with some predefined relational schema. Thus IE can be, and frequently is, thought of as the task of populating a predefined relational schema from unstructured data, or as in the context of this thesis, textual content. Prevalent tasks in this area include:

1. Sentence Boundary Detection (SBD) - deciding where sentences begin and end in a given text. Note that SBD is not as trivial as it may first appear. Even though a period (‘.’) conventionally denotes the end of a sentence, it is natural to think of it as a delimiter that separates sentences. However, a period may also denote an abbreviation, decimal point, an ellipsis, or can appear in an email address.
2. **Part-of-Speech (PoS) tagging** - assigning to each word a label, such as noun, verb, adjective, etc., indicating its part of speech according to its context.

3. **Named Entity Recognition (NER)** - identifying named entities (i.e., real-world objects) in text, and classify them into a predefined set of categories such as Person and Organization.

4. **Relation extraction** - extracting tuples of entities that satisfy a predefined relationship, such as person-organization.

5. **Event extraction** - finding events of predefined types, along with their key player entities. For example, such an event could be the acquisition of company A by company B.

6. **Temporal information extraction** - associating mentions of facts with mentions of their validity period.

7. **Dependency parsing** - analyzing the grammatical structure of a sentence.

8. **Coreference resolution** - finding all expressions that refer to the same entity in a text document.

9. **Sentiment analysis** - determining the affecting state, or sentiment, expressed in a given text. A basic sentiment analysis tool would commonly classify the text into three categories: positive, negative, or neutral.

These are basic, well-studied tasks in IE, and as such, implementations of state-of-the-art algorithms were made available online to be used for downstream applications. As an example, *Stanford CoreNLP* [MSB+14] is a well-known toolset for NLP that was developed and is maintained by Stanford University.

It is often the case that IE constitutes one component in a larger software pipeline, and particularly in machine learning applications, where statistical methods are applied to reason over the data. IE plays a significant role, most notably in the following widely-used techniques:

- **Feature extraction.** This is the process of transforming the input data into a reduced set of features (i.e., observable characteristics), usually in a vector form, to be used in subsequent phases of the application. A feature can be thought of as a function that maps the data to some value; typically a Boolean value is used, indicating whether a certain characteristic of the data is present or not. For example, in an application that extracts pairs of spouses from textual data, one possible feature is the existence of two separate mentions of persons, and the words “got married”, all in one sentence.

- **Distant supervision.** Conventionally, in supervised learning there is a learning algorithm that infers a model from a labeled data. This is a set of pairs, each
consisting of an input object, and a desired output value called a label. A traditional approach for collecting labeled data is to have humans to manually label the data. However, this technique is an expensive one in terms of both time and money, and does not scale to large datasets. An alternative approach is called distant supervision, in which the data is labeled automatically based on heuristics or rules. Although much cheaper than manual labeling, this approach is more prone to mislabeling, which ultimately may result in a less accurate model if not used with care.

2.2 Rules vs. Statistics

Text written in human (natural) languages is usually highly complex, noisy, and ambiguous, which make IE a challenging task. Approaches to information extraction can roughly be divided into two main classes: approaches based on handcrafted rules [CMB+11, CKL+10, MTU+01, SDNR07, LPC+11], and approaches based on statistical models trained in a supervised fashion [BGK+05, McC02, MSB+14, WW07].

The underlying assumption in a rule-based approach is that a solution to the IE task can be fully specified as a collection of manually encoded operations that are deterministic in nature. Often, however, the IE task and the underlying textual data have inherent properties that violate this assumption. In particular, traditional rules can hardly accommodate the flexibility by which natural language can be used to express knowledge, alongside common human practices like informality, deceit and sarcasm. The gap between this traditional programming philosophy and the nature of the text of interest is typically managed by long chains of rules, filters over rules, exceptions over filters, and so on.

In contrast, the NLP community has been focusing on probabilistic modeling techniques far more than rules-based approaches. Examples include naïve Bayes classifiers, and various kinds of probabilistic graph models, such as hidden Markov models [BMSW97, BDS01], maximum entropy Markov models [KM02, MFP00], and conditional random fields (CRF) [LMP01], that model the text and the annotation as a probabilistic generative model where with specified dependencies. Over the recent years there has been a noticeable rise in popularity of methods based on neural networks [CWB+11, SPW+13, CM14], which have the advantage over traditional statistical methods of eliminating the need to manually identify features for the learning process, provided that the training data is sufficiently large.

It may be the case where the boundaries between the two approaches become unclear. Some adopt hybrid techniques that encompass components of both approaches [FRF06, CM03, CCRP05, CR03, MMR03, RJBS07, SWW+15], or alternatively formulating rules that are not written by an expert, but automatically induced [MS06].

Interestingly, Chiticariu et al. [CLR13] have found that while statistical models dominate the research in academia, commercial applications are mostly implemented
as rule-based systems. In their work they argued that this discrepancy cannot be explained by the latency of translational efforts from research to industry, and noticed that research and industry measure the costs and benefits of information extraction differently. Even though statistical models often perform better in information extraction challenges, rule-based systems sometimes fit better in the constraints and requirements of real-world use cases. For example, availability of labeled data, stability of the specification or traceability of results. Furthermore, business considerations may require high flexibility in the behavior of extractors, such as avoidance of specific types of mistakes; such flexibility is inherently built in the rule-based approaches, while it is often a challenge to tweak a statistical solution, or generally to express different kinds of domain knowledge. Gupta and Manning [GM14] reference the work of Chiticariu et al. [CLR13] and assert that “[…] rules are effective, interpretable, and are easy to customize by non-experts to cope with errors.” It is thus argued that the need for rule-based information extraction will not decrease in the foreseeable future. Some of the state-of-the-art statistical IE systems comprise components that are rule-based as rules fit well for high-level feature extraction, for solving different pre- and post-processing tasks, and for distant supervision; that is, semi-automatic creation of labeled data [RSS+14, RDSW+16, DSRR+16]. It is often faster to engineer one rule than to annotate repeating mentions of a specific entity.

2.3 Declarative IE Frameworks

As mentioned in the previous chapter, a common way to tackle various IE tasks is through writing code in an imperative programming language such as Java or Python, and then using existing tools, or alternatively, creating new ones, to suit the application. By observing that extracted information naturally resides in relational database tables, researchers and developers have begun investigating declarative approaches, which were inspired by the massive success of SQL in the field of databases for querying the underlying data. Under the declarative mindset, programmers request the desired result, and allow the machine to figure out under the hood the required set of operations, while placing trust in the system to be capable of finding and choosing the most efficient solution among all possible. A well-known argument against imperative solutions compared to declarative ones claims that the former implicitly lead to bug-prone programs, lengthy development cycles, and low maintainability; additionally, they limit the ability of the compiler/execution planner to understand the program and thus to better optimize its execution. For these reasons, declarative frameworks for IE have flourished over the past years.

The declarative paradigm goes hand in hand with the rule-based approach. Systems in this realm mostly consist of a specification of a text-based rule language and an execution engine, which is able to apply the rules on documents in order to identify new information [AO98]. With this type of frameworks, the development process of
applications mainly consists of manually writing rules is some logical form by a knowledge engineer; that is, a professional whose task is to formulate human knowledge and expertise into logical, computer-readable form. The matching algorithm of a set of rules (rule grammar) is often implemented as a Finite State Transducer (FST), an automaton which traverses the annotation lattices and creates or modifies annotations [CMT99, PSX04, BN10]. The automaton processes the document just once and does not react on its own modifications. A popular strategy is the usage of cascaded rule grammars, where one grammar is based on the results of previous grammars. This approach provides many engineering advantages, for example the easier specification of complex patterns by describing them as a combination of simpler ones, or the clear separation of the stages of engineering approaches and their contexts [BN06].

Significant efforts have been put to design and establish development frameworks for IE. One of the most commonly used IE systems is the General Architecture for Text Engineering (GATE) [CMB11], an open-source project by the University of Sheffield, which is an instantiation of the cascaded finite-state transducers [AO98]. In GATE, a document is processed by a sequence of phases (cascades), each annotating spans with types by applying grammar rules over previous annotations. Another general-purpose annotation system is the Unstructured Information Management Architecture (UIMA) [FL04]. Both GATE and UIMA are frameworks that enable users to define pipelines of various software components that annotate language data, each of which uses a different internal representation for annotations over data. Xlog [SDNR07] extends Datalog (a logic-based relational query language) with special primitive types such as documents (distinguished chunks of text) and spans (intervals of text within a document), and matchers of regular expressions.

Recently, the concept of document spanners [FKRV15] (or simply spanners for short) has been established as formalism for relational querying of text. In that formalism, primitive information extractors (e.g., regular expressions with embedded variables) instantiate relations over spans from the text, while relational algebra manipulates these relations. This formalism features a clean theoretical model, giving rise to nontrivial investigation via corresponding types of transducers. In a subsequent work [FKRV14] it has been shown that the ad-hoc cleaning strategies in the above IE systems can all be cast as instances of the well studied concept of database repairs [ABC99] in the presence of tuple priorities [SCM12], giving rise to fundamental analysis such as well definedness and expressive power. The framework of spanners was inspired by SystemT [CKL10], IBM’s principal IE tool that features an SQL-like query language. SystemT supports an SQL-like declarative language named AQL (Annotation Query Language), along with a query-plan optimizer [21] and development tooling [LCC10]. This system is typically integrated within a larger software bundle for data analytics (e.g., IBM BigInsights).

Other industrial data analytics tools that support IE development include Attensity, Clarabridge, IBM BigInsights, HP Autonomy, Oracle Collective Intellect, SAS, SAP, and more. Stanford’s DeepDive [Zha15] is a framework for knowledge base construction.
that provides a Datalog-like language to combine various technologies, such as Python scripts for information extraction, CSV readers, a PostgreSQL database, and an inference engine for Markov-Logic Networks [DL09].

In this thesis we attempt to take the above systems one step further. We establish formal foundations by extending the relational model using spanner theory. This allows a comprehensive study to take place which in turn may result in significant impacts on practical systems in terms of ease of programming, and optimization of execution plans.

2.4 Optimization

A query optimizer is an inherent part of a standard Relational Database Management System (RDBMS). Its function is to analyze a query and then determine the most efficient way to execute it. This is an important step in the processing of a query as it can greatly affect the execution time. Traditionally, such systems mainly consider a cost-based optimization technique. A cost-based optimizer generates a set of potential plans for a query and estimates to each one a cost which is proportional to the expected resource use needed to execute it. The plan with the lowest cost is then chosen.

Characteristically in text applications, the predominant components with respect to running time are the text analytics functions applied to the input text. In Spannerlog’s lingo these are the IE functions. In addition, recent years have shown an increased interest in large scale analysis, which results in relatively large amounts of data to be processed. These two facts raise a new challenge in constructing optimized execution plans for data processing. Traditional query optimizers focus their effort on reducing the number of IO requests to external memory, and do not take into account these considerations, and therefore produce suboptimal solutions. In their work, Shen et al. [SDNR07] have proposed a cost-based optimizer that indeed takes into account a special considerations for queries that involve text analytics functions. Apart from that, there have been works who focused on User-Defined Functions (UDFs) in order to create more optimized dataflows [CR10, HPS+12, WLMO11, RHH+15]. For example, in Sofa [RHH+15] each UDF is characterized by a set of properties it satisfies; then, by using a small set of rewrite rules, Sofa rearranges the dataflow according to some cost function.

Another approach for achieving optimized execution plans is by parallelism. The increased interest in large-scale analytical dataflows has given rise to cluster computing systems like Spark [ZXW+16] and Shark [XRZ+13]. To study such systems, Koutris and Suciu introduced the Massively Parallel Communication model (MPC) [KS11] where computations proceed in a sequence of parallel steps, each followed by a global synchronization step of all servers. Particularly, they showed that a query $Q$ can be evaluated by reshuffling the data over many servers, according to some distribution policy, and then computing $Q$ at each server in a parallel but communication-free manner. Ameloot et al. [AGK+17] have introduced a correctness condition, called
parallel-correctness, for the evaluation of queries w.r.t. distribution policy, which states that central execution always equals distributed execution, that is, equals to the union of the evaluations of the query at each server under the given distribution policy.

Even though an approach of UDF analysis for constructing optimized dataflows can be incorporated into our framework, it is left to future work, and we take an approach that is more similar to that of Ameloot et al. [AGK+17] where the focus is on the data. The basic idea here is to split large chunks of textual data into smaller ones in order to process each concurrently, while reducing the load of processing each chink by a single machine. The latter statement is particularly true for tools that are sensitive to the total document length. For instance, this applies to many coreference resolvers, as well as to constituency parsing. Naturally, this leads to the following question: Can we split the data, and processes each part separately, and still obtain the same results as if we had processed the data without splitting it? If so, under what split policy? We refer to this problem by the term consistent query plan, and present it in Chapter 4 with greater detail.
Chapter 3

Spannerlog

In this chapter we formally define the data model and query language of Spannerlog. We first begin with basic definitions and present the notion of IE-functions that act as a generalization of spanners. We then continue by defining the syntax and afterwards the semantics of our query language. We conclude this chapter by providing some results of a preliminary theoretical analysis on Spannerlog.

**Note:** throughout this document we frequently refer to the work by Fagin et al. [FKRV15] where a more basic framework has been introduced. For legibility and brevity purposes, we shall henceforth refer to it as the basic spanner framework, the basic framework, the basic spanner theory, or simply Fagin et al. (without the reference itself) interchangeably.

### 3.1 Preliminaries

**Strings and Spans.** We fix a finite alphabet $\Sigma$ of symbols. We denote by $\Sigma^*$ the set of all finite strings over $\Sigma$, and by $\Sigma^+$ the set of all finite strings of length at least one over $\Sigma$. We denote by $\epsilon$ the empty string. A *language* over $\Sigma$ is a subset of $\Sigma^*$. Throughout this document we will use the bald English letter ‘s’ with a bar over it (i.e., $\bar{s}$) to denote strings. Let $\bar{s} = \sigma_1\cdots\sigma_n$ be a string where $\sigma_1, \ldots, \sigma_n \in \Sigma$. The length $n$ of $\bar{s}$ is denoted by $|\bar{s}|$. A *span* points to a substring of a some string. Formally, a span has the form $[i,j]$, where $i, j \in \{1, 2, \ldots\}$, and $1 \leq i \leq j$. We will use the bald English letter ‘p’ (i.e., $p$) to denote spans. Let $p = [i,j]$ be a span. If $j \leq n + 1$ then we say that the span $p$ can be applied to $\bar{s}$, or alternatively, $p$ is a span of $\bar{s}$. The expressions $\bar{s}_p$ and $\bar{s}_{[i,j]}$ are used to denote the substring $\sigma_i\cdots\sigma_{j-1}$. In case $j > n + 1$ then we can define that $\bar{s}_{[i,j]}$ is $\bar{s}_{[i,n+1]}$, although this is not important for the purposes of this thesis. Similarly, in case $i \geq n + 1$ then we can define that $\bar{s}_{[i,j]}$ is $\epsilon$, the empty string. Note that for every $i \in \{1, \ldots, n\}$ it is the case that $\bar{s}_{[i,i]}$ is $\epsilon$, $\bar{s}_{[1,n+1]}$ is $\bar{s}$. The spans $[i,j]$ and $[i',j']$ are equal if and only if $i = i'$ and $j = j'$. We denote by $\text{Spans}$ the set of all possible spans, and by $\text{Spans}(\bar{s})$ the set of all the spans of $\bar{s}$. Note that here we deviate from the definition given in the basic spanner framework in that we allow spans not to be associated with specific
strings. In the end of Section 3.4, where we define the semantics of our framework, we elaborate more on this point. Two spans \([i, j]\) and \([i', j']\) of \(\mathcal{S}\) overlap if they share at least one index, that is \(i \leq i' < j\) or \(i' \leq i < j'\), and are disjoint otherwise. Finally, \([i, j]\) contains \([i', j']\) if \(i \leq i' \leq j' \leq j\).

Example 3.1.1. In all the examples throughout this document we consider the example alphabet \(\Sigma\) which consists of all the English letters and the standard punctuation marks. We may use the symbol ‘.’ to represent whitespace between words. Consider the following string \(\mathcal{S}\), where underneath each character appears its index:

\[
\text{the fast lane}
\]

For the spans \(p_1 = [1, 4]\) and \(p_2 = [10, 14]\) we have \(\mathcal{S}_{p_1} = \text{the}\) and \(\mathcal{S}_{p_2} = \text{lane}\).

Regular Expressions. Regular expressions over \(\Sigma\) are defined by the language

\[
\omega := \emptyset \mid \varepsilon \mid \sigma \mid \omega \lor \omega \mid \omega \cdot \omega \mid \omega^*
\]

where \(\emptyset\) is the empty set, \(\varepsilon\) is the empty string, and \(\sigma \in \Sigma\). Note that “\(\lor\)” is the disjunction operator, “\(^*\)” is the concatenation operator, and “\(^*\)” is the Kleene-star operator. By abuse of notation, if \(\Sigma = \{\sigma_1, \ldots, \sigma_k\}\), then we use \(\Sigma\) itself as an abbreviation of the regular expression \(\sigma_1 \lor \cdots \lor \sigma_k\). The language recognized by a regular expression \(\omega\) (i.e., the set of strings \(\mathcal{S} \in \Sigma^*\) that \(\omega\) matches) is denoted by \(L(\omega)\). A language \(L\) over \(\Sigma\) is regular if \(L = L(\omega)\) for some regular expression \(\omega\).

Spannerlog Relations. We assume given two disjoint sets \(V_{\text{str}}\) and \(V_{\text{spn}}\) of string variables and span variables, respectively, which may be assigned strings and spans (respectively). Throughout this document we shall use letters from the end of the English alphabet to denote variables (e.g., \(x, y, z\)). In case we can be certain the variables are string variables, we add a bar over them (e.g., \(\bar{x}\)). For a variable \(\bar{x} \in V_{\text{str}}\) we denote by \(\text{dom}(\bar{x})\) the set \(\Sigma^*\); similarly, for \(x \in V_{\text{spn}}\) we denote by \(\text{dom}(x)\) the set Spans. A relation schema is a finite sequence \(\mathbf{x} = x_1, \ldots, x_n\) of variables in \(V_{\text{str}} \cup V_{\text{spn}}\) that does not contain repetitions. In this context, we may refer to variables as attributes. A tuple over \(\mathbf{x}\) is a function \(\overline{\mathbf{t}}\) that assigns to each \(x_i \in \mathbf{x}\) a value in \(\text{dom}(x_i)\). For convenience, we shall often refer to \(\overline{\mathbf{t}}\) as the sequence \(\overline{\mathbf{t}}(\mathbf{x}) = \overline{\mathbf{t}}(x_1), \ldots, \overline{\mathbf{t}}(x_n)\). Additionally, note the dot above the letter ‘\(t\)’ that helps to distinguish tuples from spans. A Spannerlog relation, or simply a relation, over \(\mathbf{x}\) is a finite set of tuples over \(\mathbf{x}\), and is associated with a relation name (or symbol), which is commonly used to refer to it. We use the terms relation and table interchangeably. Let \(R\) be relation over some sequence \(\mathbf{x}\) of variables. We denote by \(\text{sch}(R)\) the relation schema of \(R\). Note that \(\text{sch}(R) = \mathbf{x}\). A (database) schema \(\mathcal{S}\) is a function that assigns a relation schema \(\mathcal{S}(R)\) to each relation name \(R\) in a finite set of relation names, denoted by \(\text{dom}(\mathcal{S})\). A (database) instance \(I\) over schema \(\mathcal{S}\) is a function that maps each relation name \(R \in \text{dom}(\mathcal{S})\) to a relation over
A relation $R$ is called a span relation (resp. string relation) if $\text{sch}(R)$ contains only span variables (resp. string variables). A relation $R$ is Boolean if $\text{sch}(R) = \emptyset$. In that case, $\text{true}$ denotes that $R$ consists of the empty tuple, and $\text{false}$ denotes that $R = \emptyset$.

### 3.2 Document Spanners and IE-Functions

**IE Functions.** An information extraction function (or IE function for short) is a function $F$ that is associated with a relation schema denoted by $\text{sch}(F)$. The IE function $F$ maps a string $\bar{s} \in \Sigma^*$ to a relation over $\text{sch}(F)$. 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. We illustrate this by means of an example.

**Example 3.2.1.** Let $\text{pos}$ be an IE function where $\text{sch}(\text{pos})$ consists of two attributes: a span attribute and a string attribute. Let $\bar{s}$ be a string. A tuple in $\text{pos}(\bar{s})$ is of the form $(p, t)$, where $p$ is a span of some token (word) in $\bar{s}$, and $t$ is the corresponding part-of-speech for that token. For instance, applying $\text{pos}$ on the string $\bar{s}$ described in Example 3.1.1 should yield the relation $\{(\left[1, 4\right], \text{DT}), (\left[5, 9\right], \text{JJ}), (\left[10, 14\right], \text{NN})\}$, where ‘DT’, ‘JJ’ and ‘NN’ stand for determiner, adjective and noun, respectively. State-of-the-art NLP tools, such as the Stanford CoreNLP library [MSB+14], can be used as implementations of IE-functions that represent standard NLP algorithms, as in the IE-function $\text{pos}$ that has the functionality of a PoS tagger.

**Document Spanners.** By the definition given by the basic framework, a document spanner (or just spanner for short), is a function that is associated with a finite set of (span) variables that maps every string to a span relation over the same variables. Observe that spanners form a subset of the set of IE functions. More specifically, a spanner $P$ is an IE function such that:

- $\text{sch}(P)$ consists only of span variables. That is, the relations returned by $P$ contain only spans, and not strings.
- For each $\bar{s} \in \Sigma^*$ and for every span $p$ occurring (as a value) in $P(\bar{s})$, the span $p$ is a span of $\bar{s}$; that is, can be applied to $\bar{s}$.

**Spanner Representation Systems.** A spanner representation system refers to any manner of specifying spanners through finite objects. Fagin et al. defined several representation systems by means of regular expressions, special types of automata, and relational algebra. Here we only provide the definition of the regex formula system, as it will be used throughout this work. A regular expression with capture variables, or just variable regex for short, is an expression of the following syntax that extends that of regular expressions:
\[ \omega := \emptyset \mid \epsilon \mid \sigma \mid \omega \lor \omega \mid \omega \cdot \omega \mid \omega^* \mid x\{\omega} \]

The added alternative is \( x\{\omega} \) where \( x \in \text{Spans} \). A variable regex can be matched against a string in multiple ways, or more formally, there can be multiple parse trees showing that a string matches a variable regex. Each such a parse tree naturally associates variables with spans. It is possible, however, that in a parse tree a variable is not associated with any span, or is associated with multiple spans. If every variable is associated with precisely one span, then the parse tree is said to be functional. A variable regex is called a \textit{regex formula} if it has only functional parse trees on every input string. An example of a variable regex that is not a regex formula is \((x\{a\})^*\), because a match against \( aa \) assigns \( x \) to two spans simultaneously. That is, the variable \( x \) is associated with more than one span (these are the spans of \( a \) and \( aa \)), and therefore the corresponding parse tree is not functional. We refer the reader to the work of Fagin et al. [FKRV15] for the full formal definition of regex formulas. By \text{RGX} we denote the class of regex formulas. A regex formula is naturally viewed as representing a spanner, and by \( [\omega] \) we denote the spanner that is represented by \( \omega \). In accordance with our definition of IE functions, 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. Following is an example of a spanner represented as a regex formula.

\textbf{Example 3.2.2.} Consider the following regex formula:

\[ \omega_1(x,y,z) := (\Sigma^* \cdot \omega)^* \cdot z\{\omega_{cap}\} \cdot \omega \cdot y\{\omega_{cap}\} \cdot (\omega \cdot \Sigma^*)^* \]

Here, \( x, y \) and \( z \) are span variables, and \( \omega_{cap} \) is the regular expression \((A \lor \ldots \lor Z) \cdot (a \lor \ldots \lor z)^*\). The spanner represented by \( \omega_1 \) 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.

A spanner representation system like \text{RGX} can be extended with algebraic operator symbols to form a spanner algebra. Formally, if \( \text{SR} \) is a class of spanner representations and \( O \) is a spanner algebra, then \( \text{SR}^O \) denotes the class of all the spanner representations defined by applying (compositions of) the operators in \( O \) to the representations in \( \text{SR} \). In other words, \( \text{SR}^O \) is the closure of \( \text{SR} \) under \( O \) (when \( O \) is taken as a set of operator symbols); consequently, \([\text{SR}^O] \) is the closure of \([\text{SR}] \) under \( O \). With this notation, we briefly describe two other spanner representation systems defined by Fagin et al. that we consider in later chapters:

\begin{itemize}
  \item \textbf{Regular Spanners.} A \textit{regular spanner} is a spanner that can be expressed in the closure of the regex formulas under the relational algebra. More formally, the
  \end{itemize}
class of regular spanners, denoted by \( \text{REG} \), is defined as \( \text{RGX} \{\cup, \pi, \bowtie, \setminus\} \). That is, it is the class of expressions in the closure of \( \text{RGX} \) under union (\( \cup \)), projection (\( \pi_x \), where \( x \) is a sequence of variables), natural join (\( \bowtie \)) and difference (\( \setminus \)). Note that the natural join is based on span equality, and not string equality. A spanner is regular if it is definable in \( \text{REG} \).

- **Core Spanners.** Core spanners extend the algebra of regex formulas with the string-equality selection, denoted by \( \varsigma \). That is, given an expression \( \omega \) in \( \text{REG} \) and two span variables \( x,y \) occurring in \( \omega \), the spanner defined by \( \varsigma = x,y(\omega) \) selects all those tuples from \( \omega \) in which \( x \) and \( y \) span equal strings (though \( x \) and \( y \) can be different spans). The class of core spanners, denoted by \( \text{Core} \), is defined by \( \text{RGX} \{\cup, \pi, \bowtie, \varsigma\} \). That is, it is the closure of \( \text{RGX} \) under union, projection, natural join and string-equality selection.

Note that we may extend the use of the term regex formulas to refer to expressions in \( \text{REG} \) and \( \text{Core} \), in addition to expressions in \( \text{RGX} \).

**Example 3.2.3.** Let \( \omega_{12} \) be the regex formula that captures all spans \( x_1 \) and \( x_2 \) such that \( x_1 \) ends before \( x_2 \) begins; that is:

\[
\omega_{12}(x_1, x_2) := \Sigma^* \cdot x_1 \{\Sigma^*\} \cdot x_2 \{\Sigma^*\} \cdot \Sigma^*
\]

Let us use \( \omega_1 \) from Example 3.2.2 to define the following core spanner:

\[
\omega_2(x_1, x_2) := \left(\varsigma_{y_1,y_2}(\omega_1(x_1, y_1, z_1) \bowtie \omega_1(x_2, y_2, z_2) \bowtie \omega_{12}(x_1, x_2))\right)
\]

Observe that \( \omega_2 \) selects all the spans \( x_1 \) and \( x_2 \) that occur in tuples of \( \omega_1 \), such that the corresponding \( y_1 \) and \( y_2 \) delimit the same substrings (even though \( y_1 \) and \( y_2 \) themselves are not required to be equal as spans), and moreover, \( x_1 \) ends before \( x_2 \) begins. \( \square \)

**Boolean IE Functions.** Lastly, we highlight a special type of IE functions. An IE function \( F \) is called Boolean if \( \text{sch}(F) = \emptyset \). In that case, for a given string \( \bar{s} \), \( F(\bar{s}) = \text{true} \) denotes that \( F(\bar{s}) \) consists of the empty tuple, and \( F(\bar{s}) = \text{false} \) denotes that \( F(\bar{s}) = \emptyset \). Note that in this case \( F \) is also a spanner. If \( F \) is Boolean, then we say that \( F \) recognizes the language of strings that evaluate to \text{true}. Furthermore, we say that a language \( L \) over \( \Sigma \) is recognizable by \( \text{RGX} \), \( \text{REG} \) or \( \text{Core} \) if there exists a Boolean spanner \( P \) in \( \text{RGX} \), \( \text{REG} \) or \( \text{Core} \), respectively, such that for each \( \bar{s} \in \Sigma^* \), it holds that \( P(\bar{s}) = \text{true} \) if and only if \( \bar{s} \in L \).

### 3.3 Syntax

For convenience, and with a slight abuse of the terminology, we use the name Spannerlog to refer to the data model we presented above, and to the query language that is
presented next. The query language Spannerlog is similar to Datalog \cite{AHV95}, but possesses two notable modifications:

- Given a string \( s \) and a span \( p \) of \( s \), we can construct the substring of \( s \) spanned by \( p \), denoted by \( s_p \).
- We can call IE functions as subroutines.

Spannerlog is parameterized by a class \( C \) of IE functions that can be called. We use the notation Spannerlog\( (C) \) to specify the \( C \) parameter, but we may not explicitly do so if \( C \) is clear from the context or irrelevant. Let \( C \) be a class of IE functions. The syntax of the query language Spannerlog\( (C) \) is formally defined as follows.

**Terms.** We use two types of terms:

- A **span term** is either a constant span \([i,j] \in \text{Spans}\) or a span variable \( x \in V_{\text{spn}} \).
- A **string term** is inductively defined as follows.
  
  - Every constant string \( s \in \Sigma^* \) and every string variable \( \bar{x} \in V_{\text{str}} \) is a string term.
  
  - If \( s \) is a string term and \( p \) is a span term, then \( s_p \) is a string term.

We shall use letters from the beginning of the Greek alphabet to denote terms (e.g., \( \alpha, \beta, \gamma \)). As with variables, in case we can be certain the terms are string terms, then we add a bar over them (e.g., \( \bar{\alpha} \)).

**Atomic Formulas.** Let \( S \) be a schema. Let \( R \) be a \( k \)-ary relation symbol in \( \text{dom}(S) \), and let \( F \) be an IE function in \( C \). The sequence \( \alpha_1, \ldots, \alpha_k \) of terms is properly typed for \( R \) (resp. \( F \)) if each \( \alpha_i \), \( 1 \leq i \leq k \), is (1) a string term if the \( i \)-th attribute of \( S(R) \) (resp. \( \text{sch}(F) \)) is a string attribute, or (2) a span term if the \( i \)-th attribute of \( S(R) \) (resp. \( \text{sch}(F) \)) is a span attribute.

An atomic formula is either one of the two following forms:

- A **DB-atom** over \( S \) has the form \( R(\alpha_1, \ldots, \alpha_k) \) where \( R \in \text{dom}(S) \) and \( \alpha_1, \ldots, \alpha_k \) are properly typed for \( R \).
- An **IE-atom** over \( C \) has the form \( F(\bar{\alpha})(\beta_1, \ldots, \beta_k) \) where \( F \in C \), \( \bar{\alpha} \) is a string term, and \( \beta_1, \ldots, \beta_k \) are properly typed for \( F \).

Before proceeding, let us consider again the IE functions we encountered in Examples 3.2.1 and 3.2.2.

**Example 3.3.1.** Let us denote by CoreNLP the IE functions class of the NLP tools provided by the Standford’s CoreNLP library. The PoS tagger \( \text{pos} \) we defined in Example 3.2.1 is an IE function in CoreNLP if it is implemented by the CoreNLP library. The corresponding IE-atom is given by \( \text{pos}(\bar{\alpha})(\beta, \bar{\gamma}) \), where \( \bar{\alpha} \) and \( \bar{\gamma} \) are string terms, and \( \beta \) is a span term. \( \square \)
Example 3.3.2. Recall the regex formula $\omega_1$ we presented in Example 3.2.2. The corresponding IE-atom of $\omega_1$ is given by $\omega_1(\bar{\alpha})(x, y, z)$. For convenience, we can define the same IE-atom directly by the following expression.

$$\text{RGX}(\bar{\alpha})\left[\left(\Sigma^* \cdot \omega_1 \cdot \omega \cdot \omega_1\right)\cdot \left(\omega \cdot \Sigma^*\right)^*\right]$$

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 $\text{dom}(\mathcal{I})$ and $\text{dom}(\mathcal{E})$ are disjoint. A rule is an expression of the form

$$\varphi \leftarrow \psi_1, \ldots, \psi_m,$$

where $\varphi$ is a DB-atom over $\mathcal{I}$ and each $\psi_i$ is either a DB-atom (over $\mathcal{I}$ or $\mathcal{E}$) or an IE-atom. We call $\varphi$ the head and $\psi_1, \ldots, \psi_m$ the body. Let $x$ be a variable, $\rho$ be a rule, and $\psi$ be an atom in the body of $\rho$ of the form $R(\beta_1, \ldots, \beta_k)$ if $\psi$ is a DB-atom, or of the form $F(\bar{\alpha})(\beta_1, \ldots, \beta_k)$ if $\psi$ is an IE-atom. We say that $x$ is bound by $\psi$ if $x$ is one of the $\beta$s. A rule is safe if (1) every variable in its head occurs at least once in its body, and (2) for every variable in its body is bounded by a the atom appearing in the body.

Example 3.3.3. The rule $R(x, y) \leftarrow S(s), \text{RGX}(\bar{s}_y)[x\{a^*\} \cdot b]$ is unsafe (i.e., not safe) because the variable $y$ is not bounded by any body atom. However, the rule $R(x, y) \leftarrow S(s), T(y), \text{RGX}(\bar{s}_y)[x\{a^*\} \cdot b]$ is safe because here the variable $y$ is bounded by the body atom $T(y)$.

Programs. Let $\mathcal{I}$ and $\mathcal{E}$ be defined as previously. A program over Spannerlog($\mathcal{C}$) is a set of safe rules. Let $\mathcal{P}$ be a program. The extensional schema of $\mathcal{P}$, denoted by $\text{edb}(\mathcal{P})$, is $\mathcal{E}$. Similarly, the intensional schema of $\mathcal{P}$, denoted by $\text{idb}(\mathcal{P})$, is $\mathcal{I}$. The schema of $\mathcal{P}$, denoted by $\text{sch}(\mathcal{P})$, is $\text{edb}(\mathcal{P}) \cup \text{idb}(\mathcal{P})$. Informally, the semantics of a program is a mapping of database instances over $\text{edb}(\mathcal{P})$ to database instances over $\text{edb}(\mathcal{P}) \cup \text{idb}(\mathcal{P})$. We may refer to the query language Spannerlog($\mathcal{C}$) as the set of all programs over Spannerlog($\mathcal{C}$). Next we provide a formal definition of the semantics we consider in our framework. Figure 3.1(b) shows an example of a program. We shall explain it in detail later on.

3.4 Semantics

In order to later analyze Spannerlog in a theoretical manner, we establish a link to logic-programming [AHV95] 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, $\bar{\alpha}$ be a string term, $\beta$ be a span
term, and $\gamma$ be a term (string or span). The grounding function of $v$, denoted by $\tilde{v}$, is defined inductively as follows.

$$\tilde{v}(\gamma) = \begin{cases} 
\gamma & \text{if } \gamma \in \Sigma^* \cup \text{Spans} \\
v(\gamma) & \text{if } \gamma \in V_{\text{str}} \cup V_{\text{spn}} \\
\tilde{v}(\bar{\alpha})\tilde{v}(\beta) & \text{if } \gamma \text{ is of the form } \bar{\alpha}_\beta 
\end{cases}$$

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 $\alpha$ if (1) $\alpha$ is either a constant or a variable, or (2) $\alpha$ has the form $\bar{\alpha}_\beta$ where $\beta$ is a span term.

Let $\alpha = \alpha_1, \ldots, \alpha_k$ be a sequence of terms. The grounding of $\alpha$ by a valuation function $v$, denoted by $\tilde{v}(\alpha)$, is the sequence $\tilde{v}(\alpha_1), \ldots, \tilde{v}(\alpha_k)$. Let $S$ be a schema, $I$ be an instance over $S$, and $R$ be a relation name in $\text{dom}(S)$. We say that the DB-atom $R(\alpha)$ is satisfied under $v$ by $I$, denoted by $I \models_v R(\alpha)$, if (1) $\tilde{v}$ is valid for every $\alpha_i \in \alpha$, and (2) $\tilde{v}(\alpha) \in I(R)$. Let $C$ be a class of IE functions, and $F$ be an IE function in $C$. We say that the IE-atom $F(\bar{\beta})(\alpha)$ is satisfied under $v$ by $\bar{\beta}$, denoted by $\tilde{\beta} \models_v (F, \alpha)$, if (1) $\tilde{v}$ is valid for $\bar{\beta}$ and for every $\alpha_i \in \alpha$, and (2) $\tilde{v}(\alpha) \in F(\bar{v}(\bar{\beta}))$.

Let $P$ be a program in Spannerlog($C$), $\rho$ be a rule in $P$ of the form $\phi \leftarrow \psi_1, \ldots, \psi_m$, and $I$ be an instance over $\text{sch}(P)$. We say that $I$ satisfies $\rho$ by $v$, denoted by $I \models_v \rho$, if it holds that if (1) for each DB-atom $R$ in the body of $\rho$ it holds that $I \models_v R$, and (2) for each IE-atom $F(\bar{\beta})(\alpha)$ in $\rho$ it holds that $\tilde{\beta} \models_v (F, \alpha)$, then it holds that $I \models_v \phi$. $I$ is called a model of $P$ if for every $\rho \in P$ and every valuation $v$ it holds that $I \models_v \rho$. Let $I^E$ be an instance over $E$. As in the case of traditional Datalog, it is easy to verify that the intersection of all the models of $P$ is a model by itself, and also the minimal one w.r.t. set inclusion. As in traditional Datalog, we take this model as the semantics of Spannerlog programs. More formally, the semantics of the program $P$ on input $I^E$, denoted by $P(I^E)$, is the minimum model of $P$ containing $I^E$, if it exists. In some contexts, we may refer to $P(I^E)$ as the output of $P$.

**Example 3.4.1.** Figure 3.1 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 $\text{Review}$, given in (a), in which there are four attributes: $\text{text}$ (the review body), $\text{bid}$ (business ID), $\text{uid}$ (user ID) and $\text{rid}$ (review ID). For each review in $\text{Review}$, 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 ($\text{great food}$), but expresses a negative sentiment towards the price ($\text{exceptionally expensive price}$).

The program in (b) consists of two rules. The IE-atom $\text{dep}(\bar{t})(x, y)$ in the first rule
Great food, and excellent wine, but for an exceptionally expensive price.

The best pancake I have ever eaten.

(a)

\[
\text{Signal}(\bar{r}, x, y) \leftarrow \text{Review}(\bar{t}, \bar{b}, \bar{u}, \bar{r}), \ \text{dep}(\bar{t})(x, y), \ \text{pos}(\bar{t})(y, \text{JJ}).
\]

\[
\text{Sentiment}(\bar{r}, \bar{b}, \bar{t}_{x}, \bar{t}) \leftarrow \text{Review}(\bar{t}, \bar{b}, \bar{u}, \bar{r}), \ \text{Signal}(\bar{r}, x, y), \ \text{sentiment}(\bar{t}_{y})(\bar{t}).
\]

(b)

<table>
<thead>
<tr>
<th>Sentiment</th>
<th>rid</th>
<th>bid</th>
<th>entity</th>
<th>class</th>
</tr>
</thead>
<tbody>
<tr>
<td>301</td>
<td>101</td>
<td>food</td>
<td>positive</td>
<td></td>
</tr>
<tr>
<td>301</td>
<td>101</td>
<td>wine</td>
<td>positive</td>
<td></td>
</tr>
<tr>
<td>301</td>
<td>101</td>
<td>price</td>
<td>negative</td>
<td></td>
</tr>
<tr>
<td>302</td>
<td>102</td>
<td>pancake</td>
<td>positive</td>
<td></td>
</tr>
</tbody>
</table>

(c)

Figure 3.1: A run of a Spannerlog program.

extracts a dependency parse tree of \( \bar{t} \), 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 one can deduce that the sentiment in the review towards the food is positive. We already encountered the IE-atom \( \text{pos}(\bar{t})(y, \text{JJ}) \) in Example 3.3.1.

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 utilize a Sentiment Analysis tool to determine the sentiment for each entity; this is performed by the IE-atom \( \text{sentiment}(\bar{t}_{y})(\bar{t}) \). The output of the program is shown in (c).

We point out a distinction between the span semantics in Spannerlog to that used in the basic spanner framework. In Section 3.1 we defined a span \( p \) to be simply an interval specified by bounding indices, and we allowed \( p \) to be applied to some string \( \bar{s} \) to extract a substring of \( \bar{s} \). Therefore, a span is string independent in that it can be applied to every string, whereas in the basic framework a span is defined similarly, but is also associated with the string from which it was originated, and can only be applied to that string; thus we can consider this type of spans 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 the basic framework. 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
3.5 Expressiveness

In this section we provide preliminary results obtained by analyzing Spannerlog. Since our framework is built on the formalism of spanners, our initial analysis involves comparing Spannerlog to spanners and their representations as they were defined in the basic spanner theory. Interestingly, Peterfreund et al. [PCFK17] have shown that if we allow stratified negation [AHV95] in programs over (a restricted class of) Spannerlog(RGX) then we can precisely express all spanners computable in polynomial time. However, we leave this discussion out of this thesis’ scope. Here we shall focus only on lower bounds. Namely, what one can express using Spannerlog, rather than what one cannot express.

Before proceeding, we define what it means for a language (i.e., a subset of \( \Sigma^* \)) to be recognized by a Spannerlog program. This will enable us to discuss the expressive power of Spannerlog compared to the different spanner classes that were introduced earlier w.r.t. the notion of language recognition by spanners (see Section 3.2).

Let \( P \) be a program where \( edb(P) \) consists of only the relation name \( Doc \) such that the relation schema \( edb(P)(Doc) \) has a single string attribute. Let \( Out \) be a relation name in \( idb(P) \) such that \( sch(Out) = \emptyset \) (i.e., \( Out \) is Boolean). We say that \( L \) is recognizable by \( P \), if for every \( s \in \Sigma^* \) it holds that for the instance \( I \) over \( edb(P) \), in which \( I(Doc) \) consists only of \( s \), \( P(I)(Out) = true \) if and only if \( s \in L \). Recall from Section 3.4 that \( P(I) \) is an instance over the intensional schema of \( P \), and from Section 3.1 that an instance is a mapping from relation names to relations, thus the expression \( P(I)(Out) \) is the relation \( Out \) in the output of \( P \) on \( I \). 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. Additionally, we say a spanner \( P \) is definable by Spannerlog(\( \mathcal{C} \)) if there exists a program \( P \in \text{Spannerlog}(\mathcal{C}) \) such that (1) \( P \) is defined as before with the exception that \( sch(Out) = sch(P) \), and (2) for every \( s \in \Sigma^* \) it holds that for the instance \( I \) over \( edb(P) \), in which \( I(Doc) \) consists only of \( s \), \( P(I)(Out) = P(s) \).

Example 3.5.1 is given to clarify the above definitions.

**Example 3.5.1.** Consider the program \( P \) over Spannerlog(RGX) consisting of the following rule:

\[
Out(\bar{s}) \leftarrow Doc(\bar{s}), \ RGX(\bar{s})[a^* \cdot b]
\]

For the instance \( I_1 \) that maps the relation \( Doc \) to \( \{ (aab) \} \) it holds that \( P(I_1)(Out) = true \), whereas for the instance \( I_2 \) that maps the relation \( Doc \) to \( \{ (aabb) \} \) it holds that \( P(I_2)(Out) = false \). Furthermore, it is easy to verify that the spanner \( P \) defined by the regex formula \( a^* \cdot b \) is definable by \( P \). \( \square \)

As in Datalog, the algebraic operators union, projection and natural join can be expressed in any Spannerlog program. This leads to the following proposition:
Proposition 3.5.2. Every spanner in \( \text{REG} \) is definable by a program over non-recursive \( \text{Spannerlog}(\text{RGX}) \).

The proof of this proposition is quite straightforward, as was also mentioned for a similar proposition in a subsequent work on spanners [FKRV14] (Proposition 4.1). The basic idea is the following: for a program in \( \text{Spannerlog}(\text{REG}) \) we replace any rule that includes regex formulas over \( \text{REG} \) with matching rules with regex formulas over \( \text{RGX} \), where the algebraic operations (union, projection and natural join) of the regexes over \( \text{REG} \) are expressed by the usual way they are expressed in Datalog.

3.5.1 Span Semantics

The following program over Boolean \( \text{Spannerlog}(\text{RGX}) \) illustrates the flexibility we gain by adopting the independent spans semantics as was discussed at the end of Section 3.4.

\[
\begin{align*}
\text{Out} & \leftarrow \text{Doc}(\bar{s}), & (3.1) \\
\text{RGX}(\bar{s})[x\{a^*\} \cdot y\{b^*\}], & (3.2) \\
\text{RGX}(\bar{s}_y)[x\{b^*\}] & (3.3)
\end{align*}
\]

The program in lines (3.1) to (3.3) consists of a single rule. The IE-atom \( \text{RGX}(\bar{s})[x\{a^*\} \cdot y\{b^*\}] \) demands the span variables \( x \) and \( y \) to be grounded to spans that span substrings of \( \bar{s} \) of the form \( a^* \) and \( b^* \), respectively. In the last atom \( \text{RGX}(\bar{s}_y)[x\{b^*\}] \), \( x \) is matched against the substring of \( \bar{s} \) that contains only \( b \)'s. Note that under the dependent spans semantics, the program is malformed: \( x \) occurs in both the IE-atoms, any valuation function that satisfies the first two atoms can only satisfy the third atom if \( \bar{s} \) has as many \( b \)'s as \( a \)'s. From here we conclude that the above program recognizes the non-regular Context-Free Language (CFL) \( \mathcal{L}_1 = \{a^n b^n \mid n \in \mathbb{N}\} \).

The following proposition states that programs over Boolean \( \text{Spannerlog}(\text{RGX}) \) can recognize also non-CFLs:

Proposition 3.5.3. There exists a language that is not context-free, and yet, is recognizable by a program over non-recursive Boolean Spannerlog(\( \text{RGX} \)).

Proof of Proposition 3.5.3. Consider the following program:

\[
\begin{align*}
\text{Out} & \leftarrow \text{Doc}(\bar{s}), & (3.4) \\
\text{RGX}(\bar{s})[x\{a^*\} \cdot y\{b^*\} \cdot z\{c^*\}], & (3.5) \\
\text{RGX}(\bar{s}_y)[x\{b^*\}], & (3.6) \\
\text{RGX}(\bar{s}_z)[x\{c^*\}] & (3.7)
\end{align*}
\]

This program is similar in concept to that appearing in lines (3.1) to (3.3). We leave it to the reader to verify that the program given in lines (3.4) to (3.7) recognizes language
Let \( L_2 = \{ a^n b^n c^n \mid n \in \mathbb{N} \} \) which is known to be non-CFL.

This leads us to the following proposition:

**Proposition 3.5.4.** the class of spanners definable by \( \text{Spannerlog}(RGX) \) strictly contains the class of regular spanners.

*Proof of Proposition 3.5.4.* The correctness follows from Proposition 3.5.2, Proposition 3.5.3, and from the observation that Boolean regular spanners can recognize only regular languages (as they can be represented by standard NFAs [FKRV15]).

### 3.5.2 String Equality

With core spanners we get string equality by applying the string-equality selection operator \((\varsigma^=)\). In the basic framework Fagin et al. have shown that string comparison cannot be expressed by a regular spanner. The following proposition states that it is possible to express string equality by a program over \( \text{Spannerlog}(RGX) \).

**Proposition 3.5.5.** String equality can be expressed in non-recursive \( \text{Spannerlog}(RGX) \).

*Proof of Proposition 3.5.5.* Consider the following program.

\[
\text{Equals}(\bar{s}_x, \bar{s}_x) \leftarrow \text{Doc}(\bar{s}), \text{RGX}(\bar{s})[\Sigma^* : x\{\Sigma^*\} \cdot \Sigma^*] \quad (3.8)
\]

The relation \( \text{Equals} \) consists of all the pairs \((\bar{t}, \bar{t})\) such that \( \bar{t} \) is a substring of \( \bar{s} \).

We can use \( \text{Equals} \) to recognize the non-CFL \( L_3 = \{ \bar{s} \cdot \bar{s} \mid \bar{s} \in \Sigma^* \} \) by adding the following rule.

\[
\text{Out()} \leftarrow \text{Doc}(\bar{s}), \text{RGX}(\bar{s})[x\{\Sigma^*\} \cdot y\{\Sigma^*\}], \text{Equals}(\bar{s}_x, \bar{s}_y) \quad (3.9)
\]

Note that the program given in lines (3.8) to (3.9) serves as an alternative proof for Proposition 3.5.3. All of the above leads us to the following theorem:

**Theorem 3.1.** The class of spanners definable in \( \text{Spannerlog}(RGX) \) is a strict superset of \( \text{Core} \).

*Proof of Theorem 3.1.* By Proposition 3.5.5 we are capable of expressing string equality using the \( \text{Equals} \) relation. Every occurrence of the string-equality selection can now be simulated by replacing it with this relation in a rather straightforward way (and therefore, and for the sake of brevity, the more formal argument is omitted here). Thus any core Spanner is definable by a program over \( \text{Spannerlog}(RGX) \). Together with Proposition 3.5.4 we have that the class of spanners definable in \( \text{Spannerlog}(RGX) \) is a superset of \( \text{Core} \) (recall that core spanners extend the regular spanners with the string-equality selection). Fagin et al. have shown that the non-regular language
\[ L_1 = \{a^nb^n \mid n \in \mathbb{N}\} \] is not recognizable by any Boolean core spanner (Theorem 4.21). However, we have shown that \( L_1 \) is recognizable by the program given in lines (3.4) to (3.7).

### 3.5.3 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 CFLs. Recall that a Context-Free Grammar (CFG) \( G \) is defined as a 4-tuple:

\[
G = (V, T, P, S)
\]

where:

1. \( V \) is a finite set. Elements in \( V \) are called nonterminals or variables.
2. \( T \) is a finite set, disjoint from \( V \). Elements in \( T \) are called terminals or characters.
3. \( P \) is a finite set of production rules, each of the form \( A \rightarrow \sigma \) where \( A \in V \) and \( \sigma \) is a finite sequence of symbols from \( V \cup T \).
4. \( S \in V \) is the start symbol.

By convention, capitalized letters denote nonterminals and Greek letters denote terminals. Given a sequence \( \varphi_1 \) of symbols in \( V \cup T \), we say that \( \varphi_1 \) reduces immediately to \( \varphi_2 \) by the rule \( A \rightarrow \sigma \) if there are two sequences \( \eta_1 \) and \( \eta_2 \) in \( V \cup T \) such that \( \varphi_1 = \eta_1 A \eta_2 \) and \( \varphi_2 = \eta_1 \sigma \eta_2 \). We say that \( \varphi_0 \) reduces to \( \varphi_n \) if for every index \( i \in \{1, 2, \ldots, n\} \), the sequence \( \varphi_{i-1} \) reduces immediately to \( \varphi_i \). A sequence, or a string, \( \bar{s} \in T \) is said to be recognized by \( G \) if the start symbol \( S \) reduces to \( \bar{s} \). The language produced by the grammar \( G \) is the set of all the strings that are recognized by \( G \), and denoted by \( \mathcal{L}(G) \). Finally, we remind that a language \( L \) is said to be context free if there exists a CFG \( G \) such that \( L = \mathcal{L}(G) \).

The following theorem formally states a connection between Spannerlog and CFLs:

**Theorem 3.2.** Any CFL is recognizable in Spannerlog(RGX).

In other words, Theorem 3.2 asserts that for every CFG \( G \) there exists a program \( P \) over Boolean Spannerlog(RGX) such that \( \mathcal{L}(G) \) is recognizable by \( P \).

**Proof of Theorem 3.2.** The following proof shows similarities to a reduction employed by Shmueli used to simulate decision problems for CFLs using Datalog [Shm93]. Let \( G \) be a CFG, 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 \( A \rightarrow \sigma \) or \( A \rightarrow BC \). We define a program \( P \) as follows. For every production rule of the form \( A \rightarrow \sigma \) in \( G \), we define the following rule in \( P \):

\[
A(\bar{s}_x) \leftarrow \text{Doc}(\bar{s}), \text{RGX}(\bar{s})[\Sigma^* \cdot x\{\sigma\} \cdot \Sigma^*]
\]
For every production rule of the form $A \rightarrow BC$ in $G$, we define the following rule in $P$.

$$A(\bar{s}_2) \leftarrow Doc(\bar{s}), \text{RGX}(\bar{s})\{\Sigma^* \cdot z\{\Sigma^*\} \cdot y\{\Sigma^*\} \cdot \Sigma^*\}, B(\bar{s}_2), C(\bar{s}_y)$$

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

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

where $S$ is the start symbol of $G$. \hfill \square

We conclude this discussion by noting that the other direction of the theorem does not hold. The context-sensitive language $L_2 = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ is recognizable by the program given in lines (3.4) to (3.7), but it is known that there does not exist a CFG $G$ such that $L(G) = L_2$ [HMU06].
Chapter 4

Data Splitting

As mentioned in Section 2.4, in this work we have been focusing on optimization by means of data manipulation. The idea in a nutshell is to first split the data, then run the program in a parallel manner on the split data, and finally collecting all the partial results. For this end, we dedicate this chapter to introduce the notion of split correctness that enables the construction of analyzable parallel execution plans of Spannerlog programs for optimization purposes. The main question we wish to explore here is how can we automatically construct a parallelized program correctly. That is, be assured that the optimized query plan is equivalent to the original one. Furthermore, we would like to know under which split policy we can obtain this assurance, and can we automatically deduce it only by statically analyzing the program. While this is an intriguing notion in its own right, here we are interested in split correctness in the context of Spannerlog. We shall see in the next chapter that this strategy can lead to a significant improvement in running time. We begin our discussion with several definitions, and then provide initial analysis based on them.

4.1 Preliminaries

Span Shift Operator. The span shift operator for two spans, denoted by $\gg$, is defined as follows:

$$[i_1, j_1] \gg [i_2, j_2] = [i_2 + i_1 - 1, j_2 + j_1 - 1]$$

Let $p$ be a span. Given a tuple $t = t_1, \ldots, t_n$, we define the operation $p \gg t$ as a mapping of $t$ and $p$ to a new tuple $t' = t'_1, \ldots, t'_n$ where $t'_i = p \gg t_i$ if $t_i$ is of type span, and $t'_i = t_i$ otherwise. Given a relation $R$, we define the operation of $p \gg R$ as a mapping of $p$ and $R$ to the relation $\{p \gg t \mid t \in R\}$. Example 4.1.1 tries to provide more intuition regarding the span shift operator, as it demonstrates a simple data manipulation that is in fact the basic idea we wish to leverage for performance purposes.

1At the time of writing this essay, an independent work on split correctness has gone underway and is currently still in progress.
Example 4.1.1. The string $s_1$ appearing in Figure 4.1(a) consists of two sentences over the alphabet $\Delta = \{a, b, \ldots\}$ where $a$ and $b$ represent English letters, ‘ ‘ a white space, and ‘.’ a punctuation mark that indicates the end of a sentence. The span $[9, 17]$ delimits the second sentence of $s_1$, also appearing as the string $s_2$ given in Figure 4.1(b). By shifting the span $[1, 5]$ using $[9, 17]$ we get the span $[9, 13]$. With notation:

$$[9, 17) \gg [1, 5) = [9, 13]$$

Observe that applying the span $[1, 5]$ to $s_2$ would yield the string $bba$. Applying the shifted span $[9, 13]$ on the original string would result in the same string. □

Splitters. We call a spanner $L$ splitter if $sch(L)$ consists of a single (span) variable. The class $\text{SPLITTER}$ is defined as the set of all splitters in $\text{RGX}$. This can more formally be expressed by

$$\text{SPLITTER} \overset{\text{def}}{=} \{\omega \mid \omega \in \text{RGX}, |sch(\omega)| = 1\}$$

Example 4.1.2 shows how we use a splitter for string processing. Note that in spite of what the name implies, a splitter does not necessarily splits its input string, as it is not forced to do so by the above definition. Indeed it is possible for a splitter to produce overlapping spans, and furthermore, not to cover the entire string by the spans it produces.

Example 4.1.2. Consider the following regex formula:

$$\omega_3 := \Delta^* y \{(a|b)^*\} \cdot \Delta^*$$

Applying $\omega_3$ to $s_2$ would result in the following relation:

<table>
<thead>
<tr>
<th>$[<a href="s_2">\omega_3</a>]$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1, 5]</td>
<td></td>
</tr>
<tr>
<td>[6, 8]</td>
<td></td>
</tr>
<tr>
<td>$\vdots$</td>
<td></td>
</tr>
</tbody>
</table>

Shifting this relation by using the span $[9, 17]$ will result in the following:
\begin{align*}
[9, 17) & \succ \begin{array}{c}
\{y\} \\
\{1, 5\} \\
\{6, 8\} \\
\vdots
\end{array}
= \begin{array}{c}
\{y\} \\
\{9, 13\} \\
\{14, 16\} \\
\vdots
\end{array}
\end{align*}

Notice that applying the spans in the resulted relation to \(s_1\) would yield the same strings as applying the spans in the original relation to \(s_2\). \(\square\)

### 4.2 Split Correctness

In this section we introduce the notion of \textit{split correctness}. Its purpose is to provide a formal guarantee that using an optimization strategy of data splitting, as described in beginning of this chapter, is valid. That is, both execution plans, the original plan and the optimized one, are equivalent in terms of their outputs. Formally, given a splitter \(L\) and two IE functions \(F, F'\), the problem of \textit{split correctness} addresses the question of whether for every string \(s\) applying \(F'\) to each substring of \(s\) produced by \(L\) is equivalent to applying \(F\) to \(s\). We say that \(F'\) \textit{correctly computes} \(F\) under \(L\), denoted by \(F \prec_L F'\), if for every string \(s \in \Sigma^*\), \(F(s)\) can be evaluated by applying \(F'\) to every substring of \(s\) extracted by \(L\), following by properly indenting the indices of the extracted spans. Formally,

\[
F(s) = \bigcup_{p \in L(s)} p \succ F'(s_p)
\]

We may simply write \(F \prec F'\) if \(L\) is clear from the context, or irrelevant. As an example, if \(F\) extracts person names and \(L\) is a sentence splitter, and if we have another IE function \(F'\) such that \(F \prec_L F'\), then the above definition states that applying \(F\) to every document would yield the same results as applying \(F'\) to every sentence of that document independently and taking the union of the results. In this case \(F'\) can be \(F\) itself. The question if there is any difference between executing \(F\) and \(F'\) under the split policy of \(L\) clearly depends on the definitions of \(F, F'\) and \(L\). Note the abuse of notation: formally \(L(s)\) consists of tuples and not spans, thus the expression \(p \in L(s)\) is malformed, and should have been written as \(p \in L(s)(x)\). However, since the schema of \(L\) has only one attribute, we omit the \((x)\) from the expression in order to simplify the notation.

It turns out that many standard NLP tools correctly compute themselves (i.e., \(F' = F\) in the above notation) under natural splitters like a sentence splitter that splits the text to sentences according to some Sentence Boundary Detection technique. Example 4.2.1 illustrates this point, and as well may strengthen the intuition behind the above definitions. It is not surprising to learn that NLP toolkits, such as the Stanford CoreNLP library, split the data into sentences as a pre-step for applying other NLP components. This is done in order to save memory, and more importantly, to shorten the running time since in many cases it is greatly influenced by the length of the document.
to be processed\textsuperscript{2}.

Example 4.2.1. Let us consider the task of Named Entity Recognition (NER). Recall from Chapter 2 that in NER we are interested in classifying named entities (i.e., real-world objects) mentioned in the text into predefined categories; typically these include persons, organizations and locations. Usually, the context required to determine the appropriate category for each named entity is the sentence the word appears in. Thus, applying a NER tool to an entire document should yield the same set of results as applying it to each sentence separately and collecting the results. This property can be formally expressed by \texttt{ner \cdot \text{sentence} \cdot ner}, where \texttt{ner} is an IE-function for some NER tool, and \texttt{sentence} is a splitter which delimits sentences in a given document.

The document shown in Figure 4.2(a) contains two sentences. We can apply a NER tool to the entire document and this should yield the relation shown in Figure 4.2(b). However, we can do the following instead. First apply a sentence splitter to the relation \texttt{Doc} which consequently yields the relation given in Figure 4.3(a). Then we apply a NER tool on each sentence separately, and subsequently we take the union of the results. Figures 4.3(b) and 4.3(c) show theses computations and their outcomes. The spans [1, 127] and [128, 196] delimit the two sentences appearing in the document. The two tables appearing on the right hand side of the shift operator in Figures 4.3(b) and 4.3(c) are the results of applying the NER tool on each sentence separately. By applying the shift operator on these tables, we get the spans that match the positions of the extracted entities in the original document. The union of the relations in Figures 4.3(b) and 4.3(c) results in the relation that appears in Figure 4.2(b).

Due to the fact that \texttt{ner \cdot \text{sentence} \cdot ner}, we know in advance that both alternatives are equivalent in terms of their outcomes, however, they may differ significantly performance-wise, as we shall see in Chapter 5 where we report on our test results based on running similar applications on our implementation.

The above naturally leads to the following decision problem:

<table>
<thead>
<tr>
<th>PROBLEM:</th>
<th>split correctness</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT:</td>
<td>splitter $\delta \in \text{SPLITTER}$ and two regex formulas $\omega, \omega' \in \text{REG}$</td>
</tr>
<tr>
<td>QUESTION:</td>
<td>Does $\omega'$ correctly compute $\omega$ under splitter $\delta$?</td>
</tr>
</tbody>
</table>

Although a more general problem could have been considered, the analysis that follows discusses the special case where the splitter and IE functions are chosen from the above specific classes.

The following lemma\textsuperscript{3} characterizes data splitting by means of regex formulas (extended with algebraic operations). Thus, instead of discussing data splitting in terms

\textsuperscript{2}Read more here for the case of CoreNLP: https://stanfordnlp.github.io/CoreNLP/memory-time.html#where-does-all-the-time-go

\textsuperscript{3}This lemma has been established in a collaboration with Johannes Doleschal, Wim Martens and Frank Neven.
Co-founded by Jan Koum and Brian Acton, WhatsApp is an instant messaging subscription service that initially launched in 2009. In February 2014, Facebook announced it would be acquiring WhatsApp.

(a) A document with named entities

<table>
<thead>
<tr>
<th>location</th>
<th>category</th>
</tr>
</thead>
<tbody>
<tr>
<td>[15, 23]</td>
<td>Person</td>
</tr>
<tr>
<td>[28, 39]</td>
<td>Person</td>
</tr>
<tr>
<td>[41, 49]</td>
<td>Organization</td>
</tr>
<tr>
<td>[122, 126]</td>
<td>Date</td>
</tr>
<tr>
<td>[131, 144]</td>
<td>Date</td>
</tr>
<tr>
<td>[146, 154]</td>
<td>Organization</td>
</tr>
<tr>
<td>[187, 195]</td>
<td>Organization</td>
</tr>
</tbody>
</table>

(b) Extracted named entities

Figure 4.2: Extracting named entities from a document

of functions, we can refer to them by using the syntax and semantics of regex formulas that we introduced in Section 3.2.

**Lemma 4.2.2.** Let \( \omega \) be a regex formula in \( \text{REG} \) over the schema \( \vec{\gamma} = \text{sch}(\omega) \), and \( \delta \) be a splitter in \( \text{SPLITTER} \) over the schema \( x = \text{sch}(\delta) \) such that \( x \notin \vec{\gamma} \). For every \( \vec{s} \in \Sigma^* \) the following holds:

\[
\bigcup_{p \in \delta[[\vec{s}]]} p \gg [[\omega]](\vec{s}_p) = [[\pi_{\vec{\gamma}}((\Sigma^* \cdot x\{\omega\} \cdot \Sigma^*) \preceq \delta)]](\vec{s})
\]

Intuitively, the lemma asserts that taking the union of all the span relations produced by applying \( \omega \) to each substring of \( \vec{s} \) under the split policy of \( \delta \) would yield the same results as applying the regex formula \( \pi_{\vec{\gamma}}((\Sigma^* \cdot x\{\omega\} \cdot \Sigma^*) \preceq \delta) \) to the entire string \( \vec{s} \). Recall from Section 3.2 that for some regex formula \( \omega \) we denote by \( [[\omega]] \) the spanner \( \omega \) represents, which is by itself a function that maps a string \( \vec{s} \) to a span relation. The expression on the left hand side of the equation processes the input string \( \vec{s} \) by (1) applying the spanner \( [[\omega]] \) to a substring of \( \vec{s} \), that is \( \vec{s}_p \), and then (2) taking union of the results. On the right hand side of the equation we define a new spanner consisting of the variables of \( \omega \) and \( \delta \). Informally, for each substring delimited by \( \delta \) we look for matches according to \( \omega \). Since we are only interested in the variables of \( \omega \), then we use
Therefore it must hold that we have that

Let \( \hat{t} \) such that \( \hat{t} \in \Sigma^* \cdot \omega \cdot \Sigma^* \). Recall that \( \hat{t} \) is a tuple over \( \hat{t} \). Let us extend \( \hat{t} \) to \( \hat{t} \) such that \( \hat{t} \) is a tuple over \( \hat{t} \) and \( \hat{t} = \hat{t} \ || \ p \), where \( || \) denotes concatenation of values. Notice that here we assume that the value of \( \hat{t} \) corresponds to variable \( x \) is the last one in \( \hat{t} \). It is the case that \( \hat{t} \in \Sigma^* \cdot \omega \cdot \Sigma^* \). Together with \( p \in [\delta](s) \), we have that \( \hat{t} \in \pi_{\hat{t}}((\Sigma^* \cdot \omega) \cdot \Sigma^*) \equiv \delta \). Therefore there is a span \( \hat{t} \in \pi_{\hat{t}}((\Sigma^* \cdot \omega) \cdot \Sigma^*) \equiv \delta \).

We proceed by showing that

As before, let \( s \) be a string, and let \( \hat{t} \) be a tuple in \( \bigcup_{p \in [\delta](s)} p \gg [\omega](s_p) \). Therefore there is a span \( p \in [\delta](s) \) such that \( \hat{t} \in \pi_{\hat{t}}((\Sigma^* \cdot \omega) \cdot \Sigma^*) \equiv \delta \). Then, there is a span \( p \in [\delta](s) \) such that \( \hat{t} \in \pi_{\hat{t}}((\Sigma^* \cdot \omega) \cdot \Sigma^*) \equiv \delta \). Therefore it must hold that \( \hat{t} \in p \gg [\omega](s_p) \). It follows that \( \hat{t} \in \bigcup_{p \in [\delta](s)} p \gg [\omega](s_p) \).
From Lemma 4.2.2 we obtain the following Theorem.

**Theorem 4.1.** Let $\omega$ and $\omega'$ be two regex formulas in REG over the same schema $\mathcal{Y}$. That is, $\mathcal{Y} = \text{sch}(\omega) = \text{sch}(\omega')$. Additionally, let $\delta$ be a splitter in SPLITTER over the schema $x = \text{sch}(\delta)$ such that $x \notin \mathcal{Y}$. It holds that $\omega \preceq_{\delta} \omega'$ if and only if

$$\llbracket \omega \rrbracket = \llbracket \pi_{\mathcal{Y}}\left((\Sigma^* \cdot x\{\omega'\} \cdot \Sigma^*) \bowtie \delta\right)\rrbracket$$

**Proof of Theorem 4.1.** Assume $\omega \preceq_{\delta} \omega'$. By definition, we have that for every string $s \in \Sigma^*$ it holds that $\llbracket \omega\rrbracket(s) = \bigcup_{p \in \mathcal{L}(s)} \mathcal{P} \gg \llbracket \omega'\rrbracket(s)$. By Lemma 4.2.2, we know that $\bigcup_{p \in \mathcal{L}(\bar{s})} \mathcal{P} \gg \llbracket \omega'\rrbracket(\bar{s}) = \llbracket \pi_{\mathcal{Y}}\left((\Sigma^* \cdot x\{\omega'\} \cdot \Sigma^*) \bowtie \delta\right)\rrbracket(\bar{s})$. Therefore $\llbracket \omega\rrbracket = \llbracket \pi_{\mathcal{Y}}\left((\Sigma^* \cdot x\{\omega'\} \cdot \Sigma^*) \bowtie \delta\right)\rrbracket$.

Now assume $\llbracket \omega\rrbracket = \llbracket \pi_{\mathcal{Y}}\left((\Sigma^* \cdot x\{\omega'\} \cdot \Sigma^*) \bowtie \delta\right)\rrbracket$. By Lemma 4.2.2, we know that for every string $\bar{s} \in \Sigma^*$ it holds that $\bigcup_{p \in \mathcal{L}(\bar{s})} \mathcal{P} \gg \llbracket \omega'\rrbracket(\bar{s}) = \llbracket \pi_{\mathcal{Y}}\left((\Sigma^* \cdot x\{\omega'\} \cdot \Sigma^*) \bowtie \delta\right)\rrbracket(\bar{s})$. Therefore $\llbracket \omega\rrbracket(\bar{s}) = \bigcup_{p \in \mathcal{L}(\bar{s})} \mathcal{P} \gg \llbracket \omega'\rrbracket(\bar{s})$. \qed

It was not initially clear that the problem of split correctness is decidable. The following corollary tells us that it is so, and it is even solvable in polynomial space.

**Corollary 4.2.** The problem of split correctness is in PSPACE.

**Proof of Corollary 4.2.** A variable set automaton, or vset-automaton for short, is a special type of automaton that is used to represent spanners [FKRV15]. Moreover, in the work by Fagin et al. it has been shown that any regular spanner can be represented by a vset-automaton. In a recent work by Maturana et al. [MRV17] it has been shown that containment of vset-automata is in PSPACE-complete. Therefore, and due to Theorem 4.1, we can apply the strategy of testing standard automata equivalence for the case of split correctness: Let $\omega, \omega'$ be two regex formulas in REG, and $\delta$ be a splitter in SPLITTER. Denote by $\omega''$ the regex formula $\pi_{\mathcal{Y}}\left((\Sigma^* \cdot x\{\omega'\} \cdot \Sigma^*) \bowtie \delta\right)$. Let $A$ be the vset-automaton for $(\omega \setminus \omega'') \cup (\omega'' \setminus \omega)$, where \ is the difference operator [FKRV15]. Return True if $A$ is empty, and False otherwise. \qed
Chapter 5

Implementation and Experiments

In this chapter we present a full implementation that follows our definitions from Chapter 3. We give an outline of the implementation, and delve into a few interesting aspects. We then turn our discussion to concurrent execution on top of Apache Spark, a cluster-computing framework. Specifically, through several case studies, we investigate the impact of the data splitting strategy for performance optimization as it was presented in Chapter 4.

5.1 Implementation atop PostgreSQL

5.1.1 Implementation Strategy

Being an extension to Datalog, Spannerlog as a query language requires standard relational algebra operation such as join and projection. In order to avoid reinventing the wheel, in our implementation of Spannerlog we used PostgreSQL\(^1\), an open source Relational Database Management System (RDBMS), as an execution engine. This enables us to use PostgreSQL’s built-in capabilities like query processing as well as more advanced features such as User-Defined Functions (UDFs). These are functions written in Python by the user that can be plugged into SQL queries. In our implementation, IE functions, such as regex formulas and standard NLP tools, are represented as UDFs.\(^2\)

In addition to PostgreSQL, our implementation also relies on Standard’s Deep-Dive\(^3\) [Zha15], a data management system for automatic Knowledge-Base Construction (KBC). DeepDive applications are written in a designated programming language called DDlog that is used to employ statistical learning and inference. DDlog also facilitates the integration of external data sources of various formats like CSV. It also offers an easy way of incorporating into its workflow UDFs written in Python or Java. Roughly

---

\(^1\)See link: https://www.postgresql.org

\(^2\)We have made the source code available through publicly open GitHub repositories:

- https://github.com/TechnionTDK/spannerlog
- https://github.com/TechnionTDK/spannerlog-shell

\(^3\)See link: http://deepdive.stanford.edu
speaking, DDlog programs consist of (1) normal Datalog-like rules, and (2) inference rules. Under the hood, DeepDive translates rules of the first category to SQL queries, and of the second category to factor graphs, a type of probabilistic graphical model, used for running probabilistic inference algorithms [KF09]. Inference rules carry an important role in developing IE applications: When dealing with noisy and uncertain data, past experience has shown that incorporating statistical inference capabilities can go a long way in achieving high-quality results while reducing the need for developers to exactly specify rules for each possible case. DeepDive and other systems like ProbLog [DRKT07] use declarative statements to allow users to easily employ this capability in their development process.

We translate the Spannerlog queries to SQL ourselves, since we found that DDlog materializes its UDFs. This means that intermediate results are stored in normal PostgreSQL tables. In the case where the intermediate results are considerably large, this behavior can have a dire impact on running time, as indeed we have witnessed during the development process. The reason for this is due to blowup in the database size, as well as to the insertion of additional join operations involving large, non-indexed tables. These in turn increase the disk I/O load as well. We use DeepDive for integrating CSV files to the RDBMS, which is done quite easily using this system.

5.1.2 Dataflow

In this section we explain the dataflow of our implementation of Spannerlog. Figure 5.1 visualizes the main components of the system and the way they interact with one another. A square with a folded corner represents an input file of the format specified by the extension of the filename. A square without a folded corner represents a script written in the programming language specified at the bottom. A script receives one or more input files and outputs a text file which in turn serves as another input file for another script. After installing Spannerlog and all of its components, the user can compile Spannerlog programs by running the command `spl compile` from the command line. Before doing so, the user must verify that the directory from which the command is executed contains the following files:

- **EDB/** - a folder consists of CSV files, each represents a table and stores records of data.
- **IEF/** - an optional folder consists of python files, each defines a user-defined IE functions. We note that the implementation automatically includes built-in IE functions such as Named Entity Recognition (NER) tool, and sentence splitter implemented on top of Stanford’s CoreNLP library [MSB+14].
- **app.spl** - a file consists of Spannerlog rules.
- **db.url** - specifies via a URL the RDBMS server location, and the database name that will be used for the program.
The compilation process of a Spannerlog program begins with an automatic extraction of the EDB and IEF schemas. The script `extract-edb-schema.py` peeks at the first record row of each CSV file and according to each value deduces the schema of the table it represents. For example, if the first row consists of the three values `train`, `4` and `1.36`, then the script determines that the corresponding types are string, integer and float. Note that formally our framework considers all of them as type string, however for real-world scenarios it is useful to consider other types as well. These extracted schemas are represented as a JSON object stored in file `edb.schema.json`. The script `extract-ief-schema.py` does the same for the files in the IEF folder. In this case, the script does not deduce the IE schema from the code since it is explicitly written in a designated format. Its output file is `ief.schema.json`.

**Example 5.1.1.** Let us describe an application that will be used as a running example for this section, and will appear again in the following section as well. Consider the task of extracting mentions of p53 kinases from scientific literature. Various multicellular organisms, including humans, possess a particular protein that is called p53. This protein carries an important role in maintaining a normally functioning body as it functions as a tumor suppressor. A kinase is a special type of protein that interacts with other proteins by means of phosphorylation, a biochemical process that involves the addition of phosphate molecule to an organic compound. The p53 kinases are kinases that phosphorylate the p53 protein, a process that affects the activation and deactivation of p53. A malfunctioning p53 kinase may be a cause of cancer, and research on cancer prevention has continuously held a special interest in P53 kinases. This application is inspired by a rather recent work that explored this idea more comprehensively, and reported on positive results [SWB+14, NWB+15]. For illustration, let us examine a concrete example. In Wu et al. [WLP+14] the following sentence appears in the abstract:

> We report here that TAF1 phosphorylates p53 at Thr55, leading to dissociation of [...] 

The application given in lines (5.1) to (5.5) extracts the fact that kinase TAF1 is a p53 kinase.

\[
Q(kid, pid) \leftarrow \quad (5.1)
\]

\[
Abstracts(_-, pid, abstract, _, _), \quad (5.2)
\]

\[
\langle abstract \rangle [\sum^* \cdot x \{\sum^* \cdot \text{phosphorylates p53} \cdot \sum^* \} ], \quad (5.3)
\]

\[
\text{lowercase}(\text{abstract}_x)(\text{alias}), \quad (5.4)
\]

\[
\text{Aliases}(kid, alias). \quad (5.5)
\]

Here, `Abstracts` is an EDB relation whose attribute names are kinase_id, pubmed_id, abstract, link, and publication date. Recall that variables denoted with overlines are of type string. The underscore is used here to indicate that the corresponding attributes
are irrelevant in this particular rule. The IE function lowercase returns the input string it receives with all of its characters converted to their lowercase form. Aliases is another EDB relation whose schema is kinase_id and alias. It stores the common aliases each kinase has.

Given the suitable data files, the script extract-ief-schema.py generates for the above program the file edb.schema.json that appears here:

```json
{
    "aliases": {
        "kid": "int",
        "alias": "text"
    },
    "abstracts": {
        "kid": "int",
        "pubmed_id": "int",
        "abstract": "text",
        "link": "text",
        "publication_date": "text"
    }
}
```

The main component of the system is the compiler. The program, written in Java, takes as input the above two schema files and the file app.spl that consists the rules of the program. The compiler defines a new grammar that corresponds to the definitions of the query language presented in Chapter 3. It does so by using the Antlr [Par13] tool, an open-source parser-generator software. Additionally, the compiler defines a second grammar for a rather simple but comprehensive standard of regular expressions. Each regex formula in a Spannerlog rule is translated into a PostgreSQL function written in the Python language. These will later be plugged into the RDBMS.

In the compilation process, a regex formula is translated to a Python’s regular expression using the re library. Each span variable appearing in a regex formula is replaced by a named group carrying the name of the variable. Spans are handled in a rather simple way where each span is represented by a pair of integers. That is, an attribute of type span is mapped to two attributes of type integer. The program compiles each Spannerlog rule into an appropriate SQL rule. The process involves deducing the IDB schema by chasing the query variables using the well known DFS algorithm. In case of conflict between attribute types, or detecting that it is impossible to deduce the type from the program, the compilation process ends with a corresponding error message.

4See link: https://docs.python.org/3/library/re.html
Example 5.1.2. The rule appearing in the program of Example 5.1.1 is translated into the following SQL query:

```
SELECT
    R2.lower AS column1,
    R0.pubmed_id AS column2
FROM
    abstracts R0,
    rgx1(R0.abstract) R1
lowercase(substr(R0.abstract, R1.x_start, (R1.x_end - R1.x_start))) R2,
    aliases R3
WHERE
    R3.alias = R2.lower
```

Finally, the DFS algorithm is being run again, but this time on the relation names, in order to construct an execution plan. In our running example, there is only one rule, therefore the execution plan is straightforward. All the outputs of the above scripts are being collected into a JSON file called compiled.spl.json.

The following three scripts take this JSON file as input:

- `compile-to-ddlog.py` - creates a file written in DDlog to be run by DeepDive. This DDlog program in charge of loading the data stored in the CSV files found in the EDB folder to the RDBMS.

- `compile-rgx-to-sql.py` - creates a PostgreSQL script that acts as a wrapper for each compiled regular expression. When running the script, a PostgreSQL UDF is created for each regular expression that can later be invoked from inside SQL queries.

- `compile-execution-plan.py` - According to the execution plan created by the Java Program, this script creates another script, written in the Bash language, that will be executed upon invoking the command `spl run` from the command line, after a successful compilation process.

Example 5.1.3. In our running example, the following code is being generated by the script `compile-rgx-to-sql.py`. The syntax follows the PostgreSQL specifications for defining UDFs written in Python.

```
CREATE FUNCTION rgx1(s text)
    RETURNS TABLE (x_start int, x_end int)
AS $$
import re
pattern = re.compile(r".*(?P<x>[A-Za-z0-9]+)\sphosphorylates[^.]*p53.*")
$$
```
match = pattern.match(s)
if match:
    yield [
        match.start('x') + 1,
        match.end('x') + 1,
    ]

$\text{LANGUAGE plpythonu;}$

5.1.3 Web Interface

Lastly, in order to facilitate in the experimentation with our implementation by other users, and for the ease of presentation, we created a simple web interface that users can write Spannerlog applications with. Figure 5.2 shows a screenshot of an application that extract complaints on costumer service from a review database. The application will be further explained in the next section.

5.2 Parallel Execution atop Spark

In this section, we describe our efforts to put into practice the optimization technique of data splitting that we presented in Section 2.4. We present three case studies on which
we examine how such an approach affects the running times.

In addition to the implementation we previously described, we also explored the option of an implementation on top of Apache Spark on several use cases in order to examine the performance in a distributed setting. Apache Spark is an open-source cluster-computing framework. Originally developed at the University of California, Berkeley’s AMPLab [ZCF+10], the Spark codebase was later donated to the Apache Software Foundation, which has maintained it since, and currently it is one the most active project in Apache. Spark provides an interface for programming entire clusters with implicit data parallelism and fault tolerance. For applications that require cluster-computing capabilities, Spark would be a natural choice, as it has been in our case here.

Our Spark implantation uses the Spark SQL module [AXL+15] that integrates relational processing with Spark’s functional programming API. Importantly, using this module allows to further optimize the compiled program due to the Catalyst Optimizer which leverages the schema of the relations involved, as well as advanced features of the Scala programming language (that Spark is written in). In addition, the Spark SQL module enables us to keep the SQL syntax we used in our PostgreSQL implementation.

We note that due to time constraints, we only use Spark as a test case. Therefore, instead of providing a full implementation as we did for our PostgreSQL implementation where a Spannerlog program is automatically compiled, here the translation to Spark code was done by hand. This, however, does not come into conflict with the purpose of this section, since here we are mainly interested in examining the impact of applying the data splitting optimization as described in Chapter 4, and compare it to our main implementation on top of PostgreSQL on a few test cases.

5.2.1 Case Studies

We now describe a few case studies we examined in order to evaluate the effect of data splitting as an optimization technique. After we describe the applications, we discuss the results. Note that the applications described below can be easily improved by, for instance, adding more rules, or tweaking the current ones. However, the simple versions of them given here are sufficient for the purposes of this discussion.

A. Extracting mentions of p53 kinases from scientific papers. The first application we consider is the one we already described in Example 5.1.1.

B. Extracting business transactions from financial news articles. In this application we are given a dataset of financial news articles from Reuters, the international news agency, and wish to extract transactions between two organizations and the money involved in that transaction. For example, consider the following excerpt from some news article:

---

5See link: http://spark.apache.org
6See link: https://projects.apache.org/statistics.html
German builder Hochtief has sold its airports division to Canada’s Public Sector Pension Investment Board (PSP Investments) for 1.1 billion euros ($1.4 billion), seeking to cut debt and invest in its infrastructure business.

Our extractor should recognize that a transaction of worth 1.1 billion euros (or 1.4 billion American dollars) was made between Hochtief and PSP Investments. For this purpose we have written the program given in lines (5.6) to (5.11).

\[
Q(id, \text{text}_x, \text{text}_y, \text{text}_z, \text{date}) \leftarrow \\
\text{Articles}(\text{id, date, text}), \\
\text{ner(\text{text})(x, ORGANIZATION)}, \\
\text{ner(\text{text})(y, ORGANIZATION)}, \\
\text{ner(\text{text})(z, MONEY)}, \\
x < y.
\]

The relation Articles consists of 5 attributes of which we consider only 3: the article id (\text{id}), its publication date (\text{date}), and its text body (\text{text}). We then use the IE function \text{ner}, which we have encountered before, to extract the appropriate entities from the text. The last condition was not formally defined in Chapter 3, however this is a natural and convenient extension to the syntax where we allow to write conditions inside of a rule involving the rules’ variables. Here, the purpose of the expression \(x < y\) is to avoid duplications in the result set. Its semantics is the following: let \(x = [i_1, j_2]\) and \(y = [i_1, j_2]\) be two spans. We say that \(y\) is greater than \(x\), denoted by \(x < y\), if \(i_1 < i_2\).

C. Extracting bad reviews on costumer service from a food review dataset.

In this application we are given a review database on food products, and we wish to extract all of the negative reviews that address the costumer service. Here is an example of such a review:

\[\ldots\] I will never order anything from this company again and do not recommend it to anyone based on this horrible customer service and lack of responsibility and ownership.

To accomplish this, we wrote the following Spannerlog program:
The schema of the the relation \textit{Reviews} consists of 10 attributes, but we only take use of three of them. These are (1) \texttt{rid}, the review id; (2) \texttt{score}, the score of a scale from 1 to 5 the user gave for the review, where 1 represents the lowest rank, and 5 represents the highest; and (3) \texttt{text}, the review text. The rule appearing in lines (5.12) to (5.15) creates the intermediate relation \textit{R} that consists only reviews who were ranked with a low score (1 or 2), and include the word service in them. Lines (5.16) to (5.21) form the second rule of the program that creates the relation \textit{Q} which consists of the exact sentences that include the word service in them, and express a negative sentiment. Note that in this example, the schema of the IE function sentiment is different than the we saw in Example 3.4.1. Here, in addition to the label of the sentiment (e.g., ‘VERY POSITIVE’ or ‘NEUTRAL’), there is a second attribute that assigns a value to each sentiment. In this case, the values range from 0 to 4, where 0 is assigned to the most negative sentiment, and 4 to the most positive one.

Results

The three applications listed above were executed in both sequential as well as in parallel manner. We executed the above programs in three settings: (1) our implementation on top of PostgreSQL as we earlier described, using a 16 GB memory and two cores machine; (2) Spark with a pseudo-cluster, consisting only of one machine that has 8 GB memory and two cores; and (3) Spark with a cluster consisting of 5 (physically separate) machines, each has 8 GB memory and two cores. Every execution was run in two manners w.r.t. the input data: the data was either (1) “unsplit”, that is, taken in its original form; or (2) split, where the data had been split into pieces, which were then stored in the database. In all cases the data splitting was carried out by the CoreNLP’s
<table>
<thead>
<tr>
<th>Dataset (by application)</th>
<th>Size [MB]</th>
<th>No. of Documents</th>
<th>Matches Found</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>13.4</td>
<td>10,000</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>27.9</td>
<td>9,109</td>
<td>16038</td>
</tr>
<tr>
<td>C</td>
<td>286</td>
<td>568,454</td>
<td>4306</td>
</tr>
</tbody>
</table>

Table 5.1: Datasets descriptions

sentence splitting annotator [MSB⁺14]. When measuring running time for a Spannerlog program with split data, we assumed the splitting process has already been preformed. We justify this by the ubiquity of text extractors whose scope of consideration does not go further than the current sentence, and therefore it would make sense to make an available copy of the database with split data in order to boost performance.

We used several publicly available datasets for our experiments. The scientific papers used for application A published on the MEDLINE database that stores references and abstracts on life sciences and biomedical topics. This dataset size is 13.4 MB. For application B we used a Financial News Dataset of size 27.9 MB from Reuters. Lastly, for application C we used the The Amazon Fine Food Reviews dataset of size 286 MB that was originally published on SNAP, and is also available on Kaggle. Table 5.1 summarizes some details on the datasets. Note that the values shown under the column Matches Found were slightly varied in each execution setting.

Tables 5.2(a), 5.2(b) and 5.2(c) summarize the running times of the different execution specified above. In each table, a row represents a run using a different execution engine, as specified by the first column. The second and thirds columns specify the running times for the two settings we conducted our experiments. In the first setting we run the program on the input documents with no preprocessing involved. In the second setting we split the documents beforehand using a sentence splitter, then run the program on the sentences independently, and finally we take the union of the results. As justified earlier, the running time shown in the third column does not take into account the preprocessing time, that is the time needed for splitting the data.

By examining the results, we can note several considerable speedups, particularly in the cluster setting that consists of several nodes. To the best of our understanding, this improvement can be explained by the fact that splitting provides Spark with parallelizable tasks that are smaller in cost and larger in number; hence, we provide Spark with considerably more (smartly exploited) control over scheduling and resource allocation. We also witness a big difference between application A and B that roughly process the same size of the data. We explain this by the lack of use of NLP processing

---

7See link: https://www.ncbi.nlm.nih.gov/pubmed
8See link: https://github.com/philipperemy/financial-news-dataset
9See link: http://snap.stanford.edu/data/web-FineFoods.html
10See link: https://www.kaggle.com/snap/amazon-fine-food-reviews
<table>
<thead>
<tr>
<th>Engine</th>
<th>Running Time No-Split [sec]</th>
<th>Running Time Split [sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td>1.780006475</td>
<td>2.600266532</td>
</tr>
<tr>
<td>Spark (1 node)</td>
<td>8.189759615</td>
<td>9.247583935</td>
</tr>
<tr>
<td>Spark (5 nodes)</td>
<td>31.02037283</td>
<td>31.62642127</td>
</tr>
</tbody>
</table>

(a) Running times for Application A.

<table>
<thead>
<tr>
<th>Engine</th>
<th>Running Time No-Split [sec]</th>
<th>Running Time Split [sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td>4754.19537</td>
<td>4562.736498</td>
</tr>
<tr>
<td>Spark (1 node)</td>
<td>21859.31981</td>
<td>10867.87844</td>
</tr>
<tr>
<td>Spark (5 nodes)</td>
<td>4909.07533</td>
<td>2468.670568</td>
</tr>
</tbody>
</table>

(b) Running times for Application B.

<table>
<thead>
<tr>
<th>Engine</th>
<th>Running Time No-Split [sec]</th>
<th>Running Time Split [sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td>873.2588557</td>
<td>183.847376474</td>
</tr>
<tr>
<td>Spark (1 node)</td>
<td>8182.339951</td>
<td>1360.685119</td>
</tr>
<tr>
<td>Spark (5 nodes)</td>
<td>1495.74022</td>
<td>359.8905612</td>
</tr>
</tbody>
</table>

(c) Running times for Application C.

Table 5.2: Running times.

tools such as named-entity recognition and sentiment analysis. In applications A and B we observe that even on a single machine, splitting the data resulted in improved running times. Not in every case the data splitting was proven to be beneficial, however in none of the case it worsens the results.
Chapter 6

Conclusions

In this thesis we presented our endeavor of taking databases to the next step by incorporating into them capabilities of text-analytics. We addressed this challenge by extending the widespread relational model with the spanner theory that forms the mathematical foundation for IBM SystemT. We generalized the notion of spanners in order to be able to address a more comprehensive family of text-processing functions. By incorporating the spanner object into the relational database model, we were able to enrich its capabilities while taking benefit of our familiarity with a standard and widespread environment. Additionally, the combination of spanner theory together with the basic theory of relational databases formed a new perspective between the fields of databases, logic, automata, formal languages and complexity. Here we reported on initial results, and these, together with those of Peterfreund et al. [PCFK17], suggest interesting directions in this line of research.

In addition, using the formalism we introduced, we were able to define and analyze a technique for query optimization by means of data splitting while providing a mathematical guarantees of its correctness. Even though we focused here on spanners, we can also extend the scope of the problem of split correctness from spanners to rules, as well as to programs. More precisely, in the case we considered we asked whether the evaluation of a spanner produces the same results as the evaluation of another spanner under a given split policy. In the more general case, we can ask the same question for whole rules and programs, rather than just spanners. Detecting even fragments of a program that can be parallelized may have a significant impact on its running time.

Following the definition and analysis, we proceeded to presenting the implementation that take the role of a proof-of-concept by showing the feasibility of our framework and of the data-splitting optimization technique on top of standard systems such as PostgreSQL and Spark. We provided several applications to illustrate the ease of use, and to show the impact of this technique. It is our conclusion that in common scenarios that involve massive textual data, this methodology may significantly improve performance as it deals with the costly information extractors, which often considered as the main bottleneck in IE applications. The results suggest it is beneficial to perform
data splitting in the preprocessing phase, even though this raises the requirement of duplicating the data in order to store the splits. The first reason for this is that it is often preferable to pay with increased disk space in order to gain better running times. Secondly, storing the split may help avoiding recomputing them for other IE programs that use the same data. This is because the splits can be reused by multiple programs as it is likely that these programs will require standard fragmentations of the text such as splitting by sentences, by paragraphs and by k-grams.

Future research may continue in several directions. In cluster-computing frameworks, we may want to have more control over the distribution of the data to the worker nodes. In our experiments, we only split the data, but leave it to Spark to handle the data distribution over to the nodes in hope it will succeed in obtaining improved running times. Using the data splitting, and through the notion of split correctness, we can fine-tune data distribution according to predicted computational costs of parallel tasks based on the complexity of the involved IE functions.

Another direction to explore is the incorporation of uncertainty to the Spannerlog framework. This is an important feature when dealing with noisy data such as text documents. Frameworks like ProbLog [DRKT07] and Markov Logic Networks [RD06] have incorporated the element of uncertainty to traditional Datalog and first-order logic rules, and showed the advantages of doing so. Whether these frameworks can be extended to Spannerlog remains an open question.
Bibliography


[BGK*05] Razvan Bunescu, Ruifang Ge, Rohit J Kate, Edward M Marcotte, Raymond J Mooney, Arun K Ramani, and Yuk Wah Wong. Compar-


[CLR13] Laura Chiticariu, Yunyao Li, and Frederick R Reiss. Rule-based information extraction is dead! long live rule-based information extraction systems! In *EMNLP*, number October, pages 827–832, 2013.


ביותר ברוכים חלף מודיע יבישו מוסים. טכניקות דיווח לייעול בטור של שאלת נושא ויאפשרות אחר מושפעת עבור

t狀況 estándard de acuerdo con el caso de la tecnología. En este

מים נמצאים ראשית "כותרת פיצול" המאשרת בינו של התוכן בכינו מובילים והונים מעמשות על פיצול

נתונים טקסטואליים למסר של טקסטים קצרים וירר על סמך מודע פיצול חלבון של הנסיגה. משה

ה bulundו של התוכן המLineNumber של מעת פיצול הטקסט או מפקת את הrobote או התוכן השחרר בדרך התוכן

שה настоящее והיה גם לשה настоящее המקורית, אלא הפיצול.

פיתוח תאוריות והגדרתן הפורמליות אפשר מחקר משמשות ומעמיקיםدع, פעוט הוריאלול של המ시장

róbוט, והם ש_ALLOW להוראות בערב כ Jazeera שולמה הרביעי או את פיתוח עד יד פיניק, אלטרנטיבי

הספרטיסים ארז פיתוחה על ידי פיניק וארטימ, גודטסאט או התוכן המשופת. באנון האפקט, אנות

ש分かるיה של ספרטיסים היה על השפעה רבד עגור מתערכות ליגוי ומקMounted טקסטואלי.
The system streamlines opportunities to enhance meaning. For example, a system built to reduce the use of natural language processing algorithms because they are clearer in terms of processing time. A program that has been designed can achieve that goal if it is processed on a layer that precedes the natural language processing layer.

The main goal of the thesis here is to propose a method to strengthen its shortcomings as described. Another approach is to focus on the theoretical model of the system and extend it for textual information. In this thesis, a formal, new, efficient system for managing textual information is presented called Spenser, which is based on its theoretical structure.

Spenser systems are data processing systems that work on text. In a more formal sense, a Spenser map a given string to a relation of mixed types (spans) — that is, a relation that is composed of segments of types, only in the case of mixed types. This can be represented by a table with extra columns: a column of mixed type takes its values from the words in the text, and an additional column of textual type indicates the segmentation of the text. In Spenser, Spenser systems are called functions. The meaning, or semantics, is provided by the extracted information. The meaning is then defined by a function that maps the extracted information to another set of information.

The contribution here is a system that extracts mixed data-spans from the text and maps to the text's features and meta-data, such as identifying phrases of time, and functions that copy elements from the first set to the second set. These functions are called functions of data processing. The meaning is then defined by a function that maps the extracted information to another set of information.

As mentioned above, Spenser systems are data processing systems that work on text. In a more formal sense, a Spenser map a given string to a relation of mixed types (spans) — that is, a relation that is composed of segments of types, only in the case of mixed types. This can be represented by a table with extra columns: a column of mixed type takes its values from the words in the text, and an additional column of textual type indicates the segmentation of the text. In Spenser, Spenser systems are called functions. The meaning, or semantics, is provided by the extracted information. The meaning is then defined by a function that maps the extracted information to another set of information.
תקציר

בעידן הביג-דאטס, הרשתותاجتماعיות, הערכותאודות בטיחות (אינטראקטיביות) יוצרים닫ות ב📸 לשידוריםigitais והם מונים בחלקה של מידע ובדבריות, במילוי שאלות ותח困難ים נפרדות במודדים, המ hút המים ורבות של פיתוח ותחום. לדוגמה, גניזת תכניות ותעשיות של אינטגרציה עם ביליבות, פיתוחים לשירותי פיקוח ואיתומי מוסר, אדיבות למשתמשים, חתימות שונות, אמצעים, הם תקנות וטworthy במערכי ודיעה, הבכורה בManifest, ובו שילובים חכמים בقوي עץ הזמנים, Năm תקנות וסידור הדורשים במערכי ודיעה, הבכורה בManifest, ובו שילובים חכמים בقوي עץ הזמנים, Năm תקנות וסידור הדורשים במערכי ודיעה, הבכורה בManifest, ובו שילובים חכמים בقوي עץ הזמנים, Năm תקנות וסידור הדורשים במערכי ודיעה, הבכורה בManifest, ובו שילובים חכמים בقوي עץ הזמנים, Năm תקנות וסידור הדורשים במערכי ודיעה, הבכורה בManifest, ובו שילובים חכמים בقوي עץ הזמנים, Năm תקנות וסידור הדורשים במערכי ודיעה, הבכורה בManifest, ובו שילובים חכמים בقوي עץ הזמנים, Năm תקנות וסידור הדורשים במערכי ודיעה, הבכורה בManifest, ובו שילובים חכמים בقوي עץ הזמנים, Năm תקנות וסידור הדורשים במערכי ודיעה, הבכורה בManifest, ובו שילובים חכמים בقوي עץ הזמנים, Năm תקנות וסידור הדורשים במערכי ודיעה, הבכורה בManifest, ובו שילובים חכמים בقوي עץ הזמנים, Năm תקנות וסידור הדורשים במערכי ודיעה, הבכורה בManifest, ובו שילובים חכמים בقوي עץ הזמנים, Năm תקנות וסידור הדורשים במערכי ודיעה, הבכורה בManifest, ובו שילובים חכמים בقوي עץ הזמנים, Năm תקנות וסידור הדורשים במערכי ודיעה, הבכורה בManifest, ובו שילובים חכמים בقوي עץ הזמנים, Năm תקנות וסידור הדורשים במערכי ודיעה, הבכורה בManifest, ובו שילובים חכמים בقوي עץ הזמנים, Năm תקנות וסידור הדורשים במערכי ודיעה, הבכורה בManifest, ובו שילובים חכמים בقوي עץ הזמנים, Năm תקנות וסידור הדורשים במערכי ודיעה, הבכורה בManifest, ובו שילובים חכמים בقوي עץ הזמנים, Năm תקנות וסידור הדורשים במערכי ודיעה, הבכורה בManifest, ובו שילובים חכמים בقوي עץ הזמנים, Năm תקנות וסידור הדורשים במערכי ודיעה, הבכורה בManifest, ובו שילובים חכמים בقوي עץ הזמנים, Năm תקנות וסידור הדורשים במערכי ודיעה, הבכורה בManifest, ובו שילובים חכמים בقوي עץ הזמנים, Năm תקנות וסידור הדורשים במערכי ודיעה, הבכורה בManifest, ובו שילובים חכמים בقوي עץ הזמנים, Năm תקנות וסידור הדורשים במערכי ודיעה, הבכורה בManifest, ובו שילובים חכמים בقوي עץ הזמנים, Năm תקנות וסידור הדורשים במערכי ודיעה, הבכורה בManifest, ובו שילובים חכמים בقوي עץ הזמנים, Năm תקנות וסידור הדורשים במערכי ודיעה, הבכורה בManifest, ובו שילובים חכמים בقوي עץ הזמנים, Năm תקנות וסידור הדורשים במערכי ודיעה, הבכורה בManifest, ובו שילובים חכמים בقوي עץ הזמנים, Năm תקנות וסידור הדורשים במערכי ודיעה, הבכורה בManifest, ובו שילובים חכמים בقوي עץ הזמנים, Năm תקנות וסידור הדורשים במערכי ודיעה, הבכורה בManifest, ובו שילובים חכמים בقوي עץ הזמנים, Năm תקנות וסידור הדורשים במערכי ודיעה, הבכורה בManifest, ובו שילובים חכמים בقوي עץ הזמנים, Năm תקנות וסידור הדורשים במערכי ודיעה, הבכורה בManifest, ובו שילובים חכמים בقوي עץ הזמנים, Năm תקנות וסידור הדורשים במערכי ודיעה, הבכורה בManifest, ובו שילובים חכמים בقوي עץ הזמנים, Năm תקנות וסידור הדורשים במערכי ודיעה, הבכורה בManifest, ובו שילובים חכמים בقوي עץ הזמנים, Năm תקנות וסידור הדורשים במערכי ודיעה, הבכורה בManifest, ובו שילובים חכמים בقوي עץ הזמנים, Năm תקנות וסידור הדורשים במערכי ודיעה, הבכורה בManifest, ובו שילובים חכמים בק

Technion - Computer Science Department - M.Sc. Thesis  MSC-2018-23 - 2018

תודות

אני רוצה להודות למנחת, לחרותי ולחבריו עבורי ת униיסים רבה.
希ולוב ניתוח טקסט במסדי נתונים רלציונליים

הכותרת על המחבר

לש מילה חלקי של הדרישות לקבלת התואר
מניסיון 클לידעни מבית המואהב

יואב חסן

הנהלת ההכנת מצגת - Macro Command Line
2018

סיוון התחדשות

היפה, מאי 2018

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-23 - 2018
shallub ניתוח טקסט במשדי נתונים רלציונליים

יואב נחשון