Update-Aware Information Extraction

by

Besat Kassaie

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2023

Examining Committee Membership

The following served on the Examining Committee for this thesis.

External Examiner:	AnHai Doan Vilas Distinguished Achievement Professor Gurindar S. Sohi Professor Department of Computer Science University of Wisconsin
Supervisor:	Frank Wm. Tompa Distinguished Professor Emeritus and Adjunct Professor The David R. Cheriton School of Computer Science University of Waterloo
Internal Members:	Ihab Francis Ilyas Professor The David R. Cheriton School of Computer Science University of Waterloo
	Florian Kerschbaum Professor The David R. Cheriton School of Computer Science University of Waterloo
Internal-External Member:	Arie Gurfinkel Professor The Department of Electrical and Computer Engineering University of Waterloo

Abstract

Information extraction programs (extractors) can be applied to documents to isolate structured versions of some content by creating tabular records corresponding to facts found in the documents. When extracted relations or source documents are updated, we wish to ensure that those changes are propagated correctly. That is, we recommend that extracted relations be treated as materialized views over the document database.

Because extraction is expensive, maintaining extracted relations in the presence of frequent document updates comes at a high execution cost. We propose a practical framework to effectively update extracted views to represent the most recent version of documents. Our approach entails conducting static analyses of extraction and update programs within a framework compatible with SystemT, a renowned extraction framework based on regular expressions. We describe a multi-level verification process aimed at efficiently identifying document updates for which we can autonomously compute the updated extracted views. Through comprehensive experimentation, we demonstrate the effectiveness of our approach within real-world extraction scenarios.

For the reverse problem, we need to translate updates on extracted views into corresponding document updates. We rely on a translation mechanism that is based on value substitution in the source documents. We classify extractors amenable to value substitution as stable extractors. We again leverage static analyses of extraction programs to study stability for extractors expressed in a significant subset of JAPE, another rule-based extraction language. Using a document spanner representation of the JAPE program, we identify four sufficient properties for being able to translate updates back to the documents and use them to verify whether an input JAPE program is stable.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Prof. Frank Wm.Tompa, for his unwavering guidance, insightful feedback, and constant support throughout the journey of my PhD studies. His expertise and mentorship have been invaluable in shaping the direction and quality of this work. I also thank other members of my committee, Prof. AnHai Doan, Prof. Arie Gurfinkel, Prof. Ihab Francis Ilyas, and Prof. Florian Kerschbaum for taking the time to review and improve this work.

The motivation for pursuing this research direction stemmed from the challenges we encountered while working with medical records from the School of Optometry and Vision Science at the University of Waterloo. I extend my gratitude to Prof. Elizabeth Irving for generously providing us with the dataset and for the valuable and fruitful discussions during our frequent meetings.

Some of the results of this thesis have been published in the following papers:

Besat Kassaie and Frank Wm. Tompa. Predictable and consistent information extraction. In Proceedings of the ACM Symposium on Document Engineering, pages 14:1–14:10, Berlin Germany, 2019. ACM.

Besat Kassaie and Frank Wm. Tompa. A framework for extracted view maintenance. In DocEng '20: ACM Symposium on Document Engineering 2020, Virtual Event, CA, USA, September 29 - October 1, 2020, pages 16:1–16:4. ACM, 2020.

Besat Kassaie and Frank Wm. Tompa. Autonomously computable information extraction. Proc. VLDB Endow., 16(10):2431–2443, Aug 2023.

Dedication

This thesis is dedicated to Alireza Vazifedoost, whose patience, understanding, encouragement, and belief in my abilities have been a source of strength and inspiration.

To my dear Barzin and Ava, this thesis stands as a living proof that obstacles fade when you choose not to acknowledge their presence.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Textual data is a convenient medium to communicate and store knowledge. There is a huge body of research in developing techniques to process and understand text by computers. Text understanding can be realized by tasks such as text sentiment analysis, designing a mathematical model to represent the language in the text and producing similar text using that model (language modeling), or finding pieces of a text that are of interest such as mentions of people's names. To perform these tasks, text can be represented in various formats such as a bag/sequence of words, a graph, a relational table, or a combination of the formats. In this thesis, we focus on one of the computationally convenient formats, namely, relational tables.

Information extraction identifies and isolates words and phrases within documents and stores them in relational tables in order to present the underlying data in a structured form (Figure 1.1). When the semantics of the extracted tuples are predetermined through a relational schema, the extraction process is classified as *closed* information extraction. On the other hand, information extraction techniques that do not rely on a predefined schema and extract facts and their relationships in the form of a general tuple $\langle subject, predicate, object_1, \cdots, object_n \rangle$ are referred to as *open* information extraction [76]. In this thesis, our focus lies within the former category, and when we mention extractor, we are always referring to a closed information extraction technique.

Research in the area of information extraction was first promoted through the Message Understanding Conferences (MUCs) [44]. The extraction process has since evolved to address specific concerns, such as efficiency and accuracy. Recent research in this area includes designing new languages and extraction platforms [10, 43, 81], choosing an appropriate algorithmic approach respecting the domain and the syntactic and semantic properties of anticipated data sources and outputs [82], facilitating the incorporation of human knowledge in algorithm design [20], and adapting existing extractors to deal with new documents added to the system [22]. Despite many technical differences, most of

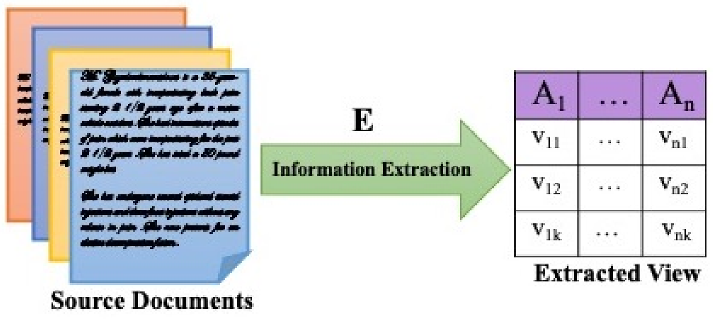

Figure 1.1: Information extraction processes text and creates a structured representation of its content.

the proposed extraction approaches share a subtle and important assumption, which we call "fading attachment." The flow of information between the three main components of information extraction—source documents, the extraction program, and the extracted relations—is maintained during the development period but evaporates once the extraction program reaches a satisfactory level of accuracy and robustness. Once deployed, the information extraction process ignores the relationship between the contents of the source documents and the extracted relations.

We observe that the fading attachment assumption is inappropriate in many applications. Extracted relations might be modified due to privacy concerns [54] or for data cleaning purposes [48] (for example, the collection of extracted records might disclose some inconsistencies among the source documents), but thereafter they are inconsistent with the contents of the source document. Consider, for example, the problem of maintaining privacy for personal information contained in a document collection of electronic health records. A feasible approach to protecting documents can be applying a suitable perturbation algorithm to the table(s) obtained from a document collection through information extraction. However, to keep the documents consistent with the tables, regardless of the nature of updates to extracted relations, a mechanism is essential to translate updates on relations to updates over source documents.

On␣**03/24**,␣we␣rented␣and␣watched␣‘**A␣Man**’.␣

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41

I␣highly␣recommend␣it␣as␣an␣inspiring␣and

42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82

␣humorous␣film␣to␣enjoy.

83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106

(a) Original document. The substrings in **red** and **orange** undergo updates for various reasons.

On␣03/24,␣we␣rented␣and␣watched␣‘**A␣Man␣Wi**

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41

th␣Cats’.␣I␣highly␣recommend␣it␣as␣an␣ins

42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82

piring␣and␣humorous␣film␣to␣enjoy.

83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116

(b) Updated document I.

On␣**2023/24/03**,␣we␣rented␣and␣watched␣‘A␣M

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41

an’.␣I␣highly␣recommend␣it␣as␣an␣inspirin

42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82

g␣and␣humorous␣film␣to␣enjoy.

83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111

(c) Updated document II.

Figure 1.1: A sample input document and its updated versions are provided, with associated offsets indicated beneath each character, starting from 1.

1.1 Motivating Example I

Consider a simple scenario. The goal is to extract all movie names within the collection of documents, individually replace any shortened movie name with the corresponding full title, and finally release the data. For simplicity, we consider a sequence of two, three, or four upper-initial words as a movie name if it is surrounded by $\boxed{\text{‘}}$ $\boxed{\text{’}}$ [1] *and appears in a sentence with terms such as* $\boxed{\texttt{watched}}$ *or* $\boxed{\texttt{rented}}$*; words are considered in the same sentence if there are no end punctuation characters such as* $\{\boxed{?}, \boxed{.}\}$ *between them. Let* Σ *represent the set of Latin alphanumeric characters. The extraction semantics can be represented using a regular expression with variables. We use the variable* M *to mark each matched substring:*

$$\begin{aligned}\gamma_m = &\Sigma^*\boxed{\llcorner\!\lrcorner}(\boxed{\texttt{watched}} \vee \boxed{\texttt{rented}})\gamma_b\gamma_s^*\boxed{\llcorner\!\lrcorner\,\text{‘}}M\{(\gamma_u\gamma_l^*\boxed{\llcorner\!\lrcorner}\gamma_u\gamma_l^*) \vee (\gamma_u\gamma_l^*\boxed{\llcorner\!\lrcorner}\gamma_u\gamma_l^*\boxed{\llcorner\!\lrcorner}\gamma_u\gamma_l^*)\vee \\ &(\gamma_u\gamma_l^*\boxed{\llcorner\!\lrcorner}\gamma_u\gamma_l^*\boxed{\llcorner\!\lrcorner}\gamma_u\gamma_l^*\boxed{\llcorner\!\lrcorner}\gamma_u\gamma_l^*)\}\boxed{\text{’}}\gamma_b\Sigma^*\end{aligned}$$

where $\gamma_u = [\boxed{A}, \boxed{Z}]$, $\gamma_l = [\boxed{a}, \boxed{z}]$, $\gamma_b = \boxed{\llcorner\!\lrcorner} \vee \boxed{,}$, *and* $\gamma_s = \Sigma \setminus \{\boxed{?}, \boxed{.}\}$. *Note that we adhere to the ‘all matches’ semantic for regular expression matching. To simplify the process, we assume that string values are assigned to variables and get extracted for every match.*

After applying the extractor to the input text in Figure 1.1a, we replace the movie name $\boxed{\texttt{A␣Man}}$ *in the extracted relation (Figure 1.2a) with its complete form* $\boxed{\texttt{A␣Man ␣With ␣Cats}}$ *resulting in Figure 1.2b. The key question is whether we can update the source document (Figure 1.1a) in such a way that we will extract the new value if we repeat the extraction process.*

The central notion is that both extracted relation and source documents are two representations of the same information, and they need to stay consistent. Consequently, when updates are made to the extracted relations, a translation mechanism is required to reflect these new values back into the source document. In this thesis, we adopt an intuitive approach to the translation process: we replace the old values with the new values in their corresponding positions within the source document.

After applying this simple update to the source document in Figure 1.1a, we get the document in Figure 1.1b. Re-running the extractor over the updated document would produce the updated relation (Figure 1.2b) because: I) $\boxed{\texttt{A␣Man ␣With ␣Cats}}$ is extracted, II) no new rows appear, and III) no rows disappear. It is important to note that in practical scenarios, we often work with complex extractors that involve multiple, and potentially conflicting, expressions. In this thesis, our objective is to establish whether, for a given

[1]Throughout this thesis, characters in Σ appearing in a formula are represented $\boxed{\texttt{like this}}$ to distinguish them from a regular expression's meta-characters.

M
A ␣Man

(a) Original extracted string relation.

M
A␣Man ␣With ␣Cats

(b) Updated string relation.

A	T
$[70, 79\rangle$	$[93, 97\rangle$
$[84, 92\rangle$	$[93, 97\rangle$

(c) Original extracted relation.

A	T
$[75, 84\rangle$	$[98, 102\rangle$
$[89, 97\rangle$	$[98, 102\rangle$

(d) Updated extracted relation.

Figure 1.2: Extracted relations and their update versions for motivating examples.

extractor, it is possible to update the relation and reflect the update back in the source document for all possible source documents.

1.2 Motivating Example II

On the other hand, source documents might also be modified, perhaps for versioning purposes or to accommodate updates that reflect the most recent data; but again the extracted relations become inconsistent with the content in the source documents. For example, many records are added, removed, or modified in the *DBLP* database monthly, see [1]. These changes need to be reflected in extracted bibliographic profiles.

The objective now is to identify all occurrences of words attributing qualities, e.g., $\boxed{\texttt{inspiring}}$*, to some movies. For simplicity, we define an attributing word associated with a movie as one that appears in the same sentence as any of the terms:* $\boxed{\texttt{film}}$*,* $\boxed{\texttt{movie}}$*, or* $\boxed{\texttt{flick}}$*. The pattern can be represented using the following regular expression:*

$$\gamma_{mv} = \Sigma^* \boxed{\llcorner\!\lrcorner} A\{\boxed{\texttt{inspiring}} \vee \boxed{\texttt{boring}} \vee \boxed{\texttt{humorous}}\}\gamma_b \gamma_s^* \boxed{\llcorner\!\lrcorner} T\{\boxed{\texttt{film}} \vee \boxed{\texttt{movie}} \vee \boxed{\texttt{flick}}\}\gamma_b \Sigma^*$$

where subexpressions and the matching semantics are defined as illustrated in Example 1.1 and the variables A and T specify the fields to be extracted.

For this example, assume that instead of extracting the matching text, we instead extract the offsets of matched substrings. Running this extractor on the text shown in Figure 1.1a yields the extraction of two records, forming the relation in Figure 1.2c.

Assume now that we execute a global update as follows: within each document in the database, we re-order Month and Day (separated by a slash) and insert $\boxed{\texttt{2023/}}$ *at the beginning of each instance. We can represent the search part of the update using a regular* *expression:*

$$\gamma_{date} = \Sigma^* \boxed{\texttt{␣}}\, F\{M\{\gamma_d\gamma_d\}\boxed{\texttt{/}}D\{\gamma_d\gamma_d\}\}\boxed{\texttt{␣}}\Sigma^*$$

where $\gamma_d = [\boxed{0}, \boxed{9}]$ *and* F *marks those substrings in the input text that are replaced by new values. Running* γ_{date} *on the text presented in Figure 1.1a results in marking two substring as* M *and* D *to form a record in the following update relation:*

F	M	D
$[4,9\rangle$	$[4,6\rangle$	$[7,9\rangle$

Once we've located the regions of the text for update, we wish to replace all the substrings marked as F *by new values. The replacement semantics can be described by the following expression:*

$$U_{date} = \boxed{\texttt{2023/}}\$(D)\boxed{\texttt{/}}\$(M)$$

where $\$(D)$ *and* $\$(M)$ *are named back-references, as defined in several programming languages. The result of the update is depicted in Figure 1.1c. Following document updates, the extracted relation must again reflect the new values (Figure 1.2d). A natural solution is to run the extractor on the updated document. However, when we deal with many documents the extraction process is prohibitively slow, a concern highlighted by various researchers [22, 86].*

In the current example, by considering γ_{date} *and* U_{date}*, we observe that every substring marked as* F *expands by five character after the update. Leveraging this insight, rather than re-running the extractor over the updated corpus (or, preferably, isolated parts of it only), we can compute the updated extracted relation (Figure 1.2d) using only the expansion factor, the update relation, and the original extracted relation.*

More specifically, we can prove that there is no input document for which the extracted regions (for this example marked as A and D) overlap with a potential update region whether it contains the original value or the new value. This straightforward realization, along with some other conditions discussed throughout the thesis, can be used to prove that the updated relation can be computed for the current example, without re-extraction. In this thesis, our approach involves a static analysis of the provided extraction expression, the update expression, and the replacement specification. We are able to prove sufficient conditions under which we can avoid re-extraction and, instead, simply update the extracted relation using a *shift* function for all possible input documents.

1.3 Extracted Relations as Materialized Views

In this thesis, we propose that an extracted relation can be treated as a materialized view of the database of documents. To this end, we require extractors to have three general characteristics: i) strict: for every possible input document the set of extracted values in the corresponding record is a subset of words and phrases appearing in the input. That is, each extracted value comes from a continuous span in the documents. Hence, a hypothetical extractor that mines the input text and infers information that does not appear explicitly in the document is not a strict extractor[2]; ii) computable: for all possible input documents and corresponding extracted attributes, we have access to positions from which the attributes are extracted. Some extraction mechanisms directly extract positions from text while considering corresponding string values as by-products of extraction process, whereas others do the reverse; iii) deterministic: for every possible set of input documents, the set of extracted values remains consistent across multiple runs of the extractor; some extractors, such as document spanners, are inherently deterministic, but others, such as some written in JAPE, are not, as discussed on Section 4.3.4.

Treating extraction algorithms as a view mechanism will allow extractors to be adopted in a broader range of applications. For instance, finding and repairing violations of constraints in a large corpus of uncleaned documents is a difficult problem: an extractor that guarantees the preservation of consistency between the source text and the extracted relations can be adopted to solve this problem. When extraction time is a bottleneck and updates to source documents occur frequently, it has been proposed to apply incremental update for the extracted relations [22]. From this perspective, the problem of updating extracted views resembles the classical view update problem for relational databases [30], and the problem of updating source documents resembles the problem of maintaining materialized views [46]. Both problems are well-motivated and thoroughly addressed in the relational setting. However, due to the diverse range of extraction techniques and inherent complexity in text processing and understanding, tackling these problems for databases of documents introduces new challenges.

Our hypothetical extraction system supports updates to documents and extracted relations. Thus, the system comprises five components: a collection of documents, a set of extraction programs, a corresponding set of extracted relations, an instance of a document update specification, and an instance of a view update specification (Figure 1.3). In this thesis, we develop solutions for managing updates for rule-based information extraction

[2]We leave to future research the exploration of whether the extractor can still be considered to be strict when the inference is realized through a bijective function, mapping words or phrases in the input documents to other values present in the extracted table.

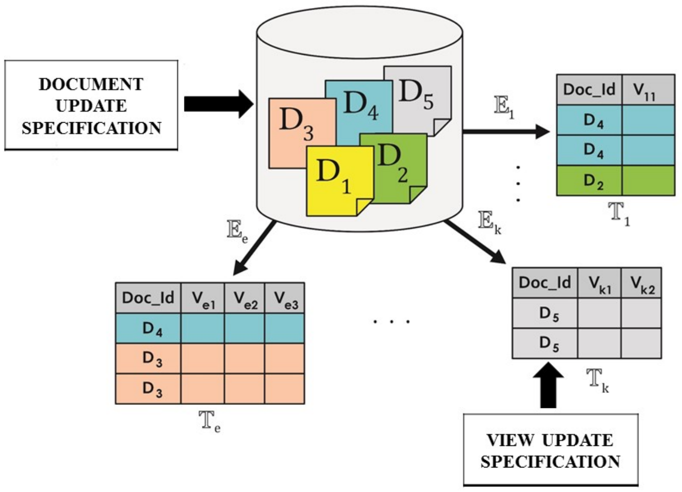

Figure 1.3: Extraction system that supports updates to source documents as well as extracted views.

systems, such as GATE [28] and SystemT [81], which are based on the theory of regular languages. We address the challenges associated with each problem using one of these frameworks. Our aim is to demonstrate that our approach is not limited to a specific framework but can be applied to any extraction mechanism developed based on the theory of regular languages. SystemT benefits from a formal model, *document spanners*, which facilitates analysis of extraction programs. Unfortunately, GATE lacks such a mathematically rigorous formalization, which hinders the study of the properties of extraction programs. Consequently, a significant portion of Chapter 4 is devoted to devising a conversion mechanism from JAPE [29], a rule-based language underlying GATE, to document spanners, bridging the gap and facilitating a more comprehensive study of properties for a JAPE program.

1.4 Information Extraction in the Era of Pre-trained Language Models

Pre-trained Language Models (PLMs) such as BERT [32] and particularly Large Language Models (LLMs) [97, 102] have demonstrated impressive performance across a wide range of text processing and understanding tasks, including question answering [91], code/text generation [103], and relation extraction [18]. With such success, is it still necessary to represent text in a well-structured form like relational tables?

There have been studies that indicate PLMs are not as proficient at handling certain tasks, especially complex natural language queries that require non-trivial reasoning or aggregation [93, 90]. One effective approach for harnessing the potential of LLMs is to treat them as auxiliary tools rather than the central component of the system. This strategy allows applications to accommodate the shortcomings of LLMs. The advances in LLMs have not detrimentally impacted the relevance of other technologies, like Information Extraction, for the majority of medium-scale applications. For instance, relational databases have a strong reputation for handling complex and large queries with no strict limits on input size. Furthermore, recent studies have demonstrated that LLMs excel at translating natural language queries into SQL queries [80]. In a recent work [90], Tan envisions the development of a hybrid system capable of seamlessly managing both simple and complex queries, while harnessing the power of structured query answering (SQL) to address complex natural language queries. The proposed approach involves the creation of a relational lens over documents (information extraction), and leveraging LLMs to transform complex natural language queries into SQL queries. Therefore, information extraction continues to be a crucial component in many document processing pipelines.

1.5 Novelty and Contributions

1.5.1 Novelty

The fundamental realization that information extraction can be viewed as a mechanism for document database management has not been previously recognized in the context of unstructured data management. This new perspective facilitates a variety of improvements in text processing similar to the ones identified in relational database views. Particularly, the dual problems of efficiently maintaining materialized views (Chapter 3) and of updating documents to reflect updates applied against extracted views (Chapter 4) open up many opportunities for developing processing techniques that will ultimately lead to practical solutions.

While the former problem has been explored within the research community, the formulation adopted in this thesis introduces two novel perspectives:

1. We focus on global updates, which are expressed using a formal language and applied to all documents in the database at once. The optimizer is fully aware of the affected files, the updated regions in each document, and, most importantly, the semantics of the update. This contrasts with earlier work by Chen et al. [22], who examined updates in a setting that is agnostic to the type of update itself, choosing instead to compare updated versions of documents to their original counterparts in order to discover their points of difference. This thesis represents an innovative approach as we take on the challenge of optimizing unstructured data management in the context of global updates.

2. In this work, we consider the extraction program to be *transparent*, not as a blackbox for which we must infer programs' properties from their runtime behaviour. For instance, Shen et al. [86] conducted blackbox analyses of extraction programs manually to gain insights into their general characteristics, from which they could develop an optimizer. Or Chen et al. [22] assume that such chracteristics (scope and context) are given to the optimizer by the developers of the extraction program. We perform static analyses on the extraction program to validate multiple sufficient conditions. This approach ensures that the runtime of our optimizer remains independent of the database size, which is highly advantageous, especially when dealing with large databases.

In the context of unstructured data management, we are pioneers in identifying, formulating, and tackling the latter problem, namely, extracted view update. We anticipate that the immediate application of our research will involve the identification and resolution of numerous impactful research problems in the unexplored area of unstructured data quality.

1.5.2 Contributions

Our detailed contributions to both problems are summarized as follows:

1. For the extracted view maintenance problem:

- We introduce extracted view maintenance.
- We propose a document update model that is compatible with SystemT and can express global insertions, deletions, and replacements across a document collection.
- We formalize three categories of document updates for which we can preserve consistency without repeating the extraction process: *irrelevant*, *autonomously computable*, and *shiftable updates*.
- We propose polynomial algorithms to determine whether an update is shiftable with respect to extractors expressed as document spanners, a formalism that models the basis of the SystemT extraction system.
- We design and implement a verification system based on the proposed formalism.
- Finally, we conduct experiments to show the effectiveness of the proposed optimization strategy in practice and to discuss opportunities for areas of improvement.

2. For the extracted view update problem:

    - We identify and formalize the extracted view update problem.
    - We formalize a general view update model, i.e., *domain preserving updates* along with an intuitive update translation mechanism.
    - We introduce a sufficient property called *stability* for extraction programs for which we are able to translate view updates to document updates.
    - We propose a mechanism to convert a program in JAPE to its corresponding document spanner.
    - We present a verification process that can be applied to a JAPE program's spanner representation to determine whether the program is stable.

Chapter 2

Related Work

In this chapter, we thoroughly review the literature's most relevant work, in conjunction with the key factors that motivated our focus on specific systems.

2.1 Information Extraction

Information extraction is a crucial step in processing and understanding text. By identifying strings of interest in unstructured or semi-structured data sources, extractors provide selected data to populate relational records. For instance, an application can extract the relationships between events, dates, and venues as described in a collection of documents.

Information extraction might be the main task or a sub-task in broader projects, such as information retrieval and text analytics. For instance, *Knowledge Panels* [88], which are used to improve the presentation of retrieval results in a commercial search engine, is an application of information extraction in information retrieval systems.

2.1.1 Approaches Used for Information Extraction

Sarawagi presents a thorough review on information extraction [84]. There are two primary approaches for developing information extraction systems:

Rule-Based Approaches. In rule-based systems, handcrafted rules (or sometimes learned rules) are developed based on features of entities and possibly relationships between entites that are derived from a training corpus. There are two classes of rule-based approaches:

```
Phase: Names
Input: Token
Options: control = appelt
Rule: capital
(({Token.orth==upperInitial})+):mark-->:mark.Capital={}
```

Figure 2.1: A JAPE phase that marks the longest sequence of upperInitial tokens.

- **Grammar-based**: Extraction systems that use rule-based languages, such as the Common Pattern Specification Language (CPSL) [10], fall under this category. In such systems, input texts are matched against user-defined rules. Extraction rules may be organized into multiple phases, with each phase augmenting the text with various annotations and passing it on to the next phase. For instance, in CPSL and its extension JAPE [29], a rule consists of two parts: a pattern expressed as a regular expression over lexical features of the input text (e.g., syntactic features like noun, proper noun, verb, or orthographic features such as punctuation and capitalization), and an action specifying the interpreter's response when the input matches the pattern. Figure 2.1 depicts a JAPE phase containing a single rule.

  The fundamental drawbacks of grammar-based systems have been highlighted in some studies [86, 81] and primarily revolve around two aspects. First, the readability, scalability and maintainability of extraction programs suffer as their complexity and size increase. Grammar-based systems offer ad-hoc abstraction and modularization. For example, developers can write their programs in multiple phases (modules) in JAPE, but rules are applied procedurally (similar to procedural languages), making it difficult to manage and understand large programs. Second, grammar-based systems exhibit slow performance due to their sequential nature of evaluation. The sequential evaluation process can be inefficient, especially when dealing with large volumes of data. These systems often lack systematic potential for optimization, aside from small opportunities for fine-tuning.

- **Declarative Extraction Systems**: To address shortfalls of pure grammar-based systems, Shen et al. [86] propose a declarative approach to information extraction using Datalog with embedded procedural predicates to express extractors. This enables the composition of large extraction programs from small extraction modules. Additionally, it supports the creation of execution plans for the extraction program and deployment of cost-based optimization techniques similar to query optimization in relational databases. Similarly, Reiss et al. [81] recommend using a declarative language, AQL (used by SystemT), to be able to exploit query optimization strategies.

Statistical Approaches: It is a common practice in natural language processing to devise a statistical technique, perhaps a new machine learning pipeline, for tackling a specific extraction task such as named entity recognition, event detection, and argument extraction. In this work, we do not focus on any particular extraction task, instead, the problems formulated in this thesis require only that the extractor is strict, computable, and deterministic, as explained in Section 1.3. Verifying whether an extractor is strict is straightforward; it is something that the designer decides beforehand. In the remainder of this section, we review some extraction tasks that can be designed as strict extractors.

Various machine learning algorithms are applied in the information extraction domain. Some techniques, such as conditional random fields [60] and LSTM [47], are sensitive to the sequential nature in text. Other techniques such as Support Vector Machines [27] are trained based on word level features to identify named entities [50]. Furthermore, deep learning techniques have demonstrated their efficacy in information extraction tasks, particularly when trained on large corpora. The deep learning models learn complex patterns and generalize well, leading to accurate extraction of information from diverse sources [75], and they are appealing when the reasoning behind particular data being extracted does not need to be explained. Advances in devising PLMs, based on transformer architecture, such as BERT [32], GPT3 [14], and GPT4 [16], have significantly enhanced various natural language processing tasks, including information extraction.

- **Pre-trained Language Models:** Three strategies have been taken in adopting PLMs for information extraction: I) *Fine-tuning*: a model is initially trained on a large volume of unlabeled data to initialize its parameters. Subsequently, these parameters are adjusted using labeled data specific to a downstream task in a process known as fine-tuning. It is important to note that each downstream task typically involves separate fine-tuned models, even though they all start with the same pre-trained parameters. For instance, Obuchowski et al. [77] fine-tune a pre-trained Polish language model for entity extraction in medical reports written in Polish. In another work, for an underrepresented natural language, a language model is constructed and then fine-tuned for named entity detection and part-of-speech tagging [98]. II) *In-context learning*(ICL): the extraction task is formulated as a language generation task[1]: given a sequence of one or more inputs, i.e., prompts, the model generates the most likely output [14]. Unlike in fine-tuning, parameters of the model are not

[1]The model generates human-readable text, based on certain input.

updated in ICL. For a given task, the research is devoted to forming an appropriate input format, i.e., prompt tuning. Agrawal et al. [7] explore the potential of ICL for clinical extraction tasks such as biomedical evidence extraction and coreference resolution. To perform named entity detection, Li et al. [63] propose to generate a code-style representation of the input text, Python-like code, and prompt a LLM trained specifically for code, i.e., Code-LLMs, such as Codex [23]. III) *Hybrid*: in hybrid settings both fine-tuning and in-context learning are used. Josifoski et al. [52] use LLMs to generate many synthetic labeled samples, which can subsequently be used to fine-tune a PLM. Yubo et al. [67] propose to combine a LLM with a small PLM for multiple extraction tasks, including named entity detection and event/event argument detection. They propose to use LLMs when the PLM exhibits low confidence in its responses, an approach that can be adopted for various tasks.

There are hybrid approaches that aim to leverage the strengths of both rule-based systems and statistical methods to improve extraction accuracy and performance. For instance, Califf and Mooney have designed a machine learning model to learn extraction rules [19].

2.1.2 Update-Aware Information Extraction

Expectations from extractors have risen as requirements have become more diversified, from the point that there were no criteria to evaluate their performance [42] to the point that extraction algorithms need to work under various stresses such as noisy data, low response time, and diverse types of input and output [84]. In this section we review related work to an update-aware information extraction mechanism.

Efficient Information Extraction

Extraction time can be a bottleneck for many applications [86, 81], and therefore efficient processing is an important consideration for information extraction. Some optimization approaches are general and can be deployed in any system for specifying extractors. For instance, Chandel et al. [21] propose an efficient algorithm for dictionary-based entity recognition, which can be used on many extraction platforms. Iperotis et al. [49] deal with many documents by keeping only “promising” documents for the extraction process. As previously mentioned, Shen et al. [86] and Reiss et al. [81] leverage the advantages of declarative languages to design efficient extraction mechanisms.

Considering extractions within larger applications, Jain et al. [51] treat information extraction and subsequent relational queries as an integrated system and propose optimizations that takes into account the characteristics of extractors, document retrieval methods, and join algorithms on the extracted relations.

However, none of these strategies consider that an extraction might need to be recomputed to keep extracted information synchronized with source documents as they are updated. Recognizing this situation, Chen et al. have developed an approach for incrementally updating extracted relations [22]. They do not assume that a description of the update is available, but instead compare each updated document with the previous version to find regions that have not changed. Then, based on user-provided properties of the extractor, they decide which of the extracted items from those regions can be reused. Doleschal et al. [34] explore conditions for determining that a spanner is *split-correct*, that is, if the extracted relation can be computed by combining the extractions from sub-documents. If so, extractions from various sub-documents can be run in parallel, but additionally incremental update is applicable: re-extraction after an update can be avoided for those sub-documents that are not altered. With a different prespective, Lerman et al. [61] propose continuous adaptation of extractors as their information sources changes. Lerman et al. monitor updates to information sources for a specific class of extraction algorithms (wrappers) and rebuild the extractor if the extractor's performance (precision and recall) decreases due to the updates over their sources. For our application of interest concerning translating updates on extracted views to updates to source documents, we identify extractors that can tolerate supervised updates over the sources without applying any adjustments to the extractor itself.

Updatable Extracted Views

The body of research related to the problem of updatable views over unstructured data is relatively limited. However, we can draw insights from existing work on updatable views over semi-structured data, such as XML, which is slightly relevant to our study. Similar to the relational setting, creating views over XML databases can provide various advantages, including faster query processing and convenient access control over specific sections of a larger XML database [6]. Kozankiewicz et al. [59] propose to incorporate information about forseeable updates over views into the view definition. Therefore, the ultimate affects of updates are specified solely by the query designer which, if not verified, might leave the XML database in an incorrect state .

2.1.3 Rule-based versus LLM-based Extraction

The interpretability of rules makes rule-based information extraction systems suitable for domains in which a user's confidence in the validity of the extraction algorithm is critical. Chiticariu et al. [26] have identified the advantages of rule-based systems, namely easy comprehension and maintainability, that make them appealing for commercial uses. There are several disadvantages that come with PLMs, including LLMs, which make them less appealing, specifically when they are used in critical domains such as legal or medical fields:

- **Computable Extractor:** Being a strict extractor implies that the extracted items must occur in the input text, and computability requires that the extraction process is capable of generating the necessary provenance. Because, not all instances of a term or phrase in the source document may correspond to those that are extracted, we require a mechanism for identifying the corresponding positions, i.e., fine-grained data lineage. Extractors written in certain rule-based extraction languages, such as AQL (used in SystemT) and JAPE (used in GATE), are inherently computable, i.e., the lineage of extracted items are available as a by-product of the extraction process. However, extractors expressed as PLMs do not come with these inherent capabilities. PLMs are complex and perceived as black-box solutions. As a result, a significant body of research is dedicated to inventing novel techniques to explain how PLMs operate [101]. The fine-grained lineage of extracted items can be considered a *local explanation* mechanism[2], which aims to provide insight into how a model responds to a specific input instance. Several techniques belong to this category, among which explanations based on *attribution* might offer a viable mechanism for pinpointing corresponding positions in the input. An attribution-based explainer assigns a relevancy score to each input word, highlighting its contribution to generating the output (extracted items, in the extraction case). In our work, computability is treated as a non-probabilistic property, whereas lineage based on attribution is inherently probabilistic [100], so current PLMs do not meet our requirement for computability.

- **Deterministic Extractor:** The generated outputs in PLMs are the outcomes of a combination of multiple stochastic and/or heuristic steps. Consequently, running a PLM multiple times for the same input can yield different responses [94, 5], which characterizes LLMs as non-deterministic extractors. Again they do not meet the requirements required for update-aware information extraction.

In addition to these major drawbacks for our purposes, today's LLMs have a few other disadvantages. PLMs are not knowledge-bases themselves; they are probabilistic models of

[2]For a comprehensive overview of explainability, refer to [101].

knowledge-bases. Therefore, they have the potential to generate non-factual answers [33]. They also have limitations when it comes to their input size[3]. Furthermore, they are costly [4] and rely on extensive data. Therefore, it is not likely that average-sized enterprises will be able to develop their own LLMs, making them reliant on external sources, which potentially contain outdated data.

2.2 Fine-grained Data Lineage.

Data lineage, or provenance, has been defined and formalized for structured and semi-structured data [17, 24]. Given a value that is the outcome of executing a well-defined query over some data sources, often relational tables, provenance determines three aspects related to the value: data points in the source that contribute to form the value, the way that data points collaborate to produce the value, and the exact location(s) in the data source from which the value originates. The last aspect is similar to the notion of lineage that we use in this work, i.e., we require the extractor to provide the positions in a document from which a value is extracted.

Provenance-based techniques have also been applied to information extraction problems. Roy et al. [83] propose a provenance-based technique to improve the quality of extraction by refining the dictionaries that are used in a rule-based extraction system. A set of entries from the dictionaries that have been involved in generating the output are analyzed to determine which should be removed to improve the extractor's performance most. In other work, Liu et al. [66] use provenance techniques to determine the most effective rule refinements, i.e., those that result in removing undesirable tuples and keeping correct ones. Chai et al. [20] examine the provenance of a multi-stage extraction program that can include relational operators on intermediate tables. Users' feedback is expressed as updates over tuples that appear at any stage of the extraction process, and these updates are translated into modifications of the corresponding extraction program.

2.3 Relational Materialized Views

In the relational setting, a view is a relation that is derived from base tables. Since views need to be reconstructed with each access, they may be materialized to provide fast

[3]Currently, the input character limit for ChatGPT is 2048 characters, taken from https://chatgptdetector.co/chatgpt-character-limit/.

access to data. A plethora of problems have been identified and studied in the relational setting. They range from deciding whether to materialize a view to designing efficient strategies for utilizing and maintaining materialized views. In this thesis, we propose to treat extracted relations from documents as views. From this point of view, we deal with two problems: how to efficiently maintain the content of an extracted view when the content of source documents is updated? how to translate updates on tuples in extracted views to updates on documents? In the following two subsections we review some influential work concerning similar research questions in the relational setting. The reader may refer to other work [25, 46, 40] for a thorough review of all aspects of research in the domain of relational materialized views.

Possible actions that invalidate a relational materialized view include: altering the schema of associated base tables or the definition of views, updating the content of corresponding base tables, and updating the content of views independent from base tables. The last two problems are directly related to our research questions, see Figure 2.2.

$$\begin{tikzcd}
R \arrow[r] \arrow[d, "U"'] & V \arrow[d, dashed, "?"] \\
R' \arrow[r] & V'
\end{tikzcd}$$

(a) View maintenance

$$\begin{tikzcd}
R \arrow[r] \arrow[d, dashed, "?"'] & V \arrow[d, "U"] \\
R' \arrow[r] & V'
\end{tikzcd}$$

(b) Updatability of views

Figure 2.2: U is an update process. $R(V)$ and $R'(V')$ represent an instance of the base table (a view) and the base table's instance (a view' instance) after the update, respectively. For 2.2a finding an efficient mechanism to create V' is a possible research problem. For 2.2b the research objective involves determining how to effectively translate U to its most likely intended update over R.

2.3.1 Materialized View Maintenance

For the first research question, we deal with a system in which updates on documents are already completed, and there is a need to efficiently bring the views' content up-to-date. Similarly, in the relational setting, a view definition, expressed in relational algebra, along with tuples to be inserted or removed from base tables provides information that is utilized to maintain the content of a view without recomputing it from scratch [13, 45]. Blakeley et al. address this problem where the definition of materialized views written as a *Select-Project-Join* (SPJ) expression and the update involves insertion or deletion of a set of tuples into/from one base relation at a time. Blakeley et al. derive a Boolean expression from the selection predicate in the view definition. The predicate's attributes that are involved in an insertion or deletion are substituted by associated values taken from an update tuple. The unsatisfiability of the composed Boolean expression implies that regardless of the database instance the view is not affected by the insertion or deletion of tuples into the base relation. They give a sufficient and necessary condition on the irrelevancy of an update which is stronger than our work proposing only sufficient conditions. For a particular class of Boolean expressions, their proposed algorithm is polynomial in the number of attributes present in the view. Furthermore, they formalize a stronger method for materialized view maintenance, *differential view update*. For the differential view update, Blakeley et al. require one more piece of information: the content of associated base relations before the updates. For a view written as an SPJ expression, they utilize a truth table with 2^k rows, where k is the number of updated base relations that participate in a view, to determine sub-expressions in the view definition that need to be re-evaluated. For instance, take a view v that is created by SPJ on two base relations with current instances denoted as a and b. Assume that a set of tuples T are inserted to a. To update the view, Blakeley et al. first compute a partial view using T instead of a in the view definition, this needs access to the content of b. Then they take the union of the partial view and the old view to generate the update-to-date view.

In follow-up work [12] in addition to insertion, Blakeley et al. handle i) deletion which is expressed as a select condition over base tables; ii) modification of base tables defined as a selection of tuples along with an update function that determines how to modify each associated attribute using attributes in the base table that is to be modified. Some necessary and sufficient conditions are devised for detecting irrelevant and autonomously computable updates on relational views. The essence of the proposed approach is to test the satisfiability of specific Boolean expressions that are derived from the view and update description. Particularly for modification, they recommend five properties for various constructed Boolean expressions to be able to autonomously compute the new values in the view after modification to base tables. Similarly, one of our contributions is formalizing a specific kind of autonomously computable update called shiftable update for which modifications to source documents can be applied to the extracted views using a predefined shift function. The main caveat of both studies [12, 13] is that they require testing satisfiability of Boolean expressions that in general is NP-Complete. However, both works present experiments to show the applicability of the approaches in practical scenarios despite being NP-Complete.

2.3.2 Updatability of Relational Views

Our second research question involves translating updates on the content of an extracted view to updates on the content of the associated document. This is similar to the problem of updatability of relational views that is thoroughly studied in relational databases [40, 41, 57, 69, 70].

In the relational setting, the problem of a view updates is defined as finding a translation of a view update (denoted by U in Figure 2.2b) to a database update (denoted by ? in Figure 2.2b) such that running the same view definition query on R' produces V' regardless of the database instance. Interesting research challenges are raised from this definition such as how to deal with the problem of multiple possible translations? Are views always updatable? if not how to identify views that cannot be updated? how to derive a specific translation mechanism for a given view definition, database schema, and update specification?

Two general approaches are proposed for updatability of relational views. First, along with a view definition, all authorized updates and their corresponding translations should be provided. However, the provided translation mechanism also needs to be verified. The second approach is to exploit the information provided by the view definition, the update mechanism, and database constraints to derive conditions on the legitimacy of a translator. For example, the view dependency graph that is constructed using only the view definition and database schema is used to verify a translator for some classes of deletions, insertions, and replacements [31]. Our solution to the extracted view updates is aligned with the latter approach. We limit ourselves to a class of view updates that is realized by a domain-preserving function that maps each extracted value to a value from the same domain, similar to perturbing values of a table to protect privacy. However, we do not impose any constraints on the input documents. We pick the most natural translation which is to substitute old values in the source document with new values, and we expect to see them extracted by running the same extractor (Figure 2.2).

In summary, applying solutions proposed in the relational setting to the information extraction domain poses significant challenges. The relational setting benefits from various constraints, including schema-based constraints such as data types, referential integrity constraints, key constraints, and functional dependency constraints, among others. These constraints serve to structure and regulate the problem space. However, in the context of information extraction, such constraints are generally not present: there are usually no inherent limitations on the content of the source documents.

2.4 Static Analysis of Programs Using Regular Languages

For both research problems, we use an extended form of finite-state automata to statically analyse the extraction program and update mechanism. Similar static analyses of regular expressions or deterministic finite automata have been used in diverse areas, including access control, feature interactions, and vulnerability detection of programs. For example, Murata et al. [73] propose an automaton-based access control mechanism for XML database systems. Regular expressions are derived from given queries, access-control policies, and schemas. Based on the characteristics of the derived automata, element/attribute level access requests by queries are determined to be either granted, denied, or statically indeterminate, independently of any actual input XML documents. An event-based framework is introduced by Kin et al. [58] for developing and maintaining new gestures that can be used in multi-touch environments. Using that framework, application developers express each gesture as a regular expression over some predefined touch events. Regular expressions associated with gestures are then statically analyzed to identify possible conflicts between various gestures. Yu et al. [99] present a method for detecting security vulnerabilities in programs that use string manipulation operators such as *concatenation* and *replacement*. In essence, their approach involves constructing DFAs to represent the program's data dependency graph. Subsequently, they perform static analysis on the provided attack pattern, expressed as a DFA, and the graph's DFAs to detect potential vulnerabilities. Dynamiclly generated SQL queries can enhance the flexibility of programs, written in JAVA or other languages, by allowing them to adapt to changing conditions and requirements without the need to write multiple static queries. However, the host compiler, i.e., JAVA compiler, does not test the generated query strings for possible errors such as type errors. To this end, Wassermann et al. [95] propose a static analyzer to verify the correctness of dynamically constructed SQL queries embedded in programs. Their approach involves creating a DFA representation of the generated query strings and performing static analysis on the DFA.

Chapter 3

Maintenance of Extracted Views

We consider an extracted relation to be a materialized view of the document corpus. From this perspective, in this chapter, we address the problem of keeping extracted relations in sync with the document corpus when the corpus is updated, i.e., *extracted view maintenance.* The natural way to reflect changes in source documents is to wipe out any extracted relations and repeat the extraction process. Although this approach guarantees the preservation of consistency between the source text and the extracted relations, as in the relational database context, extracting relations from scratch can be costly. For instance, in some applications where updates to source documents occur frequently, extraction time might be a bottleneck or, in a distributed setting in which extracted relations and source documents reside in different physical sites, the communication cost for repeatedly transferring newly extracted relations might be significant. Thus, avoiding re-extraction is sometimes highly desirable. That is, we wish to translate updates over documents into differential updates over extracted relations. Although the problem can be studied for various extraction languages, we focus on extractors expressed as *document spanners.* In this chapter

- We introduce the extracted view maintenance problem.
- We propose a match-and-replace document update model where replacement values depend on the strings being replaced.
- We formalize three categories of document updates for which we can preserve consistency without repeating the extraction process: *irrelevant*, *autonomously computable*, and *shiftable updates.*

- We propose a new algorithm to verify that an update is shiftable with respect to an extractor and prove that it runs in polynomial time via three theorems. The algorithm relies on four external tests (two for verifying the precondition and two tests for independence) that are each proven to execute in polynomial time.

- Finally, we show experimentally that our algorithm is practical: it can be used effectively in realistic update and extraction scenarios, and it runs far faster than the time needed for re-extraction. If this approach is combined with incremental update, the overhead is relatively small, and it will often perform much faster.

3.1 Preliminaries

3.1.1 Documents, Regular Expression, and Document Spans

In order to develop specific algorithms, we assume that extracted views are defined using SystemT, an information extraction platform that benefits from relational database concepts to deal with text data sources [81]. SystemT models each document as a single string and populates relational tables with *spans*, directly extracted from the input document. With SystemT users encode extractors with a SQL-like language, i.e., **A**nnotation **Q**uery **L**anguage (AQL), to manipulate tables. AQL offers operators to work directly on text or on the extracted tables (standard relational operators that accept span predicates).

The underlying principles adopted by SystemT have been formalized as *document spanners* by Fagin et al. [37]. Most of the material in this section has been introduced in that work, which contains additional details.

Given a finite alphabet Σ, *regular expressions* over Σ conform to the grammar:

$$\gamma \;:=\; \varnothing \mid \epsilon \mid \sigma \mid (\gamma \,\vee\, \gamma) \mid (\gamma \,\bullet\, \gamma) \mid (\gamma)^* \tag{3.1}$$

where σ is shorthand for the disjunction of all characters in Σ. A set of strings that is recognized by a regular expression conforming to grammar 3.1 forms a regular language denoted by $L(r)$. Given a regular expression r, the corresponding regex tree $\mathcal{T}(r)$ represents the hierarchical structure of r, in which the tree's leaves have labels $\varnothing$, ϵ, or $\sigma \in \Sigma$ and internal nodes have labels $\bullet$, $\vee$, or $*$. For convenience of notation, when writing a regular expression, we follow common practice in allowing the following shorthand: omission of parentheses, relying instead on left associativity of all operations and precedence of $*$ over $\bullet$ over $\vee$; omission of the operator $\bullet$; and use of Σ to represent the disjunction of all characters in Σ.

A document D is a finite string that is generated by Σ, i.e., $D \in \Sigma^*$ (Figure 3.1). A *span* of D, denoted $[i, j\rangle$ ($1 \leq i \leq j \leq |D| + 1$), specifies the start and end offsets of a sub-string in D, which is in turn denoted $D_{[i,j\rangle}$, and extends from offset i through offset $j - 1$.

Example 3.1.1

In the document presented in Figure 3.1, $D_{[86,132\rangle}$ represents the sub-string "Maintaining␣Knowledge ␣ about␣Temporal␣Intervals".

If $i = j$, this denotes an empty span at offset i. Spans $s_1 = [i_1, j_1\rangle$ and $s_2 = [i_2, j_2\rangle$ are identical if and only if $i_1 = i_2$ and $j_1 = j_2$. If $\mathcal{S}$ is the set of all spans of D, then a *span relation* R is a relation that contains spans of D, that is, $R \subseteq \mathcal{S} \times ... \times \mathcal{S}$.

3.1.2 Extractors Expressed by Document Spanners

Regular expressions extended using variables chosen from a set V are called *regular expressions with capture variables* and conform to γ in the grammar $G_S(\Sigma, V)$ as follows:

$$\begin{aligned}
\gamma &:= \varnothing \mid \epsilon \mid \alpha \mid \Sigma \mid (\gamma \lor \gamma) \mid (\gamma \bullet \gamma) \mid (\gamma)^* \mid x\{\gamma\} \\
\alpha &:= \beta \mid \delta \\
\beta &:= \sigma \mid [\sigma, \sigma] \\
\delta &:= \Sigma \mid \delta - \beta
\end{aligned} \tag{3.2}$$

where the terminal symbol Σ is now explicitly included (since documents typically use very large character sets), σ represents the disjunction of characters in Σ, α extends individual characters in Σ, and x represents the disjunction of variables in V. Also to allow proper subsets of Σ, $[\sigma, \sigma]$ represents the disjunction of characters having their encoding between or equal to the encoding of the first and second character in the pair and $\Sigma - \beta$ represents the disjunction of all characters except for those in β, i.e., *exclusion.* Note that exclusion takes precedence over $*$, $\bullet$, and $\lor$. This expression may also be represented as (extended) regex trees, and they may also use the usual shorthand conventions for regular expressions. The use of a sub-expression of the form $n\{g\}$ is to denote that whenever the regular expression matches a string, sub-strings matched by g are to be *marked by the capture variable* $n \in V$. If E is a regular expression with capture variables, then we denote the set of capture variables in E as $SVars(E)$. We distinguish two classes of variables based on their relative

<	a	r	t	i	c	l	e	␣	k	e	y	=	"	c	a	c	m	/	A	l	l	e	n	8	3	"	␣	m	d	a	t	e	=	"	2	0	1	1	-	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41
6	-	0	7	"	>	<	a	u	t	h	o	r	>	J	a	m	e	s	␣	F	.	␣	A	l	l	e	n	<	/	a	u	t	h	o	r	>	<	t	i	t
42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82
l	e	>	M	a	i	n	t	a	i	n	i	n	g	␣	K	n	o	w	l	e	d	g	e	␣	a	b	o	u	t	␣	T	e	m	p	o	r	a	l	␣	I
83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123
n	t	e	r	v	a	l	s	.	<	/	t	i	t	l	e	>	<	/	a	r	t	i	c	l	e	>														
124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150														

Figure 3.1: A sample input document D for our running example.

positioning. A variable $x \in SVars(E)$ is *exposed* if is not enclosed in any other variables, otherwise is *nested*. We use G_S in place of $G_S(\Sigma, V)$ whenever Σ and V are immaterial or understood from the context. Throughout this thesis, characters in Σ appearing in a formula are represented $\boxed{\texttt{like this}}$ to distinguish them from a regular expression's meta-characters.

Example 3.1.2

Let Σ be the set of Latin alphanumeric, punctuation and space characters (the last represented by $\boxed{␣}$). Note that this is the setting for all future examples unless otherwise stated. γ_{date} is a regular expression with capture variables

$$\gamma_{date} = \Sigma^* \boxed{\texttt{mdate="}}\, F\{Y\{\gamma_d\gamma_d\gamma_d\gamma_d\}\boxed{\texttt{-}}M\{\gamma_d\gamma_d\}\boxed{\texttt{-}}D\{\gamma_d\gamma_d\}\}\boxed{\texttt{"}}\Sigma^*$$

where $\gamma_d = [\boxed{\texttt{0}}, \boxed{\texttt{9}}]$ and $SVars(E) = \{F, Y, M, D\}$. F is the only exposed variable in $SVars(E)$. For every successful matching of γ_{date} against a string, a sub-string matching $\gamma_d\gamma_d\gamma_d\gamma_d$ is marked by the capture variable Y. Matching γ_{date} against the document presented in Figure 3.1, $\boxed{\texttt{2011}}$ is marked as Y. We extend conventional set notation to write $Y \subset F$ and $\boxed{\texttt{-}} \in F$.

3.1.3 Matching Model

Just as determining whether or not a string is in a regular language $L(r)$ can be accomplished by executing a finite state automaton corresponding to r, determining membership in a language defined by a regular expression with capture variables can be accomplished by executing a corresponding *variable-set automaton* or *vset-automaton*. A *vset-automaton* is a non-deterministic finite automaton augmented with a set of variables denoted as V, a designated set, and two variable operators. These two specific operators defined on a variable $x \in V$ include: $x \vdash$ ("open x") and $\dashv x$ ("close x"), which symbolize inserting and removing the variable x into/from a designated set, respectively. Besides the standard character transitions, a vset-automaton has operation transitions to operate on variables without consuming the input string. An input string is accepted by a vset-automaton if, after scanning the whole input, we end up in a final state given that every variable has been inserted in and removed from the designated set exactly once. For a thorough review of vset-automata, the reader may refer to the description by Fagin et al. [37].

Applying an information extractor to a document D produces a *span relation*, i.e., a relation that contains spans of D. To this end, if E is a regular expression with capture variables, it specifies a *document spanner*, denoted $[\![E]\!]$, which is a function mapping strings over Σ^* to span relations. In particular, for a given document D, the spanner specified by E produces a span relation $[\![E]\!](D)$ in which there is one column for each variable from V appearing in E, each row corresponds to a matching of E against D when the variables are ignored, and the value in a row for the column corresponding to $x \in V$ is the span marked by x. To ensure that the extracted relation is in first-normal form with no null values, we restrict our attention to a specific class of document spanners, namely *functional document spanners*, that assign exactly one span to each variable for all produced rows, regardless of the input document D.

Example 3.1.3

Let γ_d be as defined in Example 3.1.2. Applying the spanner represented by

$$\gamma_{partialDate} = \Sigma^* F\{M\{\gamma_d\gamma_d\}\boxed{\text{-}}D\{\gamma_d\gamma_d\}\}\Sigma^*$$

to the document in Figure 3.1 results in the span relation in Figure 3.2.

F	M	D
$[38,43\rangle$	$[38,40\rangle$	$[41,43\rangle$
$[41,46\rangle$	$[41,43\rangle$	$[44,46\rangle$

Figure 3.2: The extracted relation $[\![\gamma_{partialDate}]\!](D)$, where D is depicted in Figure 3.1.

Definition 1 *Throughout this chapter, a functional document spanner used for the purpose of information extraction is called an* extraction spanner *or simply an* extractor*, the regular expression with capture variables defining it is called an* extraction formula*, and the span relation produced for a document is called an* extracted relation*.*

For extraction spanners an algebraic operator is defined as a function that maps one (for unary operators) or two extraction spanners (for binary operators) to exactly one extraction spanner. Let D be an input document and let E, E_1, and E_2 be extraction spanners, where the last two are union-compatible, i.e., $E_1 = \varnothing \vee E_2 = \varnothing \vee SVars(E_1) = SVars(E_2)$, and $S \subseteq SVars(E)$. The algebraic operators essential for our work are defined as:

- The *projection* of E is a spanner represented as $\pi_S(E)$. For any arbitrary D, $\pi_S(E)(D)$ is obtained by restricting the domain of $E(D)$ to S.
- For two union compatible extraction spanners, $E_1 \cup E_2$ represents the union spanner where $SVar(E_1 \cup E_2) = SVars(E_1) \cup SVars(E_2)$. For an arbitrary input document we have $E_1 \cup E_2(D) = E_1(D) \cup E_2(D)$.
- The *join* spanner is represented as $E \bowtie E_1$ where $SVar(E \bowtie E_1) = SVars(E_1) \cup SVars(E)$. For an input document the span relation associated with the join spanner, $E \bowtie E_1(D)$ is the cross product of $E(D)$ and $E_1(D)$. However, similar to relational algebra if $SVars(E_1) \cap SVars(E) \neq \emptyset$, tuples must have identical spans assigned to the common variables to be included in $E \bowtie E_1(D)$.
- The spanner resulting from a *binary string selection* operator is denoted as $\zeta^{=}_{x,y}E$ where $x, y \in SVars(E)$. For an arbitrary input document D $\zeta^{=}_{x,y}E(D)$ is obtained by restricting records of $E(D)$ to those that string values corresponding to x and y are identical.

In this thesis, we hypothesize systems that include a document database $\mathbb{D}$ and a set of extractors $\{\mathbb{E}_1, \cdots, \mathbb{E}_e\}$ that run independently over $\mathbb{D}$. The union of span relations produced by an extractor $\mathbb{E}_k$ against the document database is stored in a relation $\mathbb{T}_k$ that includes an additional column to associate each document identifier with the spans for the corresponding span relation (Figure 1.3). These tables serve as materialized views of the document database. In this chapter, we assume each $\mathbb{E}_k$ is an *extraction program* written using the core of SystemT's AQL a.k.a. *core spanner*. A *core spanner* is an element in the closure of extraction formulas under the algebraic operations projection, join, union, and string selection.

3.1.4 Restricted Extraction Formula

For reasons that become clear later on, we are interested in a restricted form of extraction formulas which are defined below. A restricted extraction formula conforms to $\bar{\gamma}$ in the grammar $G_S(\Sigma, V, v)$:

$$\bar{\gamma} \;:=\; (\bar{\gamma} \,\vee\, \bar{\gamma}) \mid (\gamma' \,\bullet\, \bar{\gamma}) \mid (\bar{\gamma} \,\bullet\, \gamma') \mid v\{\gamma''\} \tag{3.3}$$

where γ' is a standard, variable-free regular expression (grammar 3.1), γ'' is a regular expression with capture variables V (grammar 3.2), and v is the only exposed variable. Note that v encloses all other capture variables. The extraction formula presented in Example 3.1.2 conforms to $\bar{\gamma}$ and F is the exposed variable.

Definition 2 *An extraction formula E conforming to $G_S(\Sigma, V, v)$ is in* normalized form *with regard to v if it is written as $\bigvee_{i=1}^{k} E_i$ where each E_i is a formula conforming to $\hat{\gamma}$:*

$$\hat{\gamma} := (\gamma')?\ v\{\gamma''\}\ (\gamma')?$$

where '?' denotes optional and γ' and γ'' are defined in grammars 3.1 and 3.2, respectively.

In short, to normalize a formula conforming to grammar 3.3 with regard to v, all disjunctions that have v in their disjuncts[1] can be "pulled up" over concatenations in the extended regex tree to create separate disjuncts at the outermost level of the formula.

Example 3.1.4

The extraction formula γ_{DOI} conforms to grammar 3.3:

$$\begin{aligned}\gamma_{DOI} =&\Sigma^*\boxed{\texttt{<ee>}}(F\{\boxed{\texttt{https://doi.org/}}C\{(\gamma_{word})^*(\gamma_{word} \vee \boxed{\texttt{/}} \vee \boxed{\texttt{.}})^*\}\} \vee \\ &F\{\boxed{\texttt{doi:}}C\{(\gamma_{word})^*(\gamma_{word} \vee \boxed{\texttt{/}} \vee \boxed{\texttt{.}})^*\}\})\boxed{\texttt{</ee>}}\Sigma^*\end{aligned}$$

where $SVars(\gamma_{DOI}) = \{F, C\}$, F is an exposed variable, and $\gamma_d = [\boxed{\texttt{0}}, \boxed{\texttt{9}}]$, $\gamma_{upper} = [\boxed{\texttt{A}}, \boxed{\texttt{Z}}]$, and $\gamma_{lower} = [\boxed{\texttt{a}}, \boxed{\texttt{z}}]$, $\gamma_{word} = \gamma_{lower} \vee \gamma_{upper} \vee \gamma_d$.

The corresponding normalized form with respect to F is:

$$\begin{aligned}\gamma_{DOI} =&\Sigma^*\boxed{\texttt{<ee>}}F\{\boxed{\texttt{https://doi.org/}}C\{(\gamma_{word})^*(\gamma_{word} \vee \boxed{\texttt{/}} \vee \boxed{\texttt{.}})^*\}\}\boxed{\texttt{</ee>}}\Sigma^* \\ &\vee\ \Sigma^*\boxed{\texttt{<ee>}}F\{\boxed{\texttt{doi:}}C\{(\gamma_{word})^*(\gamma_{word} \vee \boxed{\texttt{/}} \vee \boxed{\texttt{.}})^*\}\}\boxed{\texttt{</ee>}}\Sigma^*\end{aligned}$$

[1]Because the formulas are functional, if a capture variable appears in one disjunct, it must appear in all disjuncts.

Lemma 1 *Given any functional extraction formula g conforming to $G_S(\Sigma, V, v)$, Algorithm 1 normalizes it by producing the list of disjuncts $\Delta(g, v)$ in time that is polynomial in $|g|$, where $|x|$ denotes the length of x.*

Proof. Proof by induction on the height of the expression tree h:
Base Case: $h = 1$: g is $v\{\sigma\}$ or $v\{\epsilon\}$ which is in normal form and returned after executing line 6.
Induction Step: Assume the algorithm generates $\Delta(g, v)$ for all functional formulas conforming to grammar 3.3 with expression tree having height $h < n$. The expression tree T_h of height h for a regular formula g based on the grammar $G_S(\Sigma, V, v)$ can be written as a root and subtrees of height less than h. Each subtree either has no variables, or it has all the capture variables except for v, or (by the induction hypothesis) it can be rewritten in the desired form. Now consider the root of T_h:

1. The root is $\vee$. The tree must include v and because g is functional both subtrees must include v. Therefore, by the induction hypothesis, each subtree can be rewritten in the desired form, so when lines 9 and 10 are executed and the result added to the disjunct list, the list of normalized disjuncts for the whole expression is returned.

2. The root is a $\bullet$. Functionality of g implies that exactly one of the subtrees must have v, which by the induction hypothesis can be written in the desired form, so the other subtree does not have any variable. Therefore, by standard distribution of concatenation over disjunction, we get the desired form $\Delta(g, v)$ by executing either line 14 or line 17 as appropriate.

3. The root is v. Since g is functional, the subtrees cannot include v. Thus, the expression is already in the desired form and line 6 is executed to return the (already) normalized expression.

Complexity Analysis:
The expression tree can be constructed in linear time in the input expression length. To normalize the expression, the expression tree is traversed top-down, which has time complexity that is linear in the input length. For each occurrence of v a new expression is added to the list, which makes the output size $O(m * n)$ where $n = |g|$ and $m < n$ is the number of occurrences of v in g. □

Algorithm 1 to normalize a restricted formula g with respect to exposed variable v.

```
    Input: restricted extraction formula $g$, exposed variable $v$
    Output: list of disjuncts $\Delta(g, v)$
    Precondition: $g$ is functional
 1: $T \leftarrow ExpressionTree(g)$
 2: return NORMALIZE$(T, v)$
 3: procedure NORMALIZE$(T : expressiontree, v : variable)$
    Precondition: $v$ is in $T$
 4:     $\Delta(g, v) \leftarrow list()$
 5:     if $T.root == v$ then
 6:         $\Delta(g, v).add(toRegExp(T))$
 7:     end if
 8:     if $T.root == \vee$ then
 9:         $\Delta(g, v).add($NORMALIZE$(T.left, v))$        ▷ normalize left subtree
10:         $\Delta(g, v).add($NORMALIZE$(T.right, v))$       ▷ normalize right subtree
11:     end if
        ▷ if $v$ occurs in the right/left subtree normalize right/left subtree and concatenate
    every expression in the $\Delta(g, v)$ with the right/left expression
12:     if $T.root == \bullet$ then
13:         if $v$ in $subtree(T.right)$ then
14:             $\Delta(g, v).add($NORMALIZE$(T.right, v))$
15:             $toRegExp(T.left) \bullet \Delta(g, v)$
16:         else
17:             $\Delta(g, v).add($NORMALIZE$(T.left, v))$
18:             $\Delta(g, v) \bullet toRegExp(T.right)$
19:         end if
20:     end if
21:     return $\Delta(g, v)$
22: end procedure
```

3.1.5 Efficient Construction of Extractors

Inputs to our proposed verifier are expressed as extraction formulas conforming to grammar 3.2. We utilize various algebraic operations defined for spanners to statically analyze input programs. Specifically, we convert the inputs to a special automaton representation proposed by Morciano [72], i.e, *extended variable-set automaton* or *eVset-automaton*. EVset-automaton is a variant of variable-set automaton with the same expressivity [37]. By proposing some properties on eVset-automata, namely, *well-behaved*, *pruned*, and *operation-closed*, Morciano [72] designed polynomial algorithms to construct eVset-automata that simulate algebraic operators on eVset-automata. Further, Morciano [72] proves that given an extraction formula, the corresponding eVset-automaton can be created in polynomial time. We denote the eVset-automaton accepting $\emptyset$ by $A_\varnothing$. If E is an eVset-automaton, E_1, and E_2 are union-compatible eVset-automata (i.e., $E_1 = A_\varnothing \lor E_2 = A_\varnothing \lor SVars(E_1) = SVars(E_2)$), and $S \subseteq SVars(E)$, then there are automata $\pi_S(E)$, $E_1 \cup E_2$, and $E \bowtie E_1$ that can be created in polynomial time such that for any document D

- $[\![\pi_S(E)]\!](D) = \pi_S([\![E]\!](D))$,
- $[\![E_1 \cup E_2]\!](D) = [\![E_1]\!](D) \cup [\![E_2]\!](D)$, and
- $[\![E \bowtie E_1]\!](D) = [\![E]\!](D) \bowtie [\![E_1]\!](D)$.

It is trivial to show that renaming variables can also be done in polynomial time, so if E defines a spanner, $x \in SVars(E)$, and $y \notin SVars(E)$ then $\rho_{x\to y}(E)$ defines a spanner such that for any document D, $[\![\rho_{x\to y}(E)]\!] = \rho_{x\to y}([\![E]\!](D))$, that is, the column in $[\![E]\!](D)$ named x is instead named y.

3.1.6 Contextualization of Extraction Formulas

In general, an extraction formula matches a complete document, and certain strings must appear either inside and outside the regions marked by capture variables. If a document is updated, we wish to know whether the extracted content or any of the strings that specify required contextual information is disrupted in any way. To this end, given the specification of a document spanner E, we define a corresponding regular formula $\mathcal{C}_v(E)$ in which all contextual expressions are also marked with capture variables. For any document D, $[\![\mathcal{C}_v(E)]\!]$ produces the same spans for the capture variable v as does $[\![E]\!]$, but it also captures all spans of characters that cannot be replaced arbitrarily by other sub-strings.

Algorithm 2 Algorithm to capture contexts.

```
    Input: extraction formula $E$, $v_k \in SVars(E)$
    Output: modified extraction formula $\mathcal{C}_{v_k}(E)$
    Precondition: $v_k$ is an exposed variable
 1: $D_c \leftarrow emptyList()$
 2: $D \leftarrow \Delta(E, v_k)$                ▷ produce the disjunctive form using Algorithm 1
 3: for all $d \in D$ do
 4:     $T_c \leftarrow$ COVER$(ExpressionTree(d), k)$
 5:     $T_m \leftarrow mergeVars(T_c, k)$                ▷ merge consecutive variables
 6:     $D_c.add(toRegExp(T_m))$
 7: end for
 8: $result \leftarrow \epsilon$
 9:         ▷ ensure that each disjunct has all the variables to produce a functional result
10: for all $d \in D_c$ do
11:     $T_{v_k} \leftarrow findSubtree(d, v_k)$                ▷ find node with $v_k$
12:     for all $y \in getAllVars(D_c)$ and $y \notin getAllVars(d)$ do
                        ▷ change $v\{\cdots\}$ to $v\{y\{\cdots\}\}$ in the current disjunct
13:         $setParent(setParent(T_{v_k}.child, newNode(y)), T_{v_k})$
14:     end for
15:     $result \leftarrow result \vee d$
16: end for
17: return $result$

18: procedure COVER($T : expressiontree, i : int$)
19:     $T_c \leftarrow emptyTree()$
20:     if $T.root == *$ then
21:         if $IsUnigram(T.root.left)$ then
22:             $T_c = T$                ▷ found a $\hat{\Sigma}^*$ so do not cover
23:         else
24:             $T_c \leftarrow setParent(T.root, newNode(v_i))$
25:         end if
26:     end if
27:     if $T.root == \vee$ then
28:         $T_c \leftarrow setParent(T.root, newNode(v_i))$
29:     end if
30:     if $varNode(T.root)$ then
31:         $T_c = T$                ▷ already covered
32:     end if                ▷ cont. in the next page
```

Algorithm 2 Algorithm to capture contexts cont.

```
33:     if $T.root == \bullet$ then
34:         $T.left \leftarrow \textsc{cover}(T.left, i)$
35:         $i' = max(T.left)$        ▷ returns max of $i$ in $T.left$ or zero if there is no $v_i$
36:         if $i \leq i'$ then
37:             $i = i' + 1$
38:         end if
39:         $T.right \leftarrow \textsc{cover}(T.right, i)$
40:         $T_c \leftarrow T$
41:     else
42:         $T_c \leftarrow setParent(T.root, newNode(v_i))$        ▷ a leaf containing a letter
43:     end if
44:     return $T_c$
45: end procedure
```

In this work, we allow only subexpressions with the form of X_i^*, where $L(X_i)$ is a regular language that only has strings of length 1, i.e, *unigrams*, to remain uncovered.

Example 3.1.5

Let Σ be defined as in Example 3.1.2. Σ, γ_d, and $\Sigma - \gamma_d$ are examples of X_i.

In fact, in an extraction formula conforming to grammar 3.2, we retain all sub-expressions representing arbitrary strings in a smaller alphabet, $\hat{\Sigma} \subset \Sigma$, uncovered.

Lemma 2 *Given an extraction formula E and $v \in SVars(E)$, Algorithm 2 returns $\mathcal{C}_v(E)$ in quadratic time.*

Proof. By induction on the height of the expression tree of E it is straightforward to show that Algorithm 2 covers all sub-expressions except for sub-expressions written as $\hat{\Sigma}^*$. $IsUnigram(T.root.left)$ explores the subtree, and if there is no $\bullet$ or $*$, the subtree represents a $\hat{\Sigma}$. Note that since missing variables are added to each disjunct (line 13), the resulting $C_v(E)$ is functional.

Complexity analysis:
Given an input expression E, the expression tree can be constructed in linear time in the

input expression length $O(|E|)$. The normalization has time complexity that is linear in the input length (Lemma 1). $cover()$ and $mergeVars()$ perform a top-down traversal over the expression tree, which has time complexity that is linear in the input length $O(|E|)$. For an input expression E with $m < |E|$ occurrences of v_k, $cover()$ and $mergeVars()$ are called m times, which makes the time complexity of the loop (line 3) $O(|E| * m)$ The time complexity of the loop on line 10 is $O(m)$. Consequently, the time complexity of the algorithm is $O(m * |E|)$, and the output size is $|C_v(E)| = O(m * |E|)$. □

Example 3.1.6

Applying Algorithm 2 to γ_{title} results in $C_{v_0}(\gamma_{title})$:

$$\gamma_{title} = (\Sigma^*\boxed{\texttt{<booktitle>}} \vee \Sigma^*\boxed{\texttt{<title>}})\texttt{T}\{\gamma_{title}\}(\boxed{\texttt{</title>}}\Sigma^* \vee \boxed{\texttt{</booktitle>}}\Sigma^*)$$

where $v_0 = \texttt{T}$, $\gamma_{punct} = \boxed{;} \vee \boxed{,} \vee \boxed{;} \vee \boxed{!} \vee \boxed{.} \vee \boxed{`}$, and $\gamma_{title} = (\gamma_{lower} \vee \text{␣} \vee \gamma_{upper} \vee \gamma_{punct} \vee \boxed{(} \vee \boxed{)} \vee \boxed{-})^*$

$$C_{v_0}(\gamma_{title}) = v_1\{(\Sigma^*\boxed{\texttt{<booktitle>}} \vee \Sigma^*\boxed{\texttt{<title>}})\}\texttt{T}\{\gamma_{title}\}v_2\{(\boxed{\texttt{</title>}}\Sigma^* \vee \boxed{\texttt{</booktitle>}}\Sigma^*)\}$$

To be most effective in identifying contexts, we wish to identify as many instances of $\hat{\Sigma}^*$ as we can, since they indicate portions of the document that are almost irrelevant to the extractor. To this end, we note a few rewrite rules for regular expressions that can be applied even in the presence of capture variables, where R and R_i are any regular expressions with capture variables, R_0 is any regular expression without capture variables, $\sigma \in \Sigma$, and $x\{\}$ specifies a capture variable (Figure 3.3).

Example 3.1.7

Applying Algorithm 2 to the extraction formula in Example 3.1.6 after applying rewrite rules (Figure 3.3) results in:

$$C_{v_0}(rewrite(\gamma_{title})) = \Sigma^* v_1\{(\boxed{\texttt{<booktitle>}} \vee \boxed{\texttt{<title>}})\}\texttt{T}\{\gamma_{title}\}v_2\{(\boxed{\texttt{</title>}} \vee \boxed{\texttt{</booktitle>}})\}\Sigma^*$$

3.1.7 Spanners for Basic Span Relationships

A substantial portion of our work is based on investigating various relationships between spans. Allen has defined a set of 13 possible relationships between non-empty intervals [8].

$$\begin{array}{ll}
R_1 \vee R_2 \to R_2 \vee R_1 & (R_0^*)^* \to R_0^* \\
(R_1 \vee R_2)R_3 \to R_1R_3 \vee R_2R_3 & R_1(R_2 \vee R_3) \to R_1R_2 \vee R_1R_3 \\
R\epsilon \to R & \epsilon R \to R \\
(\hat{\Sigma}^*)^* \to \hat{\Sigma}^* & \hat{\Sigma}^*\hat{\Sigma}^* \to \hat{\Sigma}^* \\
L(R_0) \subseteq L(\hat{\Sigma}^*) \implies \hat{\Sigma}^* \vee R_0 \to \hat{\Sigma}^* & \\
(\hat{\Sigma}^*R_1) \vee (\hat{\Sigma}^*R_2) \to \hat{\Sigma}^*(R_1 \vee R_2) & (R_1\hat{\Sigma}^*) \vee (R_2\hat{\Sigma}^*) \to (R_1 \vee R_2)\hat{\Sigma}^* \\
(\hat{\Sigma}^*R\hat{\Sigma}^*)^* \to (\hat{\Sigma}^*R)^*\hat{\Sigma}^* & \\
\Sigma \vee \sigma \to \Sigma & (\Sigma - \sigma) \vee \sigma \to \Sigma \\
\varnothing R \to \varnothing & R\varnothing \to \varnothing \\
\varnothing \vee R \to R & \\
\varnothing^* \to \varnothing & x\{\varnothing\} \to \varnothing
\end{array}$$

Figure 3.3: Rewrite rules for extraction formulas

These can be extended to capture the same basic relationships among spans (including empty spans) as summarized in Table 3.1. All possible relationships among spans can be described by disjunctions of these basic relationships; for example, "X is disjoint from Y" can be expressed as the disjunction of the first four basic relationships ($\Gamma_{(X<Y)} \vee \Gamma_{(X>Y)} \vee \Gamma_{(X\mathbf{m}Y)} \vee \Gamma_{(X\mathbf{mi}Y)}$), which purposely differs from that of Fagin et al. [37] when X is a nonempty substring at $[i, j\rangle$ and Y is at $[i, i\rangle$[2]. "X is not equal to Y" can be similarly expressed as the disjunction of the first 12 basic relationships.

3.2 Document Update Model

Updates can add documents to or delete documents from the database $\mathbb{D}$, or they can change documents already in $\mathbb{D}$. In this chapter, we concentrate on the latter form of update, where sub-string *replacement*, *deletion*, and *insertion* are basic update operations over documents. A change to the text is typically preceded by some browsing activities

[2]The definition of overlapping spans in [37] is asymmetric for empty spans, i.e., given a span $[i, j\rangle$ the empty span at $[i, i\rangle$ is considered overlapping with $[i, j\rangle$ while the empty span at $[j, j\rangle$ is considered disjoint from $[i, j\rangle$ but we treat both as overlapping, as expressed in $\Gamma_{(X\mathbf{s}Y)}$, $\Gamma_{(X\mathbf{si}Y)}$, $\Gamma_{(X\mathbf{f}Y)}$, and $\Gamma_{(X\mathbf{fi}Y)}$.

1	$\Gamma_{(X<Y)}$	X precedes Y	$\Sigma^* X\{\Sigma^*\}\Sigma^+ Y\{\Sigma^*\}\Sigma^*$
2	$\Gamma_{(Y>X)}$	Y is preceded by X	
3	$\Gamma_{(X\mathbf{m}Y)}$	X meets Y	$\Sigma^* X\{\Sigma^+\} Y\{\Sigma^+\}\Sigma^*$
4	$\Gamma_{(Y\mathbf{mi}X)}$	Y is met by X	
5	$\Gamma_{(X\mathbf{o}Y)}$	X overlaps with Y	$\Sigma^*(X\vdash)\Sigma^+(Y\vdash)\Sigma^+(\dashv X)\Sigma^+(\dashv Y)\Sigma^*$
6	$\Gamma_{(Y\mathbf{oi}X)}$	Y is overlapped by X	
7	$\Gamma_{(X\mathbf{s}Y)}$	X starts Y	$\Sigma^* Y\{X\{\Sigma^*\}\Sigma^+\}\Sigma^*$
8	$\Gamma_{(Y\mathbf{si}X)}$	Y is started by X	
9	$\Gamma_{(X\mathbf{d}Y)}$	X during Y	$\Sigma^* Y\{\Sigma^+ X\{\Sigma^*\}\Sigma^+\}\Sigma^*$
10	$\Gamma_{(Y\mathbf{di}X)}$	Y contains X	
11	$\Gamma_{(X\mathbf{f}Y)}$	X finishes Y	$\Sigma^* Y\{\Sigma^+ X\{\Sigma^*\}\}\Sigma^*$
12	$\Gamma_{(Y\mathbf{fi}X)}$	Y is finished by X	
13	$\Gamma_{(X=Y)}$	X is equal to Y	$\Sigma^* X\{Y\{\Sigma^*\}\}\Sigma^* \vee \Sigma^* X\{\epsilon\}Y\{\epsilon\}\Sigma^*$

Table 3.1: Allen's interval relationships extended to spans. The abuse of notation in the regex for lines 5 and 6 represents a spanner described by an automaton with operators that open and close the variables X and Y as indicated.

or search operations to locate update positions in a target document. In this section we describe the proposed formal model for document update.

Target points of change in a document are specified using patterns over the input string, expressed as a document spanner with one special variable called the update variable. Specifically, an update expression is a regular expression with capture variable defined by $\bar{\gamma}$ in grammar 3.3 or $G_S(\Sigma, V, x)$ where x is called the update variable.

The functional document spanner that is represented by an update formula g maps every document D to a span relation, which we call the *update relation* and denote as $[\![g]\!](D)$. Our model of update is *global*: all documents in the database are matched against the update formula and to generate the associated span relation. When the spanner is used for updating a document D, sub-strings of D associated with the spans in the update relation are simultaneously replaced by new values specified by U.

Definition 3 *An instance of an update specification with an update spanner specified by $g \in L(G_U)$ and replacement value specified by a string $U \in (\,(\Sigma - \texttt{\$}) \cup (\,\texttt{\$(}\,SVars(g)\,\texttt{)}\,)\,)^*$ is called an* update expression *and denoted by $Repl(g, U)$. Given a document D and applying $Repl(g, U)$ to D produces a new document $Repl(g, U)(D)$ that is identical to D except that each sub-string $s_i \in D$ corresponding to a span marked by $UVar(g)$ is replaced by the string U but with every occurrence of a sub-string '$\texttt{\$(}\nu_k\texttt{)}$' in U replaced by the string in s_i corresponding to the span marked by ν_k.*[3]

Note that if U is the empty string, then the update results in the deletion of the sub-strings corresponding to spans marked by $UVar(g)$; otherwise, wherever an empty span $[i, i\rangle$ is marked by $UVar(g)$, the replacement, in effect, inserts a string before the i^{th} character (or at the end of the string if $i = n + 1$).

An update yields a specific functional mapping between the original and updated documents: $ReplSpan(g, U)([i, j\rangle) \rightarrow [i', j'\rangle$ which is a mapping from spans to spans. This is properly defined when $[i, j\rangle$ is disjoint from all spans marked by $UVar(g)$ or when $[i, j\rangle$ is a span that is exactly matched by g as x (Figure 3.4). If $[i, j\rangle$ is disjoint from all spans marked by $UVar(g)$ when updating D, then $D'_{[i',j'\rangle} = D_{[i,j\rangle}$.

[3]This corresponds to the use of named back-references in replacement text, as defined in several programming languages.

Example 3.2.1

The spanner represented by the following extraction formula appends the string `new` to the end of any non-empty input document.

$$\gamma_{append} = \Sigma^* F\{\Sigma\}$$

where F is the update variable and $U = \$(F)$`new`.

Definition 4 *An update spanner specified by g is said to be* well-defined *if the following two properties hold for each document and each pair of accepting runs: (1) if the pair of spans for $UVar(g)$ are not equal, they must not overlap, and (2) if the pair of spans for $UVar(g)$ are equal, then for each variable appearing in U, the pair of spans marked for that variable must be equal.*

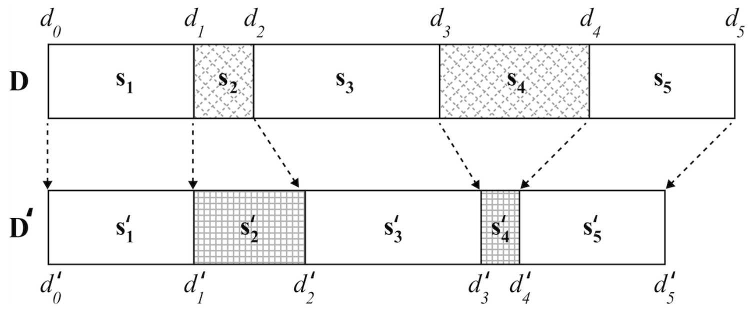

Figure 3.4: A sample document D and its updated peer D'. Filled areas in D are marked by the update variable to be updated. *RepSpan* maps $[d_i, d_{i+1}\rangle$ to $[d'_i, d'_{i+1}\rangle$.

Theorem 1 *Whether or not an update spanner specified by g is well-defined can be verified in polynomial time.*

Proof. Given an update spanner specified by g and symbols $X \notin SVars(g)$ and $Y \notin SVars(g)$, we can create the spanner

$$\mathit{conflicts}(g) = \pi_X(\rho_{UVar(g)\to X}(\llbracket g \rrbracket)) \bowtie \llbracket \Gamma_{(X \circledcirc Y)} \rrbracket \bowtie \pi_Y(\rho_{UVar(g)\to Y}(\llbracket g \rrbracket))$$

where $\Gamma_{(X \circledcirc Y)}$ is the disjunction of the fifth through the twelfth basic relationships in Table 3.1; that is, spans in $\pi_{UVar(g)}([\![g]\!](D))$ could include two distinct spans that cover identical sub-spans. If $\mathit{conflicts}(g) = \varnothing$, then g satisfies the first condition. Morciano [72] has shown that the time complexity to join two operation-closed well-behaved eVset-automata is quadratic in the size of inputs, i.e., the number of states and transitions. Also for both projection and rename operations, we need to scan transitions and rename/remove some variables, which makes the time complexity of both operations linear in the number of transitions.

Similarly, given an update spanner specified by g, symbols $X \notin SVars(g)$ and $Y \notin SVars(g)$, and $Z \in SVars(g) \setminus \{UVar(g)\}$, we can create the spanner

$$\mathit{ambig}(g, Z) = \pi_{\{UVar(g),X\}}(\rho_{Z\to X}([\![g]\!])) \bowtie [\![\Gamma_{(X\neq Y)}]\!] \bowtie \pi_{\{UVar(g),Y\}}(\rho_{Z\to Y}([\![g]\!]))$$

where $\Gamma_{(X\neq Y)}$ is the disjunction of the first 12 basic relationships in Table 3.1; that is, spans in $[\![g]\!](D)$ could include two rows that match on $UVar(g)$ but do not match on Z. If for all $Z \in SVars(g) \setminus UVar(g)$, $\mathit{ambig}(g, Z) = \emptyset$, then g satisfies the second condition. Join, rename, and projection operators have been applied to create $\mathit{ambig}(g, Z)$; thus, the time complexity of checking this property is polynomial in the size of the input. □

Example 3.2.2

It is easy to verify that the spanner represented by the extraction formula in Example 3.1.3 is not well-defined since $\mathit{conflicts}(g) \neq \emptyset$.

Example 3.2.3

The spanner represented by the following extraction formula is not well-defined since $\mathit{ambig}(\gamma'_{date}, Z) \neq \emptyset$ where $Z \in \{Y, M, D\}$:

$$\gamma'_{date} = \Sigma^* \boxed{\texttt{mdate=}}\, U\{Y\{\gamma_d^*\}M\{\gamma_d^*\}D\{\gamma_d^*\}\}\boxed{\texttt{>}}\Sigma^*$$

3.3 Irrelevant and Autonomously Computable Updates

As defined above, applying an update expression $\mathit{Repl}(g, U)$ to an input document D, where g specifies a well-defined update spanner, returns a new document D' in which the contents of each span identified by $[\![g]\!]$ is replaced by the string specified by U. Given an update expression and an extraction spanner, we wish to determine, for all potential input documents, whether the extracted materialized view can be kept consistent with the updated source documents without running the extractor after updating the documents in the database. This problem is similar to filtering out irrelevant updates or applying updates autonomously to relational materialized views [13].

Definition 5 *An update expression $Repl(g, U)$ is* irrelevant *with respect to an extractor $[\![E]\!]$ if for every input document, applying $[\![E]\!]$ to $Repl(g, U)(D)$ produces a span relation that is identical to applying $[\![E]\!]$ to D. That is, if $D' = Repl(g, A)(D)$, then $[\![E]\!](D') = [\![E]\!](D)$.*

Even if an update expression is relevant with respect to an extractor, it may be that the modification to the extracted relation can be computed without re-running the extractor.

Definition 6 *An update expression $Repl(g, U)$ is* autonomously computable *with respect to an extractor $[\![E]\!]$ if for every input document, applying $[\![E]\!]$ to $Repl(g, U)(D)$ can be computed from the update expression, the update relation, the extraction formula that defines the extraction spanner, and the extracted relation.*[4]

Interestingly, even if an update only appends text to a document, it might not be autonomously computable with respect to all extractors: the appended text might include spans that could be extracted, depending on whether the required context is provided by the original document, or it might provide the context needed for matching additional spans in the original text.

There is an important distinction between the problems of updating traditional relational views and updating materialized extractions. Span relations contain pairs of offsets from input documents, not document content. Thus a span relation might be affected by an update even if the replaced text is not within an extracted span. In particular, replacing a string of one length by a string of another length somewhere in the document might cause a span somewhere else in the document to shift, even if the content of that span is unaffected.

More specifically, given a document D and the corresponding updated document D', if span S in D is disjoint from all spans produced by the well-defined update spanner

[4]Autonomous computability for updates is analogous to determinacy [74] for queries.

$[\![g]\!]$, let $\mathit{shift}(g,U)(S)$ represent the corresponding span in D', i.e., the new location of the content of S in D'. $\mathit{shift}(g,U)(S)$ is shifted from S by an amount that is dependent on the lengths of all spans in the update relation that precede S in D and the length of the strings specified by U, as captured by Algorithm 3. For every extracted span, Algorithm 3 simply scans through the update relation to compute the amount of shift applicable to the current span. There is potential for an efficient implementation of this function. For example, the changes to the length of the updated spans need to be computed only once. Furthermore, an update affects the extracted spans with larger offsets, thus sorting might speed up the shift function.

Algorithm 3 Shift Algorithm.

```
    Input: update relation $R_U$, $U$, span $S = [i, j\rangle$
    Output: span $S' = [i', j'\rangle = \mathit{shift}(g, U)(S)$
    Precondition: $R_U$ contains no duplicates and no span that overlaps $S$ or any other
    span in $R_U$
1:  $\mathit{shift}, l \leftarrow 0$
2:  for tuple $\in R_U$ do
3:      $[m, n\rangle \leftarrow \mathit{tuple}[x]$                    ▷ span for the update variable
4:      if m < i then
5:          $l \leftarrow \mathit{computeLength}(U, \mathit{tuple})$
6:          $\mathit{shift} \leftarrow \mathit{shift} + (n - m) - l$
7:      end if
8:  end for
9:  return $[i - \mathit{shift}, j - \mathit{shift}\rangle$
```

Definition 7 *Update expression $Repl(g,U)$ is* shiftable[5] *with respect to an extraction spanner $[\![E]\!]$ if for every input document, applying $[\![E]\!]$ to $Repl(g,U)(D)$ produces a span relation that is identical to applying $[\![E]\!]$ to D except to replace each span S by $\mathit{shift}([\![g]\!](D),U,S)$. That is, if $D' = Repl(g,U)(D)$, then $[\![E]\!](D') = \{S' \mid \exists\, S \in [\![E]\!](D), S' = \mathit{shift}([\![g]\!](D),U,S)\}$.*

Thus, a shiftable update is a special case of an autonomously computable update. By definition, if an update expression is irrelevant with respect to an extraction spanner, then it is also shiftable with respect to that spanner.

In this work, extractors are expressed as a function in the closure of extraction spanners under algebraic operators, i.e., *core spanners*. Next, we show that shiftability of an update w.r.t. an extraction spanner is a property that propagates through algebraic operators.

[5]This was previously named *pseudo-irrelevant* [55, 56].

Theorem 2 *Let $\mathbb{E}_{Repl(g,U)}$ represent the set of extraction spanners for which $Repl(g, U)$ is shiftable. $\mathbb{E}_{Repl(g,U)}$ is closed under union, projection, natural join, and string selection.*

Proof. The proof can be derived according to Lemmas 3, 4, 5, and 6. □

Lemma 3 *If $[\![E_1]\!] \in \mathbb{E}_{Repl(g,U)}$ and $[\![E_2]\!] \in \mathbb{E}_{Repl(g,U)}$ then $[\![E_\cup]\!] \in \mathbb{E}_{Repl(g,U)}$ where $[\![E_\cup]\!] = [\![E_1 \cup E_2]\!]$ and $[\![E_1]\!]$ and $[\![E_2]\!]$ are union compatible .*

Proof. By definition, for every input string D, $[\![E_\cup]\!](D) = [\![E_1]\!](D) \cup [\![E_2]\!](D)$. Suppose there exists an input string D such that $D' = Repl(g, U)(D)$ and $[\![E_\cup]\!](D') \neq \{S' \mid \exists\ S \in [\![E_\cup]\!](D), S' = shift([\![g]\!](D), U, S)\}$. Thus we have $[\![E_1]\!](D') \neq \{S' \mid \exists\ S \in [\![E_1(D]\!]), S' = shift([\![g]\!](D), U, S)\}$ or $[\![E_2]\!](D') \neq \{S' \mid \exists\ S \in [\![E_2]\!](D), S' = shift([\![g]\!](D), U, S)\}$, which is a contradiction. □

Lemma 4 *If $[\![E]\!] \in \mathbb{E}_{Repl(g,U)}$, then $[\![\Pi_Y E]\!] \in \mathbb{E}_{Repl(g,U)}$ where $Y \subseteq SVars(E)$.*

Proof. As $[\![\Pi_Y E]\!](S)$ is obtained from $[\![E]\!](S)$ by removing the columns in $Vars(E) \setminus Y$, so the update stays shiftable for the remaining spans from $[\![E]\!](S)$. □

Lemma 5 *If $[\![E_1]\!] \in \mathbb{E}_{Repl(g,U)}$ and $[\![E_2]\!] \in \mathbb{E}_{Repl(g,U)}$, then $[\![E_\bowtie]\!] \in \mathbb{E}_{Repl(g,U)}$ where $[\![E_\bowtie]\!] = [\![E_1]\!] \bowtie [\![E_2]\!]$.*

Proof. By contradiction, similar to the proof for Lemma 3. □

Lemma 6 *If $[\![E]\!] \in \mathbb{E}_{Repl(g,U)}$ then $[\![\zeta^R_Y E]\!] \in \mathbb{E}_{Repl(g,U)}$ where $Y \subseteq SVars(E)$.*

Proof. By definition, for every input S, string selection selects some rows from the extracted spans $[\![E]\!](S)$. Those rows have been shifted correctly. □

3.4 Categorizing Document Updates

Given an extracted relation, an update relation (spans for strings that will be replaced), and the update expression (from which the amount of shift can be computed), the relation

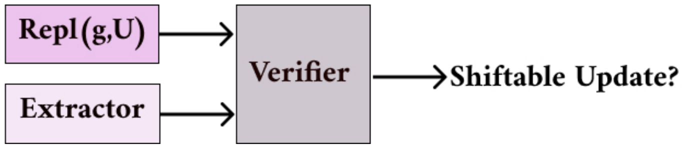

Figure 3.5: The verifier statically analyzes an update expression and an extraction formula to test sufficient conditions for being a shiftable update.

that would be extracted post-update can be computed if it were known that the update expression is shiftable with respect to the extractor. The essence of our approach is to inspect various kinds of *overlap* between an update expression and an extractor to ascertain whether or not the update expression is shiftable *independently of input documents*. The proposed process verifies some *sufficient* conditions for irrelevant, autonomously deletable, and shiftable updates.

If an update changes the content length of an extracted span, then it will be relevant; the extractor should be re-executed.[6] However, even without changing an extracted value, an update could change the context for determining that a span should be extracted. First, updated spans, with new values, could form new matches for the extraction spanner, which would create new rows in the extracted view if we re-run the extractor. Second, some extracted spans might no longer match after the update, and therefore the associated rows would disappear when the extractor is re-run after the update.[7]

After introducing a few required constructs, we present a sound, but not necessarily complete, mechanism to determine whether an update expression, specified by the update formula g and replacement specifier U, is shiftable with respect to a document spanner specified by an extraction formula E (Figure 3.5).

3.4.1 Post-update Spanner

Our proposed approach requires some conditions on documents that are the result of updates. Even if an oracle supplies the update relation and the amount to shift each extracted tuple, we still require knowledge about the specifications of the resulting documents to determine shiftability. We utilize information provided by the update expression to construct

[6]We leave for future work the determination and detection of conditions under which the extracted relation after update might be autonomously computable even when the update causes an extracted region to change length.

[7]These effects are not mutually exclusive.

the post-update spanner which represents updated documents as well as spans of those documents corresponding to the new values.

Definition 8 *An update expression Repl(g, U) is* durable *if: (1) $[\![g]\!]$ is a well-defined update spanner and (2) spans marked by the update variable and spans marked by context variables by its peer spanner $[\![\mathcal{C}_{UVar(g)}(g)]\!]$ are disjoint.*

Proposition 1 *Testing whether a well-defined update spanner is durable can be performed in polynomial time.*

Proof. Given a well-defined update spanner specified by g with update variable x, $\mathcal{C}_x(g)$ can be constructed in polynomial time. Let x_i represent the i^{th} variable marking the context in $\mathcal{C}_x(g)$ and construct a spanner $p_i(g)$:

$$p_i(g) = \pi_x([\![\mathcal{C}_x(g)]\!]) \bowtie [\![\Gamma_{(x \pitchfork x_i)}]\!] \bowtie \pi_{x_i}([\![\mathcal{C}_x(g))]\!]$$

where $\Gamma_{(X \pitchfork Y)}$ is the disjunction of the fifth through thirteenth basic relationships in Table 3.1. $\Gamma_{(X \pitchfork Y)}$ represents the universal spanner that contains spans marked as X and Y where any span that is marked by X has at least one span in common with spans marked by Y. Therefore, $p(g)$ represent all documents that can be updated while spans marked as update variable have at least one span in common with the set of spans marked as context. Therefore, if, for all i where x_i is outside x in g, $p_i(g) = \emptyset$, then $[\![g]\!]$ is durable.

Complexity analysis:
The algebraic operators utilized to construct p_i have been shown to have polynomial time complexity in the size of automaton representing the input. Converting a regular expression with capture variables to an automaton and checking its emptiness is also polynomial [72].
□

For a given specification of an update spanner we derive a new spanner to match documents that result from an update. Some sub-strings in an updated document come from the replacement specifier U and some come from the document before it is updated, either from outside the regions matched by the update variable or from the use of references to other variables inside U.

Given a replacement specifier U and an update spanner specified by g, U implicitly describes a language $L(U)$, but uses back references to variables in g. We define a regular language $\Diamond(U, g)$ that contains $L(U)$ and for which $\Diamond(U, g) \setminus L(U)$ is fairly small.

Proposition 2 *Given a replacement specifier U and a well-defined update spanner represented by g, Algorithm 4 outputs $\Diamond(U, g)$ in polynomial time in the input length.*

Algorithm 4 Algorithm to build $\Diamond(U, g)$

```
    Input: update spanner specification $g$, update specification $U$
    Output: $\Diamond(U, g)$
 1: $T_g \leftarrow ExpressionTree(g)$                          ▷ get tree representation
                          ▷ retrieve all back-references and their positions from $U$
 2: $bkrefs[< bkref, pos >] \leftarrow getBkrefs(U)$
 3: for all $< bkref, pos >$ in $bkrefs$ do
 4:     $subexps[] \leftarrow getSubexp(T_g, bkref)$   ▷ retrieve all sub-strings associated with $bkref$
    ▷ append all sub-string, with a valid disjunction symbol between every two sub-string
 5:     $resultStr \leftarrow append(toString(subexp[]), \text{“} \vee \text{”})$
                                              ▷ enclose by parenthesis and add to the list
 6:     $resultStrs[<, >] \leftarrow < \text{“(”} + resultStr + \text{“)”}, pos >$
 7: end for
 8: $U \leftarrow replace(U, resultStrs[])$          ▷ substitute each bkref with corresponding string
 9: $\Diamond(U, g) \leftarrow toRegExp(U)$                 ▷ convert string $U$ to regular expression
10: return $\Diamond(U, g)$
```

Proof. The proof is by construction. Given $U = u_1 \cdots u_i \texttt{\$(}v_i\texttt{)} u_{i+1} \cdots u_j \texttt{\$(}v_j\texttt{)} u_{j+1} \cdots u_n$ and g, Algorithm 4 scans U and g and substitutes each back-reference variable $\texttt{\$(}v_i\texttt{)}$ of U with the disjunction of all expressions marked with that variable in g: $(exp_1^{v_i} \vee exp_2^{v_i} \vee \cdots)$, omitting other variables enclosed by v_i. The update spanner specified by g is functional, which implies that in every run exactly one of the expressions marked as variable v_m in g is replaced by its associated back-reference in U.[8] The resulting regular expression is:

$$\begin{aligned}\Diamond(U, g) =& u_1 \bullet \cdots \bullet u_i \bullet (exp_1^{v_i} \vee exp_2^{v_i} \vee \cdots) \bullet u_{i+1} \bullet \cdots \bullet u_j \bullet (exp_1^{v_j} \vee exp_2^{v_j} \vee \cdots) \bullet \\ & u_{j+1} \bullet \cdots \bullet u_n = U_i \bullet (exp_1^{v_i} \vee exp_2^{v_i} \vee \cdots) \bullet U_j \bullet (exp_1^{v_j} \vee exp_2^{v_j} \vee \cdots) \bullet \cdots \bullet U_n\end{aligned}$$

where $exp_k^{v_m}$ is a regular expression marked by $v_m \in SVars(g)$.

Complexity Analysis:
To build $\Diamond(U, g)$, Algorithm 4 scans U to retrieve all back-referenced variables. Then the expression tree of g is scanned to substitute back-references in U with associated regular expressions taken from g, which makes the complexity polynomial in $|g|$ and $|U|$.

□

[8]Note that eliminating other capture variables inside v_i to create $exp_k^{v_i}$ does not affect the set of strings marked as v_i.

Definition 9 *Given an update spanner specified by g and a replacement specifier U, a spanner that matches the updated documents and marks all updated spans is called a* post-update spanner *and its specification is denoted by $\nabla(g, U)$.*

Algorithm 5 Algorithm to build $\nabla(g, U)$

```
    **Input:** update expression $Repl(g, U)$, update variable $x$
    **Output:** $\nabla(g, U)$
    **Precondition:** $[\![g]\!]$ is durable, $Repl(g, U)$ respects the alphabets in $\mathcal{C}_x(g)$
1:  $D \leftarrow \Delta(g, x)$                    ▷ get the disjunctive form running Algorithm 1
2:  $\nabla(g, U) \leftarrow \emptyset$
3:  **for all** $g_i \in D$ **do**                  ▷ process each disjunct $g_i$ in $\Delta(g, x)$
4:      $T \leftarrow ExpressionTree(g_i)$
5:      $T_l \leftarrow get\_left(T, x)$          ▷ get sub-expressions on left of update variable
6:      $T_r \leftarrow get\_right(T, x)$         ▷ get sub-expressions on right of update variable
7:      $\nabla(g, U) \leftarrow \nabla(g, U) \vee toRegExp(T_l) \bullet x\{\Diamond(U, g_i)\} \bullet toRegExp(T_r)$
8:  **end for**
9:  **return** $\nabla(g, U)$
```

Definition 10 *Given an extraction formula E with exposed variable v, an update expression $Repl(g, U)$* respects the alphabets *in $\mathcal{C}_v(E)$ if, for every expression of the form $\hat{\Sigma}^*$ not covered in $\mathcal{C}_v(E)$, the update neither deletes nor inserts a symbol in $(\Sigma - \hat{\Sigma})$.*

If $Repl(g, U)$ does not respect the alphabets in $\mathcal{C}_v(E)$, tuples might be inserted into or deleted from an extracted relation $[\![E]\!](D)$, as will be illustrated by Example 3.4.1.

Example 3.4.1

Consider the wh-word extractor specified by

$$\Sigma^* \boxed{\texttt{␣}}\ Q\{\boxed{\texttt{Wh}}\ \gamma^+_{lower}\}\ \boxed{\texttt{␣}}(\Sigma - \boxed{\texttt{.}} - \boxed{\texttt{!}} - \boxed{\texttt{?}})^*\ \boxed{\texttt{?}}\ \Sigma^*$$

If an update replaces semicolons by periods, a wh-question that matched before the update might no longer match the specification, and therefore a tuple might be deleted from the extracted relation. If it instead replaces periods by semicolons, a newly formed wh-question might be created, and thus a tuple might be inserted into the extracted relation. If it replaces “ere” by “y”, “Where” could be replaced by “Why” and thus the extracted relation might change by more than a simple shift.

Lemma 7 *For a durable update expression Repl(g, U) that respects the alphabets in $\mathcal{C}_v(g)$, Algorithm 5 outputs $\nabla(g, U)$ in polynomial time in the input length.*

Proof. The proof is by contradiction. Take an arbitrary string D that has m regions marked by $x = \mathit{UVar}(g)$ when processed by $[\![g]\!]$. Let D' denote the result of updating D by $\mathit{Repl}(g, U)$. Let us assume that there are some *problematic regions* of D that are marked by x but do not have correctly marked corresponding regions when running $[\![\nabla(g, U)]\!]$ on D'. Let s_4 as depicted in Figure 3.4 be the *leftmost* problematic region of D. Because the update spanner is *durable* all sub-expressions in g identified as contexts as well as any instances of $\hat{\Sigma}^*$ that match to the left of s_4 must correctly match the regions preceding s'_4, i.e., s'_1, s'_2, and s'_3. Based on Proposition 2, the region s'_4 must match $\Diamond(g, U)$. Thus $D_{[d_0,d_4\rangle}$ corresponds correctly to $D'_{[d'_0,d'_4\rangle}$. Therefore, there cannot be a any problematic region.

Complexity Analysis:
The time complexity of creating $\Delta(g, x)$ and any expression tree, and to traverse the expression tree to extract right and left context is $O(|g|)$ (lines 1, 4, 5, and 6). Finally, the time complexity of constructing $\Diamond(U, g_i)$ is polynomial for (line 7). Since the loop runs $m \in O(|g|)$ times where m is the number of disjuncts in $\Delta(g, x)$ (line 3), the time complexity of algorithm is also polynomial. $\square$

3.4.2 Detecting Shiftability for Spanners

We first describe two simple special cases:

1. If $E \bowtie g = \emptyset$, the update is irrelevant: there is no document on which both E and g match, and therefore any document that is updated cannot have extracted content.

2. If $E \bowtie g \neq \emptyset$ but $E \bowtie \nabla(g, U) = \emptyset$, there exist documents on which both E and g match, but if such a document is updated, the span relation produced by the extractor becomes empty. Although the update is relevant, it is autonomously computable: every extracted tuple from the updated relation is deleted.

Clearly, if a document update changes some or all of the content of an extracted span or any span specifying a contextual constraint, it will in general change the extracted span relation. Similarly, after an update, the replacement text might cause one or more additional spans to be matched, so that the span relation includes tuples that did not meet the extraction condition before the update. By examining Definition 7, we can deduce three possibilities that could cause an update (changing document D to become D') to fail to be shiftable with respect to an extractor: (1) a span s extracted from D fails to be extracted from D'; (2) a span s' extracted from D' does not correspond to any extracted span from D prior to the update; and (3) a span s extracted from D changes to become s' extracted from D', but it is not a simple shift.

We leave it to future work to determine under what conditions an update that overlaps extracted spans or its contexts happens to be shiftable. Instead, we determine when there can be no overlap, and then under which further conditions an update is shiftable.

Definition 11 *Given extraction formulas E and E', $[\![E]\!]$ is* independent *of $[\![E']\!]$ if for every document D, $[\![\mathcal{C}_v(E)]\!](D)$ where $v \in SVars(E)$, includes no span that overlaps with a span in $[\![E']\!](D)$. Otherwise, we say that $[\![E]\!]$ depends on $[\![E']\!]$.*

Proposition 3 *Given two extraction formulas, dependency of their corresponding spanners can be verified in polynomial time.*

Proof. Given E and E' where $SVars(\mathcal{C}_v(E)) \cap SVars(E') = \varnothing$, $X, Y \notin SVars(\mathcal{C}_v(E)) \cup SVars(E')$, $Z \in SVars(\mathcal{C}_v(E))$, and $Z' \in SVars(E')$ we construct the following spanner:

$$depends(E, E', Z, Z') = \pi_X(\rho_{Z\to X}([\![\mathcal{C}_v(E)]\!])) \bowtie [\![\Gamma_{(X \Xi Y)}]\!] \bowtie \pi_Y(\rho_{Z'\to Y}(E')$$

where $\Gamma_{(X\Xi Y)}$ is the disjunction of the fifth through thirteenth basic relationships in Table 3.1. $\Gamma_{(X\,\Xi\,Y)}$ represents the universal spanner that contains spans marked as X and Y where any span that is marked by X has at least one span in common with spans marked by Y. That is, $depends(E, E', Z, Z') \neq \varnothing$ if there exists at least one span in $\pi_Z([\![\mathcal{C}_v(E)]\!])$ which overlaps or equals at least one span in $\pi_{Z'}([\![E']\!])$. Therefore, if for all $Z \in SVars(\mathcal{C}_v(E))$ and $Z' \in SVars(E')$ $depends(E, E', Z, Z') = \varnothing$, E is independent of E'.
Complexity Analysis:
Given E with m occurrences of v and E', the cost of constructing $C_v(E)$ is $O(m * |E|)$(Lemma 2). All the algebraic operators used in $depends(E, E', Z, Z')$ have polynomial time complexity in the input size ([72]). The number of variables in $C_v(E)$ is $k < |E|$ and in E' is k' which makes the whole cost polynomial. $\square$

Theorem 3 *A durable update expression $Repl(g, U)$ is shiftable with respect to an extraction spanner represented by E if $[\![E]\!]$ is independent of $[\![g]\!]$ and $[\![\nabla(g, U)]\!]$ and $Repl(g, U)$ respects the alphabets in $\mathcal{C}_v(E)$.*

Proof. Take an arbitrary document D which has m regions marked by $[\![g]\!]$ to be updated, such as s_2 and s_4 in Figure 3.4. Let D' denote D's updated peer which has $m' \geq m$ regions marked by $[\![\nabla(g, U)]\!]$, among which m regions are the result of an update (like s_2' or s_4' in Figure 3.4). Assume that some spans of D are marked by $[\![E]\!]$ to be extracted. To prove that an update is shiftable w.r.t. an extractor we must prove three properties:

- For every extracted region in D there is a corresponding region in D' that will be extracted, i.e., no span disappears from the span relation after the update. Assume by contradiction that there exist extracted spans that disappear after the update, i.e., their corresponding regions are not extracted from D'. There are two possible reasons that a span does not match after an update:
  - the subexpression associated with the extracted span, does not match after getting updated. Since $[\![E]\!]$ is independent of $[\![g]\!]$ the extracted regions in D occur at spans between regions marked by $[\![g]\!]$, like s_1 and s_3 and s_5, which have not changed. This case therefore is not possible.
  - at least one of the other associated sub-expressions does not match after the update. Since the extractor spanner is independent of the update spanner, regions marked by the update variable, s_2 and s_4, must match against only those sub-expressions of E that are identified as $\hat{\Sigma}^*$, where $\hat{\Sigma} \subseteq \Sigma$. However, since $Repl(g, U)$ respects the alphabets in $\mathcal{C}_v(E)$, any update expression matching $\hat{\Sigma}^*$ includes a replacement specification U that also matches $\hat{\Sigma}^*$. The unmodified regions match either local contexts or $\hat{\Sigma}^*$ which will match after the update. Therefore, this case is not possible either.
- For every region extracted from D' there is a corresponding region extracted from D, i.e., no new span appears in the span relation due to update. Assume by contradiction that there are some spans that appear after the update in the span relation. The possible reasons that these spans are not extracted before the update can be:
  - the sub-expression associated with the extracted span did not match before the update but matched after the update. This implies that the extracted span must overlap the region marked by $[\![\nabla(g, U)]\!]$ like s_2' or s_4'. This is not possible due to independence between $[\![E]\!]$ and $[\![\nabla(g, U)]\!]$. Therefore these new spans can only reside in regions between the regions marked by $[\![\nabla(g, U)]\!]$.
  - at least one of the other associated sub-expressions did not match before but matches after the update. Due to independence, the region marked by $[\![\nabla(g, U)]\!]$ can only match the sub-expression identified as $\hat{\Sigma}^*$ where $\hat{\Sigma} \subseteq \Sigma$. This implies

that local contexts match regions marked by $[\![\nabla(g, U)]\!]$ but not regions marked by $[\![g]\!]$. This is not possible since $Repl(g, U)$ respects the alphabets in $\mathcal{C}_v(E)$, and any update expression matching $\hat{\Sigma}^*$ includes a replacement specification U that also matches $\hat{\Sigma}^*$. Thus every span found after the update must have a corresponding span found before the update.

- For extracted spans *RepSpan* is realized as a *shift* function. Because of independence, the length of the extracted regions remains the same after the update, since no update can occur inside extracted regions. But other regions preceding an extracted region that are identified as $\hat{\Sigma}^*$ can be updated, which makes the starting offset of an extracted region to move back/forth. Since the length of the extracted region is fixed, the end offset moves by the same amount, which is indeed a shift of the extracted region.

□

Proposition 4 *Shiftability of a durable update spanner can be verified in polynomial time in the input size.*

Proof. It is trivial to see that given a durable update expression $Repl(g, U)$ and an extraction formula E Algorithm 6 verifies shiftability of g in polynomial time. □

In summary, the proposed verifier performs three tests on the update expression and two tests on the extractor, considering the update and post-update expressions (Figure 3.6). After those tests, rows in the extracted table may need to be shifted if the update is shiftable, and otherwise (at least some) re-extraction is required. As a result, T_{ext}, the time to re-extract, is $O(T_e * |R|)$ and T_{diff}, the time to test for shiftability and to perform the shifts when an extraction is found to be shiftable, is $O(T_v + T_s * |R|)$, where R is the extracted relation, T_e is the time to extract one row, T_s is the time to apply the shift for one row, and T_v is the time to verify shiftability which is polynomial in the input size. In the next section, we shall see that $T_{ext} \gg T_{diff}$ in practice.

Algorithm 6 Verify Shiftability of Update Spanner

```
    **Input:** extraction program $P$, update expression $\mathit{Repl}(g, U)$, update variable $x$
    **Output:** Boolean
    **Precondition:** $[\![g]\!]$ durable, $\mathit{Repl}(g, U)$ respects the alphabets in $\mathcal{C}_x(g)$
 1: $R \leftarrow \emptyset$
 2: for all $E_i \in P$ do                                    ▷ verify all regular formulas $E_i$ in $P$
 3:     for all $Z \in SVars(\mathcal{C}(E_i))$ do
 4:         $R \leftarrow \mathit{depends}(E_i, g, Z, x) \cup \mathit{depends}(E_i, \nabla(g, U), Z, x)$
 5:         if $R \neq \emptyset$ then
 6:             return false
 7:         end if
 8:     end for
                                     ▷ test for conflicting symbols (i.e., respecting alphabets)
 9:     for all uncovered $\hat{\Sigma}^*$ in $E_i$ do
10:         $A \leftarrow \{\sigma \mid \sigma \in x$ and $\sigma$ outside all capture variables back-referenced in $U\}$
    $\cup\{\sigma \mid \sigma \in U\}$
11:         if $A \cap (\Sigma \setminus \hat{\Sigma}) \neq \varnothing$ then
12:             return false                                 ▷ $\sigma$ might be removed or added
13:         end if
14:     end for
15: end for
16: return true
```

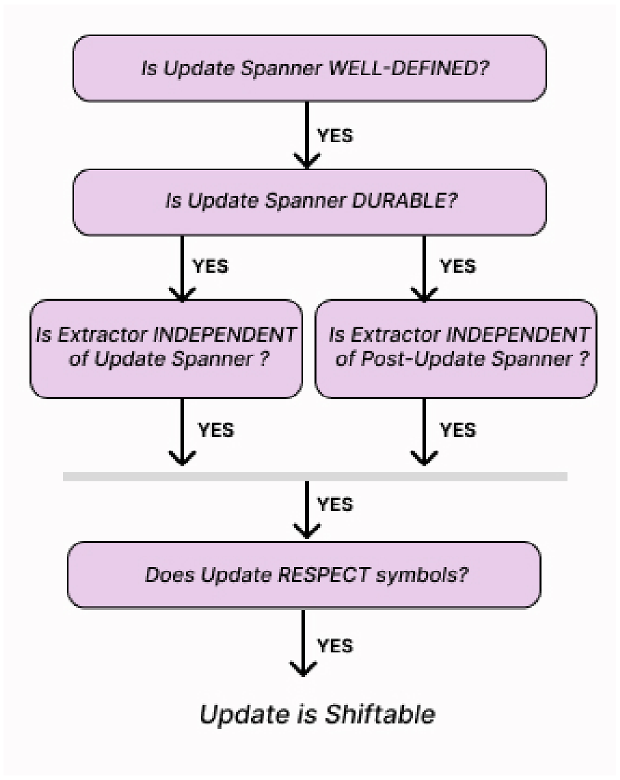
Is Update Spanner WELL-DEFINED?
YES
Is Update Spanner DURABLE?
YES
YES
Is Extractor INDEPENDENT of Update Spanner ?
Is Extractor INDEPENDENT of Post-Update Spanner ?
YES
YES
YES
Does Update RESPECT symbols?
YES
Update is Shiftable

Figure 3.6: The proposed verification process for maintaining extracted views is realized through five distinct tests.

3.5 Practicality of Detecting Shiftable Updates

We have designed experiments with two goals in mind:

1. All the proposed algorithms for extracted view maintenance such as normalization, contextualization, and property verification have polynomial time complexity. However, we are interested in showing that our optimization strategy is beneficial in realistic situations dealing with practical extraction scenarios from real-world large datasets.

2. We have identified sufficient conditions for shiftability of document updates expressed as regular formulas with respect to extractors expressed as core AQL queries. We wish to demonstrate practical update cases that can be expressed using our specification.

In addition, we point out some cases for which the proposed sufficient conditions are not met despite updates being shiftable.

3.5.1 Verification System

We have developed a system in Scala that verifies an extraction program expressed as a core AQL query. If the program passes the test, the extracted view content is updated by running a shift algorithm. If it does not pass the test the extractor needs to be executed from scratch. Through experimentation, we show the run-time overhead imposed by our verifier in practice. Also, we compare the run-time of differential maintenance of the extracted views and re-executing the extraction program. Our system is developed on top of the engine proposed and implemented by Marciano [72][9]. Therefore, in this section we first review the main functionalities offered by that engine, and second we describe how we exploit each functionality in the experiments. Marciano's engine operates in two modes:

- **compilation mode:** given a core AQL query, various polynomial construction algorithms are used to construct a well-behaved eVset-automaton that simulates the input AQL query. As shiftability of an update with regard to an extractor in closed under algebraic operators (Theorem 2) our verifier needs to test only regex formulas in a AQL query. However, the verifier operates on two different spanner representations: 1) regular expressions with capture variable for contextualizing the regex formula, constructing post-update spanners, and respect for alphabet test 2) well-behaved eVset-automata to test the durability and independence properties. The verifier uses the compiler to construct the required automaton for the latter purpose.

- **evaluation mode:** In this mode, the engine evaluates a well-behaved eVset-automaton on an input string and produces the corresponding span relation. Morciano [72] shows that in the absence of optimization techniques, compiling the AQL into its equivalent eVset-automaton and matching against documents can be faster than first matching extraction formulas and then applying algebraic operator on the span relations, similar to the approach taken in SystemT. For experiments, wherever we need to evaluate a spanner against an input string, we use Morciano's engine.

To assess the feasibility of our verifier in practice, we would ideally like to report formal metrics (such as precision, recall, and F-measure) for the number of the updates that are

[9]The runtime system designed by Morciano is available at: https://github.com/ae-mo/master-thesis/tree/master/project_sbt/arc

determined to be shiftable and to evaluate the performance of our system. Unfortunately, no benchmark of information extractors with associated updates is available to perform such experiments. We therefore describe two data sets with realistic extractors and updates against which we can evaluate our system experimentally.

3.5.2 DataSets

We use two datasets in our experiments, namely DBLP [1] and Blog Authorship [85]. We have designed specific extractors for each dataset. DBLP extractors exploit structure in the data by being sensitive to XML tags. However, Blog authorship extractors merely use the unstructured content of documents.

DBLP

DBLP is an open bibliographic dataset containing information about major computer science journals and proceedings. DBLP is an evolving semi-structured dataset with frequent updates to add new records, edit existing records, and delete records [1]. In DBLP, bibliographic records are stored as a single large XML file DBLP.xml with a fairly simple schema [62]. Since, our hypothetical system, similar to SystemT (Figure 1.3) follows the document-at-a-time processing paradigm, we split DBLP.xml to create a realistic document database. For this corpus, we utilized APIs prepared by the DBLP team to process DBLP.xml [3]. We create a distinct file for each author and include all bibliographic records for that author, one per line (i.e., what users of DBLP <https://dblp.org/> see when looking up an author, but exporting XML for each record). We have downloaded the latest available dataset at the time of running our experiments [2]. The size of this snapshot of DBLP.xml is 3.5 GB. The size of the resulting document database is 7.9 GB with $3,091,270$ documents representing $20,021,301$ publications, Each publication can take various forms, including articles, inproceedings, proceedings, books, incollection, PhD theses, and Master's theses. Note that each publication is counted as many times as the number of its authors.

Blog Authorship Corpus

The Blog Authorship Corpus [85] is a collection of unstructured blogs used by Morciano to assess his *RunTime* system. This corpus has 19,320 documents, one per blogger (Figure 3.8). There are 681,288 posts in the corpus, which consumes 8.1 MB in total.

```xml
<record>
<person key="homepages/a/JFAllen" mdate="2022-03-30">
<author>James F. Allen</author>
<note type="affiliation">University of Rochester, New York, USA</note>
<url>http://www.cs.rochester.edu/~james/</url>
...
</person>
...
<article key="journals/cacm/Allen83" mdate="2011-06-07">
<author>James F. Allen</author>
<title>Maintaining Knowledge about Temporal Intervals.</title>
<pages>832-843</pages>
<year>1983</year>
<volume>26</volume>
<journal>Commun. ACM</journal>
<number>11</number>
<url>db/journals/cacm/cacm26.html#Allen83</url>
<ee>http://doi.acm.org/10.1145/182.358434</ee>
</article>
...
<inproceedings key="conf/um/BlaylockA05" mdate="2017-05-21">
<author>Nate Blaylock</author>
<author>James F. Allen</author>
<title>Generating Artificial Corpora for Plan Recognition.</title>
<pages>179-188</pages>
<year>2005</year>
<crossref>conf/um/2005</crossref>
<booktitle>User Modeling</booktitle>
<ee>https://doi.org/10.1007/11527886_24</ee>
<url>db/conf/um/um2005.html#BlaylockA05</url>
</inproceedings>
...
 </record>
```

Figure 3.7: A sample document that contains bibliographic information for James F. Allen.

```
<Blog>
...
</post>
<date>03,August,2004</date>
<post>
Hey all. Just felt that i needed a change, so i have deleted all previous posts.
they were boring, anyway. My grandmother is in the hospital again. she's been ...
don't know exactly what's wrong with her. so much for modern technology.
Read the Da Vinci Code by Dan Brown. interesting book. it argues that Jesus was
not divine, and that the church invented his divinity. Also argues that Jesus was
married to Mary Magdalene. well i'm not going to go in depth to argue anything,
but i think Dan Brown made lots of mistakes in his assumptions. for example,
he writes that Peter was the one in the picture " The Last Supper " who was
making threatening gestures. apparently that man is Judas Iscariot, not Simon
Peter. that explains a lot about the hand gestures. Anyways, there's a reason why
the book is under the fiction section in book shops and libraries. Thinking of
getting a new cpu. Thinking of you, too. Bye all.
</post>
...
</Blog>
```

Figure 3.8: A sample document that contains blog posts for a blogger.

3.5.3 Experiment Platform

The verifier, shift algorithm, and extraction system are all single-threaded programs in Scala 2.11.2 together with Java SE 11. All experiments are performed on a AMD EPYC 7502P 32-Core Processor under Ubuntu 20.04.1 LTS (Focal Fossa). The source code and all performance details of the experiments can be found at the project's Git repository.[10]

3.5.4 Extractors

We have designed several realistic extractors (Tables 3.2 and 3.3) to show the applicability of our optimization strategy in practice. For the Blog corpus, we adapted the extractors defined by Morciano. Each of his primitive extractors uses a specially designed operator $X \bowtie_d Y$ to match $\bigcup_{i \leq d}(X\Sigma^i Y)$. Because the use of such operators will cause our verifier to

[10]https://github.com/Besatkassaie/Differential-Maintenance-Engine

deem all updates to be relevant (the replacement text may cause the string to exceed the maximum length), we replace them with $X(\Sigma - \boxed{\texttt{?}} - \boxed{\texttt{.}})^*Y$ to require instead that matches to X and Y appear in the same sentence.

Table 3.2: Primitive extractors.

Primitive extractors from Blog corpus	
$Q_1 = \llbracket\Sigma^*\boxed{\textvisiblespace}A\{\gamma_{Action}\}\gamma_b\gamma_s^*T\{\gamma_{film}\}\Sigma^*\rrbracket$	e.g., *Saw* ... *"ET"* in one sentence
$Q_2 = \llbracket\Sigma^*\boxed{\textvisiblespace}A\{\gamma_{Attribute}\}\gamma_b\gamma_s^*\boxed{\textvisiblespace}T\{\gamma_{Movie}\}\gamma_b\Sigma^*\rrbracket$	e.g., *worst* ... *flick* in one sentence
$Q_{RN} = \llbracket\Sigma^*\boxed{\textvisiblespace}A\{\gamma_{Role}\}\gamma_b\gamma_s^*T\{\gamma_{name}\}\Sigma^*\rrbracket$	e.g., *actor* ... *Brad Pitt* in one sentence
$Q_{10} = \llbracket\Sigma^*\boxed{\textvisiblespace}A\{\gamma_{Action}\}\gamma_b\gamma_s^*T\{\gamma_{film}\}\gamma_s^*\boxed{\textvisiblespace}B\{\gamma_{Attribute}\}\gamma_b\Sigma^*\rrbracket$	e.g., *Saw* ... *"ET"* ... *clever* in one sentence
$Q_{12} = \llbracket\Sigma^*\boxed{\textvisiblespace}A\{\gamma_{Sentiment}\}\gamma_b\gamma_s^*\boxed{\textvisiblespace}T\{\gamma_{Genre}\}\gamma_b\gamma_s^*\boxed{\textvisiblespace}B\{\gamma_{Movie}\}\gamma_b\Sigma^*\rrbracket$	e.g., *like* ... *crime* ... *film* in one sentence
Primitive extractors from DBLP corpus	
$S_{Jrn} = \llbracket\Sigma^*\boxed{\texttt{<title>}}T\{\gamma_{title}\}\boxed{\texttt{</title><pages>}}P\{(\gamma_d)^*\boxed{\texttt{-}}(\gamma_d)^*\}\boxed{\texttt{</pages><year>}}$ $Y\{\gamma_d\gamma_d\gamma_d\gamma_d\}\boxed{\texttt{</year><volume>}}V\{\gamma_d^*\}\boxed{\texttt{</volume><journal>}}J\{\gamma_{title}\}\boxed{\texttt{</journal>}}\Sigma^*\rrbracket$	article title, pages, year, volume, and journal from journal articles
$S_{Mod} = \llbracket\Sigma^*\boxed{\texttt{mdate="2015}}(\Sigma - \boxed{\texttt{\\n}})^*\boxed{\texttt{<title>}}T\{\gamma_{title}\}\boxed{\texttt{</title>}}\Sigma^*\rrbracket$	titles of publications last modified in 2015
$S_{AA+} = \llbracket\Sigma^*\boxed{\texttt{"><author>}}A\{\gamma_{author}\}\boxed{\texttt{</author><author>}}\Sigma^*\rrbracket$	first authors of pubs with 2+ authors
$S_{2010} = \llbracket\Sigma^*\boxed{\texttt{<title>}}T\{\gamma_{title}\}\boxed{\texttt{</title>}}(\Sigma - \boxed{\texttt{\\n}})^*\boxed{\texttt{<year>2010</year>}}\Sigma^*\rrbracket$	titles of publications from 2010
$S_{VLDB} = \llbracket\Sigma^*\boxed{\texttt{"><author>}}A\{\gamma_{author}\}\boxed{\texttt{<}}(\Sigma - \boxed{\texttt{\\n}})^*$ $\boxed{\texttt{<booktitle>}}B\{\gamma_{title}\boxed{\texttt{VLDB}}\gamma_{title}\}\boxed{\texttt{</booktitle>}}\Sigma^*\rrbracket$	first authors of publications from VLDB

Table 3.3: Compound extractors.

Compound extractors from Blog corpus	
$Q_5 = Q_1 \cup Q_2 \cup Q_{RN}$	union of binary extractors
$Q_{13} = Q_{10} \cup Q_{12}$	union of ternary extractors
Compound extractors from DBLP corpus	
$S_{VAA+} = S_{VLDB} \bowtie S_{AA+}$	multi-author VLDB articles
$S_{J2010} = \pi_J(S_{Jrn} \bowtie S_{2010})$	journal titles from 2010
$S_{2010\Delta} = S_{2010} \cup S_{Mod}$	titles from 2010 or updated in 2015

For readability, the primitive extractors are built on top of some simple grammar building blocks.
character sets:

$$\gamma_u = [\boxed{\texttt{A}}, \boxed{\texttt{Z}}] \qquad \gamma_l = [\boxed{\texttt{a}}, \boxed{\texttt{z}}]$$
$$\gamma_d = [\boxed{\texttt{0}}, \boxed{\texttt{9}}] \qquad \gamma_b = \boxed{\textvisiblespace} \vee \boxed{\texttt{,}}$$
$$\gamma_w = (\gamma_u \vee \gamma_l \vee \gamma_d) \qquad \gamma_s = (\Sigma - \boxed{\texttt{?}} - \boxed{\texttt{.}})$$
$$\gamma_p = (\boxed{\texttt{:}} \vee \boxed{\texttt{,}} \vee \boxed{\texttt{;}} \vee \boxed{\texttt{!}} \vee \boxed{\texttt{.}} \vee \boxed{\texttt{?}}) \qquad \gamma_t = (\gamma_w \vee \gamma_p \vee \boxed{\texttt{-}})$$

basic patterns:

$$
\begin{aligned}
\gamma_{name} =& \gamma_u \gamma_l^* \boxed{\texttt{␣}} \gamma_u \gamma_l^* (\boxed{\texttt{␣}} \vee \gamma_p) \\
\gamma_{film} =& ((\boxed{\texttt{"}}\gamma_w) \vee (\boxed{\texttt{'}}(\gamma_u \vee \gamma_d)))\ \ (\gamma_t^* \vee \gamma_t^*\boxed{\texttt{␣}}\gamma_t^* \vee \gamma_t^*\boxed{\texttt{␣}}\gamma_t^*\boxed{\texttt{␣}}\gamma_t^* \\
& \vee \gamma_t^*\boxed{\texttt{␣}}\gamma_t^*\boxed{\texttt{␣}}\gamma_t^*\boxed{\texttt{␣}}\gamma_t^* \vee \gamma_t^*\boxed{\texttt{␣}}\gamma_t^*\boxed{\texttt{␣}}\gamma_t^*\boxed{\texttt{␣}}\gamma_t^*\boxed{\texttt{␣}}\gamma_t^*)(\boxed{\texttt{'}} \vee \boxed{\texttt{"}})
\end{aligned}
$$

Morciano's extractors use several small word lists (for which we use names starting with upper case letters), including:

$$
\begin{aligned}
\gamma_{Action} &= (\boxed{\texttt{S}} \vee \boxed{\texttt{s}})\boxed{\texttt{aw}} \vee (\boxed{\texttt{W}} \vee \boxed{\texttt{w}})\boxed{\texttt{atch}} \vee (\boxed{\texttt{R}} \vee \boxed{\texttt{r}})\boxed{\texttt{ent}} \\
\gamma_{Aspect} &= \boxed{\texttt{visual}} \vee \boxed{\texttt{plot}} \vee \boxed{\texttt{script}} \vee \boxed{\texttt{dialogue}} \vee \boxed{\texttt{acting}} \vee \boxed{\texttt{actor}} \vee \boxed{\texttt{cast}} \vee \boxed{\texttt{special}} \\
& \vee \boxed{\texttt{effect}} \vee \boxed{\texttt{shot}} \vee \boxed{\texttt{scene}} \vee \boxed{\texttt{sequence}} \\
\gamma_{Attribute} &= \boxed{\texttt{funny}} \vee \boxed{\texttt{better}} \vee \boxed{\texttt{worst}} \vee \boxed{\texttt{worse}} \vee \boxed{\texttt{awful}} \vee \boxed{\texttt{boring}} \vee \boxed{\texttt{entertaining}} \vee \boxed{\texttt{inspiring}} \\
& \vee \boxed{\texttt{clever}} \vee \boxed{\texttt{interesting}} \vee \boxed{\texttt{smart}} \vee \boxed{\texttt{cool}} \vee \boxed{\texttt{dope}} \vee \boxed{\texttt{quirky}} \vee \boxed{\texttt{hilarious}} \\
& \vee \boxed{\texttt{amazing}} \vee \boxed{\texttt{well-done}} \vee \boxed{\texttt{rushed}} \\
\gamma_{Genre} &= \boxed{\texttt{action}} \vee \boxed{\texttt{adventure}} \vee \boxed{\texttt{children}} \vee \boxed{\texttt{family}} \vee \boxed{\texttt{comedy}} \vee \boxed{\texttt{crime}} \vee \boxed{\texttt{documentary}} \vee \boxed{\texttt{drama}} \\
& \vee \boxed{\texttt{fantasy}} \vee \boxed{\texttt{noir}} \vee \boxed{\texttt{horror}} \vee \boxed{\texttt{musical}} \vee \boxed{\texttt{mystery}} \vee \boxed{\texttt{romance}} \vee \boxed{\texttt{sci-fi}} \vee \boxed{\texttt{science␣fiction}} \\
& \vee \boxed{\texttt{thriller}} \vee \boxed{\texttt{war}} \vee \boxed{\texttt{western}} \vee \boxed{\texttt{gangster}} \vee \boxed{\texttt{epic}} \vee \boxed{\texttt{historical}} \\
\gamma_{Movie} &= \boxed{\texttt{feature}} \vee \boxed{\texttt{dvd}} \vee \boxed{\texttt{film}} \vee \boxed{\texttt{movie}} \vee \boxed{\texttt{flick}} \\
\gamma_{PlotClue} &= \boxed{\texttt{plot}} \vee \boxed{\texttt{about}} \vee \boxed{\texttt{tell}} \vee \boxed{\texttt{story}} \vee \boxed{\texttt{begins}} \vee \boxed{\texttt{ends}} \vee \boxed{\texttt{finally}} \vee \boxed{\texttt{final}} \vee \boxed{\texttt{beginning}} \vee \boxed{\texttt{middle}} \\
\gamma_{Role} &= \boxed{\texttt{protagonist}} \vee \boxed{\texttt{character}} \vee \boxed{\texttt{director}} \vee \boxed{\texttt{actor}} \vee \boxed{\texttt{role}} \vee \boxed{\texttt{characters}} \vee \boxed{\texttt{critics}} \\
\gamma_{Sentiment} &= \boxed{\texttt{love}} \vee \boxed{\texttt{like}} \vee \boxed{\texttt{hate}} \vee \boxed{\texttt{enjoy}} \vee \boxed{\texttt{cringe}} \vee \boxed{\texttt{cry}} \vee \boxed{\texttt{cried}} \vee \boxed{\texttt{recommend}} \vee \boxed{\texttt{laugh}}
\end{aligned}
$$

To avoid having spurious matches to these words (such as matching $\boxed{\texttt{fractions}}$ when intending to match $\boxed{\texttt{action}}$, or $\boxed{\texttt{aware}}$ when intending to match $\boxed{\texttt{war}}$), our extractors include additional context to require that any match to a word from a word list must be preceded by a blank ($\boxed{\texttt{␣}}$) and followed by either a blank or a comma (γ_b). His compound extractors fall into two classes: Q5 - Q9 are unions of binary joins and Q13 - Q16 are unions of ternary joins. For our experiments, we have chosen the first extractor from each class.

For the DBLP corpus we designed five primitive extractors and three compound extractors (also shown in Tables 3.2 and 3.3) that complement those for the other corpus, and are built on basic patterns for authors' names and titles:

$$
\begin{aligned}
\gamma_{author} =& \gamma_u(\gamma_u \ \vee \ \gamma_l \ \vee \ \boxed{\texttt{-}} \ \vee \ \boxed{\texttt{␣}} \ \vee \ \boxed{\texttt{.␣}})^* \\
\gamma_{title} =& (\gamma_t \ \vee \ \boxed{\texttt{␣}})^*
\end{aligned}
$$

The compound extractors select records on multiple conditions, using spans in common to enforce conjunctive conditions.

3.5.5 Updates

In this section we present a few realistic update scenarios and discuss various aspects of them. In the following sections, for an update spanner denoted as γ_S, the update variable is denoted by F and the corresponding replacement specifier is denoted by U_S.

Classifying Hashtags

A blog might include hashtags such as #photooftheday and #naturephotography, and there may be interest in annotating a subset of these as belonging to some class, such as follows, where $\gamma_h = \gamma_l \vee \gamma_d$ (any lower case letter or digit):

$$\begin{aligned}
\gamma_{hashtag} =& \Sigma^* F\{\boxed{\texttt{\#}}\gamma_h^*(\boxed{\texttt{photo}} \vee \boxed{\texttt{selfie}})\gamma_h^*\}\gamma_b\Sigma^* \\
U_{hashtag} =& \$(F)\boxed{\texttt{\textvisiblespace(photography)}}
\end{aligned}$$

The verifier confirms that this update is shiftable for all our extractors for both corpora.

Enabling URLs

URLs appear in both corpora under consideration, and these can be updated to become clickable links:

$$\begin{aligned}
\gamma_{URL} =& \Sigma^* F\{\boxed{\texttt{http://}}T\{(\Sigma - \boxed{\texttt{<}} - \boxed{\texttt{>}} - \boxed{\texttt{\textvisiblespace}} - \boxed{\texttt{"}} - \boxed{\texttt{'}})^*\}\} \\
& (\boxed{\texttt{<}} \vee \boxed{\texttt{>}} \vee \boxed{\texttt{\textvisiblespace}} \vee \boxed{\texttt{"}} \vee \boxed{\texttt{'}})\Sigma^* \\
U_{URL} =& \boxed{\texttt{<a\textvisiblespace href="}}\$(F)\boxed{\texttt{">}}\$(T)\boxed{\texttt{<a/>}}
\end{aligned}$$

This time, the verifier reports that the update is shiftable with respect to Q_1, Q_2, Q_{10}, and Q_{12} (and therefore also Q_{13}) and also that it is shiftable with respect to all the DBLP extractors. However, the verifier reports that the update is not shiftable with respect to Q_{RN} (and therefore Q_5). The problem is that an updated span F can overlap the extraction of a name marked by T, thus, for example, changing $\boxed{\texttt{\textvisiblespace actor\textvisiblespace...\textvisiblespace http://Brad\textvisiblespace Pitt\textvisiblespace...}}$ to $\boxed{\texttt{\textvisiblespace actor\textvisiblespace...\textvisiblespace<a\textvisiblespace href="http://Brad">Brad</a>\textvisiblespace Pitt\textvisiblespace...}}$. After executing this update, Q_{RN} and Q_5 would need to be re-extracted to accommodate such possible instances.

Reversing Month and Day in Dates

The DBLP corpus includes a modification date on each record, and it might become necessary to convert from yyyy-mm-dd to yyyy-dd-mm, as follows:

$$\begin{aligned}\gamma_{date} =& \Sigma^* \boxed{\texttt{mdate="}} \gamma_d\gamma_d\gamma_d\gamma_d \boxed{\texttt{-}} F\{M\{\gamma_d\gamma_d\}\boxed{\texttt{-}}D\{\gamma_d\gamma_d\}\}\Sigma^* \\ U_{date} =& \$(D)\boxed{\texttt{-}}\$(M)\end{aligned}$$

This update is reported to be shiftable with respect to all the DBLP extractors. When tested against the primitive Blog extractors, it is found to be shiftable with respect to all but Q_1 and Q_{10}, where the string $\boxed{\texttt{mdate=}}$ could immediately precede a film title beginning $\boxed{\texttt{"2015-01-03\textvisiblespace..."}}$. This update, however, demonstrates an aspect of our verifier that could be improved: although the update could apply inside regions that are extracted by Q_1 and Q_{10}, there would be no change to the lengths of the extracted regions, and thus no reason to re-extract (nor even to shift). In fact, this is an unrecognized irrelevant update with respect to those two extractors.

Changing DOIs

The DBLP corpus also includes references to DOIs in a field containing a link to an external electronic resource, using two different formats. An update to convert one format to the other is as follows:

$$\begin{aligned}\gamma_{DOI} =& \Sigma^*(\boxed{\texttt{<ee>}} \vee \boxed{\texttt{<ee\textvisiblespace type="}}\gamma_l\gamma_l\boxed{\texttt{">}})F\{\boxed{\texttt{https://doi.org/}}\}\Sigma^* \\ U_{DOI} =& \boxed{\texttt{doi:}}\end{aligned}$$

Again, the update is found to be shiftable with respect to all the DBLP extractors. However, because $\boxed{\texttt{.}} \in F$, it is not shiftable with respect to any of the Blog extractors (it does not respect their alphabets); the removal of the period might cause any of the extractors to find additional matches. (The same is true when the update is reversed: the verifier detects $\boxed{\texttt{.}} \in U$, which might result in fewer matches after the update.)

3.5.6 Run-time Performance

We use the DBLP corpus to show that the run-time overhead imposed by our verifier is acceptable in practice and that the run-time is much less than that of re-executing an extraction program, even if performed incrementally. First we tried to run the extractors

Table 3.4: Extraction Statistics ($\times$1000) for DBLP corpus.

Spanner	No. of Extracted Documents	No. of Extracted Records
S_{Jrn}	1,422.1$\pm$6.713	10,887.48$\pm$164.507
S_{Mod}	17.820$\pm$1.17	47.480$\pm$3.872
S_{2010}	302.38$\pm$5.045	1,264.36$\pm$32.876
S_{AA+}	2,304.22$\pm$3.891	23,011.44$\pm$ 419.184
S_{VLDB}	1.860$\pm$0.405	5.360$\pm$0.862
S_{VAA+}	1.680$\pm$0.347	4.920$\pm$0.729
S_{J2010}	132.36$\pm$2.409	380.08$\pm$9.480
$S_{2010\Delta}$	incomplete	incomplete

Table 3.5: Extraction times ($\times$1000 sec) for DBLP corpus.

Spanner	Extraction Time	Spanner	Extraction Time
S_{Jrn}	122.1$\pm$5.7	S_{VLDB}	11.0$\pm$0.1
S_{Mod}	8.4$\pm$0.7	S_{VAA+}	10.2$\pm$0.2
S_{2010}	26.3$\pm$3.6	S_{J2010}	22.1$\pm$0.8
S_{AA+}	156.4$\pm$26.1	$S_{2010\Delta}$	$>$450

to completion on the whole DBLP dataset, but most extractors took an excessive amount of time and did not finish within a four day window. Therefore, we report estimates of related statistics over random samples. We took five simple random samples with sample rate of 1%. Note that standard deviations are reported with 95% confidence.

Table 3.4 summarizes the number of documents with extracted content as well as the number of extracted records in thousands from DBLP corpus. This table confirms that we have extractors with different selectivities. Table 3.5 shows how many thousands of seconds each extractor requires, ranging from 2.3 hours to 43.4 hours and with the final extractor requiring more than 125 hours. The minimum time saved for S_{Jrn} and S_{Mod} when documents are updated by γ_{DOI} is 32.2 hours and 2.0 hours, respectively. Undoubtedly, various optimization techniques adopted by mature products such as SystemT can accelerate the extraction process, and these were not implemented on our research-based system. We note, however, that Chen et al. [22] report that the fastest extractor they tested on a Wikipedia corpus of 35 MB requires 100 seconds (using different hardware and software), which implies an extraction time of more than 6 hours when scaled to the size of DBLP. Elsewhere, Shen et al. [86] report a complex, but optimized extractor taking 61 minutes for a corpus smaller than 2% of the size of DBLP! Thus, we are convinced that our extraction times are indicative of the times needed for extractors more generally.

Table 3.6 shows how many seconds are required for verifying shiftability for each update, displaying the minimum, average, and maximum times over all five primitive DBLP extractors. The table also shows the minimum, average, and maximum times required for shifting the tables extracted for the first seven extractors. It is far faster to verify shiftability than to re-execute the corresponding extractor: verification times are at most 3.5 minutes per primitive extractor *independently of the database size*, and simply shifting extracted relations is far faster than re-extracting them, especially when the extractions are highly selective.

Table 3.6: Verification and shift times (sec) for all updates.

Update	Verification			Shift		
	min'm	avg	max'm	min'm	avg	max'm
Hashtags	20	64.0	127	0.2±0.4	10.2	39.2±76.8
URLs	36	108.4	212	1.1±0.1	67.1	239.3±1.8
Dates	1	33.6	64	1.5±0.2	78.7	273.5±6.5
DOIs	18	58.2	110	1.2±0.2	72.4	252.4±7.6

3.5.7 Role of Incremental Updates

Chen et al. [22] have proposed re-using extracted data when it has clearly not been affected by an update. The approach requires three steps: (1) determining which pieces of documents have been updated, (2) re-executing the extractor on portions of the updated documents where changes to the extracted data *or to its context* (specified as a window around each extracted region) might change the results of extraction, and (3) copying over previous extractions guaranteed to be unaffected by the update, but shifting their offsets so as to be able to determine overlaps with the next update.

We note first that incremental re-extraction is orthogonal to determining shiftability. An update might be provably shiftable with respect to $[\![E]\!]$ even if it changes data very near each extracted region, such as is true for γ_{date} with respect to S_{Mod}. On the other hand, if an update is not provably shiftable, such as γ_{URL} with respect to Q_{RN}, incremental extraction can be applied rather than naively re-executing the extractor on the complete document.

To determine the cost of using the approach proposed by Chen et al., we measure the time to execute a simple matcher for the DBLP corpus on the same hardware used in our other experiments. In particular, by running Unix's `diff` against 20% of the DBLP corpus we find that the program requires between 27 ms and 68 ms (mean = 49 ms) per file. This implies that the first step alone requires over 42 hours to execute. Even if the computation is distributed across 32 processors running in parallel, the execution time is excessive for updates that can be determined to be shiftable.

3.5.8 Potential of Parallelism

The previous section presents execution statistics, including extraction, verification, and shift times, all on a single-threaded machine. We have noted that incremental updates can be parallelized. In practice, document-at-a-time extraction, update, and shift can also easily be parallelized, so a system with 32 processors could reduce overall elapsed time in Table 3.5 to 12 minutes or so. The most costly operation in an extraction process is pattern matching, as highlighted by Shen et al. [86], which limits the effectiveness of multithreading for processing a single document. Therefore, as the number of documents increases we need to add more processors. However, there is a threshold at which scaling this approach becomes cost-prohibitive in terms of hardware or cloud service expenses.

Furthermore, the verification algorithm consists of five distinct tests (Figure 3.6), each of which can be executed independently in parallel. As a result, the verification time depends on the most time-consuming test. In the experiments conducted for the DBLP extractors, the tests of independence between extractors and update spanners and post-update spanners are equally the most time consuming tests.

The update and extracted relations should reside in the same server to eliminate the data transfer cost. Unfortunately, distributing the update relation over several servers will slow down the shift algorithm because the amount to shift a span depends on all updates to the document appearing before that span.

3.5.9 Additional Updates

Expansion

γ_{Expd} replaces `FOCS` appearing as book title or journal name with the string `Foundations of Computer Science`.

$$\gamma_{Expd} = \Sigma^* \boxed{\texttt{<booktitle>}} U\{\boxed{\texttt{FOCS}}\}\boxed{\texttt{</booktitle>}} \Sigma^* \ \vee\ \Sigma^* \boxed{\texttt{<journal>}} U\{\boxed{\texttt{FOCS}}\}\boxed{\texttt{</journal>}} \Sigma^*$$

$$U_{\gamma_{Expd}} = \boxed{\texttt{Foundations\textvisiblespace of\textvisiblespace Computer\textvisiblespace Science}}$$

DBLP Extractors: The verifier finds that $[\![S_{Jrn}]\!]$ depends on $[\![\gamma_{Expd}]\!]$ since the update modifies the extracted span marked by J. In fact the length of the extracted region is modified by γ_{Expd} and the shift function cannot compute the updated extracted span autonomously. The verifier stops once it finds a conflict, but if the verifier continued it would find dependency between $[\![S_{Jrn}]\!]$ and $[\![\nabla(\gamma_{Expd}, U_{\gamma_{Expd}})]\!]$ too. The verifier finds $[\![\gamma_{Expd}]\!]$ to be shiftable for $[\![S_{Mod}]\!]$, $[\![S_{AA+}]\!]$, $[\![S_{2010}]\!]$, and $[\![S_{VLDB}]\!]$, consequently for $[\![S_{VAA+}]\!]$ and $[\![S_{2010\Delta}]\!]$.
Blog Extractors: The verifier finds that $[\![\gamma_{Expd}]\!]$ is shiftable w.r.t $[\![Q_1]\!]$, $[\![Q_2]\!]$, $[\![Q_{RN}]\!]$, $[\![Q_{10}]\!]$, and $[\![Q_{12}]\!]$, consequently for $[\![Q_5]\!]$ and $[\![S_{13}]\!]$.

Rectify Space

This update rectifies a formatting issue for space character.

$$\begin{aligned}\gamma_{space} &= \Sigma^* U\{\boxed{\texttt{\&nbsp;}}\}\Sigma^* \\ U_{\gamma_{space}} &= \boxed{\texttt{\textvisiblespace}}\end{aligned}$$

DBLP Extractors: γ_{space} is determined to be not shiftable for any of the extractors. Since $\boxed{\texttt{\textvisiblespace}}$ is a character showing up in γ_{title} and γ_{author} all the primitive extractors depends on $[\![\nabla(\gamma_{space}, U_{\gamma_{space}})]\!]$ and a new row might appear after rectifying the space character.
Blog Extractors: γ_{space} is determined to be not shiftable for any of the extractors. Since $\boxed{\texttt{\textvisiblespace}}$ is a character showing up in the context preceding variables like A, T, and B also it appears in γ_{name}, γ_{film}, and γ_{Genre} therefore there is dependency between the extractors and $[\![\nabla(\gamma_{space}, U_{\gamma_{space}})]\!]$ which might cause a new row appear after the update.

Money

$$\begin{aligned}\gamma_{money} &= \Sigma^*\boxed{\texttt{\$}}(\gamma_d \vee \boxed{\texttt{.}})^* F\{\boxed{\texttt{\textvisiblespace}}\boxed{\texttt{billion}}\}\Sigma^* \\ U_{\gamma_{money}} &= \boxed{\texttt{B}}\end{aligned}$$

DBLP Extractors: γ_{money} is determined to be shiftable w.r.t all extractors in this category.
Blog Extractors: γ_{money} is determined to be shiftable w.r.t $[\![Q_1]\!]$, $[\![Q_2]\!]$, $[\![Q_{10}]\!]$, and $[\![Q_{12}]\!]$, and consequently $[\![Q_{13}]\!]$. γ_{money} is not shiftable w.r.t $[\![Q_{RN}]\!]$ since $[\![Q_{RN}]\!]$ depends on $[\![\nabla(\gamma_{money}, U_{\gamma_{money}})]\!]$: after update, $\boxed{\texttt{B}}$ can form a string in γ_{name}.

Time Format

$$\begin{aligned}\gamma_{timeFormat} &= \Sigma^* \boxed{\textvisiblespace} F\{T\{(\gamma_d \vee \gamma_d\gamma_d)\}D\{(\boxed{\textvisiblespace}(\boxed{\texttt{am}} \vee \boxed{\texttt{pm}})) \vee \boxed{\texttt{am}} \vee \boxed{\texttt{pm}}\}E\{\boxed{\textvisiblespace} \vee \gamma_p\}\}\Sigma^* \\ U_{\gamma_{timeFormat}} &= \$(T)\boxed{\texttt{:00}}\$(D)\$(E)\end{aligned}$$

The verifier finds $\gamma_{timeFormat}$ to be not durable. Consider the input string $\cdots\boxed{\textvisiblespace\texttt{14}\textvisiblespace\texttt{pm}\textvisiblespace}\cdots$ $[\![C_{UVar(g)}\gamma_{timeFormat}]\!]$ in one match marks $\boxed{\texttt{14}\textvisiblespace\texttt{pm}\textvisiblespace}$ as F and in another match marks the second $\boxed{\textvisiblespace}$ as context. For a none-durable update our construction process for the post-update spanner fails.

3.6 Additional Related Work

3.6.1 Document Spanners

Document spanners, defined in Section 3.1, have been proposed to model extractors defined by the core of SystemT [37]. Researchers have addressed many problems using the document spanner model, including how to deal with documents with missing information [68] and how to eliminate inconsistencies from extracted relations [36]. Others have studied the complexity of evaluating spanners and computing the results of various algebraic operations over span relations [9, 38, 78, 79].

Freydenberger and Thompson [39] have investigated the complexity of incrementally re-evaluating spanners in the presence of updates. However, their update model assumes that a document is encoded as a fixed-length *word structure* in which (essentially) there is a special character that represents ϵ and the only operation is replacing one character from $\Sigma \cup \{\epsilon\}$ by another.

3.6.2 Regular Expression Matching

Thompson [92] has shown how to evaluate an automaton on a string. He keeps track of the states that are reached by consuming each character and discarding the path that is explored to reach the states. Therefore, for an input string s and a regular expression r, the time complexity of Thompson's approach is $O(|r| * |s|)$. To evaluate an evset, instead of keeping only a set of states on every offset of the input, Morciano keeps a set of configurations. Each configuration contains the state that is reached, the history of variable operations, whether they have been opened/ closed as well as the unused variables to reach the state, and the current offset on the input for the corresponding configuration. Thus, Morciano's time complexity is $O(|A|^2 * |s|^{2*|V|})$ for an eVset with $|A|$ states and $|V|$ variables.

We are interested in knowing how variables are assigned to sub-strings for each accepting path. This is similar to the notion of *regular expression parsing* that determines how a string matches a regular expression, i.e., parsing. The latest work on this problem presented by Bille and Gørtz [11] shows that a parsing can be found in $O(|s| * |r|)$ with space complexity of $O(|s| + |r|)$. The proposed method finds only one parsing, but it might be feasible to design an efficient method that works for our application, i.e., finding all possible parsing.

Chapter 4

Updatable Extracted Views

In this chapter, our interest is in determining how changes to entries in an extracted table are reflected back to source documents. More specifically, given a collection of documents and an information extraction program, we wish to determine under what conditions changing a single word or phrase p to p' in a document will cause exactly the expected entry in the extracted table to change from p to p', leaving the rest of the table unchanged.

We were motivated to answer this question by the problem of applying privacy transformations to documents. Consider, for example, the problem of maintaining privacy for personal information contained in a collection of electronic health records when publishing research results derived from those records. It has been shown that simply avoiding the publication of identifiers does not protect the privacy of individuals: in the presence of other publicly available data, anonymization is vulnerable to linkage attacks [89]. The solution to this problem has been to apply *differential privacy* [35], which has been studied quite extensively in the context of relational tables. We are interested in the variant of differential privacy in which the data owner applies a randomized algorithm to map records in a table T to records in T', which has the same schema as T [53]. Thereafter, T can be removed from the system, leaving researchers with full access to the synthesized table T', a *synthetic* derivative of the input table (Figure 4.1). To preserve the table's utility, the synthetic values are created in such a way that a set of specified analyses on T', such as histogram or counting queries, produce outcomes that are indistinguishable from those that would be obtained on T.

Our approach to protecting documents is to apply differential privacy to the table(s) obtained from a document collection through information extraction. We assume that each document D (e.g., an individual's health record) in the collection produces some rows

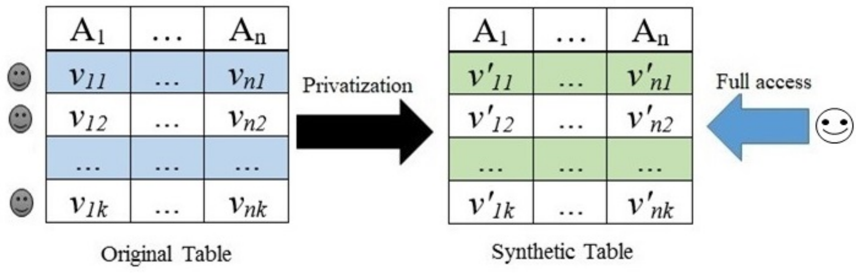

Figure 4.1: Creating and accessing a private table.

in the extracted table T and then apply a differential privacy algorithm to form T', thus protecting each individual document in the collection (as in Figure 4.1). The modified tables can be published and analyzed by untrusted parties without fearing the loss of privacy for individuals whose data is stored in the document collection. For example, to protect writers' identity, Weggenmann et al. [96] replace extracted terms from the training dataset with their synonym provided by WordNet [71] and run the authorship attribution task over updated values.

We further assume that the researchers who are preparing tables for publication have no wish to violate individuals' privacy, but may do so unintentionally; they are assumed to be "semi-trusted." Furthermore, those researchers wish to be able to read the documents that correspond to rows in extracted tables so as to be able to interpret and validate those tables, and they are not experts on differential privacy. We, therefore, wish to present to those researchers a set of documents that *would have* produced the modified table had the same information extraction procedure been applied to them.

Henceforth, we are not concerned with the particulars of differential privacy, but we merely assume that entries in the extracted table are to be replaced by new values. In fact, changing a value in an extracted table might not be for privacy reasons at all, but simply as part of a data cleaning process [48] (for example, the collection of extracted records might disclose some inconsistencies among the source documents). Thus, we are interested in detemining whether extracted views are *updatable*. Specifically, if p is to be replaced in the table by another value p' chosen from some domain of possible values, we wish to replace p by p' in the corresponding location of the document that produced that entry. Under what conditions does this change to the document produce exactly the expected change to the table with no other table entries affected? We say that a given extraction algorithm is *stable* if it is resilient to such modifications of text for *all* possible input document collections, *all* entries in the extracted table, and *all* values in each entry's domain.

In this work, we formalize stable extraction algorithms. Given an extraction program, we then need to verify whether it satisfies those properties required for stability; that is, whether the extractor is stable. Although the ideas cultivated in this thesis are general and independent of the extraction mechanism, we again focus on rule-based information extraction systems to realize the ideas.

In this chapter, we give an overview of a significant part of GATE and its underlying language JAPE, called *Core JAPE*. Then, we propose an approach for creating an equivalent spanner representation for an extraction program written in Core JAPE. Next, we present a mechanism to determine whether an extracted view expressed as a Core JAPE program is updatable. We statically analyze the spanner representation of the extraction program to verify sufficient conditions for being an updatable view.

4.1 Overview of GATE and JAPE

GATE [28] is a commonly used rule-based information extraction system that identifies critical pieces of a document using a grammar written in JAPE [29]. JAPE evolved from the **C**ommon **P**attern **S**pecification **L**anguage (CPSL) [10], a rule-based information extraction language based on regular expressions. GATE extractors include the following stages:

1. First a tokenizer and gazetteer work directly on the input characters. The tokenizer constructs a linear annotation graph in which the edges correspond to sequential spans of text (e.g., words). However the output of the gazatteer might be a nonlinear annotation graph, i.e., a span of text can be annotated by more than one gazatteer annotation, and two different gazatteer annotations can start and end at the same position at text. We describe these two components in detail in Sections 4.1.2 and 4.1.3. A JAPE program then describes how to traverse and modify the annotation graph.

2. Running a JAPE program involves executing a set of matching *rules*, written as regular expressions over annotations that label edges in a rooted directed acyclic graph. Rules are organized into a sequence of *phases* that run consecutively in the order that they appear in the program. In each phase, JAPE interprets the rules in that phase to traverse the annotation graph (from its unique source), identifying possible matches. Each phase might modify the annotation graph by adding and/or removing annotations. This is explained in further detail in Section 4.1.1.

3. In general, the output of a JAPE program in GATE is highly customizable and can be tailored to the specific requirements of the application at hand. For the purpose of our analyses, we design a special phase called the *relation generator*, which is always the final phase to be executed after all phases. It is an ordinary JAPE phase and composed of JAPE rules, except that on the right hand side of every rule there is a call to a JAVA function which forms and outputs a record, as described in Section 4.1.4.

4.1.1 Core JAPE

We describe Core JAPE, the essence of JAPE covered in this work. The Backus-Naur Form (BNF) description of Core JAPE is presented in Figure 4.2.[1] Unless otherwise stated, throughout the remainder of this monograph, we use JAPE and Core JAPE interchangeably. A JAPE program is organized as a sequence of consecutive phases. Each phase accepts an annotation graph as input, modifies it, and outputs the modified graph. A simple JAPE program with four phases is adopted from ANNIE [28], a ready-to-use IE bundle that is implemented and distributed with GATE, and shown in Figure 4.3.[2]

Given an input text, three factors of a phase specify whether a rule fires: the rule's constraints, the phase's policy, and available annotations. In this section, we explain the constituents of a phase including: user-provided JAPE rules, Input (available annotations), and Policy.

JAPE Rules: Each JAPE rule is composed of a *pattern* and an *action*, as in Equation 4.1.1. The pattern appears to the left of the separator $-->$ and describes a regular expression over *annotation constraints* (or simply *constraints*). These regular expressions are converted to finite state transducers, which consume the input and assign spans of text to *binding variables*. We assume that the match for each binding variable is unique in the scope of the left-hand sides of rules.

$$pattern --> \quad action \tag{4.1}$$

A constraint can be as simple as presence of an annotation in the annotation graph. For example, three instances of such simple constraints exist in *Rule:Url1* of Figure 4.3b

[1]For a comprehensive description about additional capabilities of JAPE read the documentation [28].

[2]The corresponding copyright is: Copyright (c) 1998-2004, The University of Sheffield. This file is part of GATE (see http:gate.ac.uk), and is software, licenced under the GNU Library General Public License, Version 2, June 1991 (in the distribution as file licence.html, and also available at http:gate.ac.ukgatelicence.html). Diana Maynard, 19 April 2001 $Id: url_pre.JAPE 5921 2004-07-21 17:00:37Z akshay

$$
\begin{aligned}
\langle\text{SinglePhaseTransducer}\rangle &\models \boxed{\texttt{phase:}}\langle\text{ident}\rangle\\
&\quad (\boxed{\texttt{input:}}(\langle\text{ident}\rangle)*)?\\
&\quad (\boxed{\texttt{Options:control=}}\ \langle\text{ident}\rangle)?\\
&\quad (\langle\text{Rule}\rangle)*\\
\langle\text{Rule}\rangle &\models \boxed{\texttt{Rule:}}\langle\text{ident}\rangle\\
&\quad (\boxed{\texttt{priority=}}\langle\text{integer}\rangle)?\\
&\quad \langle\text{LHS}\rangle\boxed{\texttt{- ->}}\langle\text{RHS}\rangle\\
\langle\text{LHS}\rangle &\models \langle\text{ConstraintGroup}\rangle\\
\langle\text{ConstraintGroup}\rangle &\models (\langle\text{PatternElement}\rangle)+(\boxed{\texttt{|}}(\langle\text{PatternElement}\rangle)+)*\\
\langle\text{PatternElement}\rangle &\models \langle\text{BasicPatternElement}\rangle \mid \langle\text{ComplexPatternElement}\rangle\\
\langle\text{BasicPatternElement}\rangle &\models \boxed{\texttt{\{}}\langle\text{Constraint}\rangle(\boxed{\texttt{,}}\langle\text{Constraint}\rangle)*\boxed{\texttt{\}}}\\
\langle\text{ComplexPatternElement}\rangle &\models \boxed{\texttt{(}}\langle\text{ConstraintGroup}\rangle\boxed{\texttt{)}}(\langle\text{integer}\rangle)?\\
&\quad (\boxed{\texttt{:}}\langle\text{ident}\rangle)?\\
\langle\text{Constraint}\rangle &\models \langle\text{ident}\rangle(\boxed{\texttt{.}}\langle\text{ident}\rangle\boxed{\texttt{=}}\langle\text{AttrVal}\rangle)?\\
\langle\text{AttrVal}\rangle &\models \langle\text{string}\rangle\\
\langle\text{RHS}\rangle &\models \langle\text{Action}\rangle(\boxed{\texttt{,}}\langle\text{Action}\rangle)*\\
\langle\text{Action}\rangle &\models \langle\text{AssignmentExpression}\rangle\\
\langle\text{AssignmentExpression}\rangle &\models \boxed{\texttt{:}}\langle\text{ident}\rangle\boxed{\texttt{.}}\langle\text{ident}\rangle\boxed{\texttt{=}}\boxed{\texttt{\{}}\\
&\quad (\langle\text{ident}\rangle\boxed{\texttt{=}}\langle\text{AttrVal}\rangle(\boxed{\texttt{,}})?)*\boxed{\texttt{\}}}
\end{aligned}
$$

Figure 4.2: BNF description of Core JAPE.

```
Phase: UrlPre
Input: Token SpaceToken
Options: control = appelt
Rule: Urlpre
((({Token.string=="http"} | {Token.string=="ftp"})
  {Token.string==":"}{Token.string=="/"}{Token.string=="/"}
  )|({Token.string=="www"}{Token.string=="."})):urlpre -->
           :urlpre.UrlPre = {rule = "UrlPre"}
```

(a) UrlPre phase defined in ANNIE's url_pre.JAPE

Figure 4.3: A simple JAPE program with four phases, continued below.

including: *{Token}*, *{UrlPre}*, and *{SpaceToken}*. A constraint might instead specify the value of static attributes like *{Token.kind==punctuation}* (Figure 4.3b) or dynamic attributes like *{Token.string=="ftp"}* (Figure 4.3a).

Core JAPE provides two operators for constraints: i) *string equality* operator, such as *{Token.string=="ftp"}* to match a *Token* annotation whose string attribute's value is `ftp`; ii) *regular expressions* operator, such as *{Token.string ==~ "[Ff]tp"}* to match a *Token* annotation whose string attribute's value is either `ftp` or `Ftp`.

According to the grammar (Figure 4.2), simple constraints can be combined in a *ConstraintGroup*. A *ConstraintGroup* may include the disjunction and sequencing of *PatternElements*, each of which specifies one or more constraints.

A *BasicPatternElement* is interpreted as a constraint (or a conjunction of constraints) which have to be satisfied at a particular point in the annotation graph. We only allow multi-constraint statements in which all constraints are associated with *exactly one annotation* and zero or more of its attributes.

Example 4.1.1

The following rules have single and multiple constraints respectively [3]:

```
Rule: Unknown
Priority: 50
({Token.category == NNP}):unknown -->
       :unknown.Unknown = {kind = "PN",rule = Unknown}
```

[3]taken from https://gate.ac.uk/sale/tao/splitch8.html

```
Phase: Url
Input: Lookup SpaceToken Token UrlPre
Options: control = appelt
Rule: Url1
Priority: 50
({UrlPre}({Token})[1,7]):urlAddress ({SpaceToken}) -->
      :urlAddress.Url = {kind="urlAddress",rule="Url1"}
Rule: Url2
Priority: 100
({UrlPre} ({Token})[1,7]{Token.string=="."}
 {Lookup.majorType==country_code} ):urlAddress -->
       :urlAddress.Url = {kind="urlAddress",rule="Url2"}
```

(b) Url phase defined in ANNIE's url.JAPE

```
Phase: Names
Input: Token
Options: control = appelt
Rule: capitalized
(({Token.orth==upperInitial})[1,10]):mark -->
      :mark.Capitalized={rule="capitalized"}
```

(c) Spans of capitalized words

```
Phase: NameAndURL
Input: Capitalized Url
Options: control = first
Rule: final
({Capitalized}):name ({Url}):page -->
      :name.Name={rule="final"},:page.Page={rule="final"}
```

(d) Collocations of names and URLs

Figure 4.3: A simple JAPE program with four phases, cont'd.

```
Rule: Surname
({Lookup.majorType == "name", Lookup.minorType == "surname"}):surname
      -->:surname.Surname = {}
```

The following rule has constraints on two different annotations *Lookup* and *Token*:

```
Rule: Surname
({Lookup.majorType == "name", Lookup.minorType == "surname", Token}):surname
      -->:surname.Surname = {}
```

A *PatternElement* may be repeated a finite number of times, and it may be bound to a *binding variable*. Binding labels make the corresponding part of the annotation graph available on the action part of a rule (appearing after $--\ >$), for further processing. Consider, for example, the phase in Figure 4.3c. If the annotation graph is as depicted in Figure 4.4, the binding variable *mark* makes the span associated to three *Token*s available on the right-hand side to get annotated as *Capitalized*.

For simplicity, the only legitimate action permitted is to include instructions to associate annotations and possibly some attributes with the spans of text that are marked by binding labels and add them to the annotation graph. We require that every binding variable be associated with *exactly* one annotation and its corresponding attributes. We assume that every rule once fired modifies the annotation graph: at least one binding variable exists for each possible firing of a rule.

Definition 12 *An assignment expression corresponding to an annotation's attribute is called* an attribute expression*.*

Similar to programming languages, we distinguish two kinds of attribute expressions: *static* ones whose value can be determined at compile time and do not change during runtime, and *dynamic* ones whose value depends on the input string.

Attribute expressions for each annotation represent a partial function from a set of attributes to a set of values. For example, for rule URL1 in Figure 4.3b, the attribute expression is realized by a function with two mappings: *kind="urlAddress"* and *rule="Url1"*; These attribute expressions are static. Possible dynamic attribute expressions are *string-="example"* or *length=7* which are determined from processing the input text.

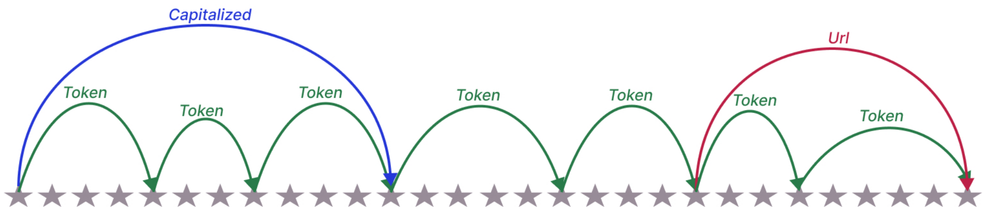

Figure 4.4: A hypothetical annotation graph to show effects of Input in a phase. Stars in gray are proxies of input characters.

In addition to rules, two declarations for a phase are essential to our work: *Input* and *Policy*. Input enumerates the annotation types that are available to the rules in a phase: only those edges in the annotation graph that are labelled by input types are visible to the rules in each phase. Thus, for example, two annotations are deemed to be adjacent if they cover two spans of text with no annotation *in the Input list* covering a span that starts between them. Consider, for example, the phase in Figure 4.3d. If the annotation graph's state is as depicted in Figure 4.4, according to the phases' input, *Rule:final* matches the graph and fires. However, if *Token* was also listed as input, the rule would not fire for this annotation graph because *Capitalized* would no longer appear to be adjacent to *Url*.

A policy specifies the order in which spans of text are considered and where scanning resumes after a rule is fired. Thus the policy determines the strategy to be taken to pick a match when more than one span can be matched and when matches might overlap. JAPE offers five policies: *Once*, *First*, *Appelt*, *Brill*, and *All*.

With the Once and First policies, spans are tested from the leftmost shortest to longest; whenever a match for a rule is found, the rule matching that span is fired and the corresponding span is marked with the relevant annotation. Thus longer matches are ignored. In the Once policy, the phase exits after firing of a rule (so scanning for the *next* phase resumes from the root of the annotation graph), whereas the First policy continues to search for additional matches in the *current* phase, starting from the end of the matched span. For both policies, if more than one rule matches the same shortest span, the JAPE processor arbitrarily chooses one of them to be fired and the others are ignored. Multiple runs over the same input could therefore produce non-identical extractions.

With the Appelt policy, spans, starting at some point, are tested from longest to shortest, and the longest matched span causes a rule to be fired. (Thus shorter matches are ignored.) When multiple rules in a phase can be fired for the same longest span of text, JAPE chooses a rule based on priority. A rule's priority with respect to other rules is determined first from the declared priority number in the rule (higher numbers having higher priority). If several matching rules have the same priority number, the rule that appears earlier in the phase fires. After firing a rule, scanning for additional matches in the current phase starts from the end of the matched span.

The Brill policy allows *all* the rules that match a span of any length starting from some point to be fired. As a result, overlapping annotations can be created within a single phase. After firing all matching rules, scanning for additional matches in the current phase starts from the end of the longest matched span.

Finally, the All policy allows all possible rules to fire, regardless of the starting point and the span of text that matches. Table 4.1 summarizes JAPE policies along with the outcome of polices once applied to Figure 4.5 that presents all possible matches for a hypothetical JAPE phase.

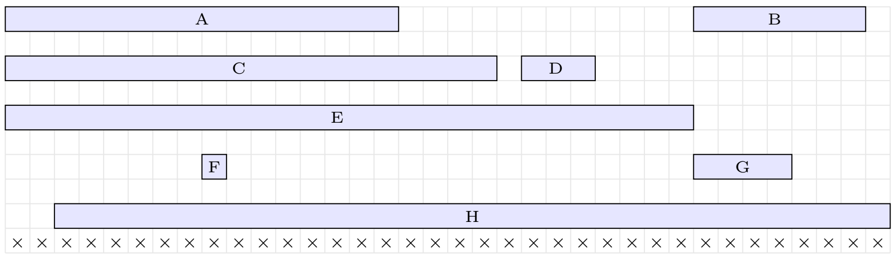

Figure 4.5: For a given phase and an input string $\times \cdots \times$, where $\times$ represents any input character, the figure visualizes the resulting matches after running the phase and scanning the corresponding annotation graph. We label each match by a capital letter.

Table 4.1: Summary of JAPE policies and their effects on Example in Figure 4.5.

Policy	Start Node	Priority	After Match	Next Node	Result
First	start of leftmost match	shortest	stay in curr. phase	end of shortest match	A D G
Once	start of leftmost match	shortest	exits curr. phase	start of annot. graph	A
Appelt	start of leftmost match	longest	stay in curr. phase	end of longest match	B E
Brill	start of leftmost match	all matches	stay in curr. phase	end of longest match	A C B E G
All	start of leftmost match	all matches	stay in curr. phase	start of next leftmost match	A B C D E F G H

The first phase shown in Figure 4.3 has one rule that matches *Tokens* and *SpaceTokens* and uses the Appelt policy (longest match). It recognizes the strings `http://`, `ftp://`, and `www.` as common indicators of the start of a URL, and annotates these as *UrlPre*. The second phase has two rules that match the annotations created during the first phase, as well as *Lookup* (a match in the gazetteer), *Token*, and *SpaceToken*, and again uses the Appelt policy. URLs that start with a string recognized in the first phase, followed by one to seven tokens with no intervening spaces, and possibly ending with either a *SpaceToken* or a period and a country code found in the dictionary are recognized by the first two rules. The third phase uses the Appelt policy to find maximal sequences of alphabetic tokens that begin with an uppercase letter followed by zero or more lowercase letters. The final phase uses the First policy (shortest match) to find strings that look like names followed by a URL (ignoring all annotations that are neither *Capitalized* nor *Url*).

4.1.2 Tokenizer

In a JAPE phase, rules only have access to annotations; the input's characters are not available. Thus, there needs to be a tokenization step in all JAPE programs which scans the input characters and, based on their lexical features, adds initial annotations. For simplicity, we design the tokenizer as a special JAPE phase with a predefined set of JAPE rules that have access to the input characters and where the outputs of the phase are *Token* and *SpaceToken* and the underlying policy is All. In each rule, the subexpression associated with the binding label is a *non-empty* regular expression. Both *Token* and *SpaceToken* have an attribute called *kind*. For *Token*, based on the characteristics of the input, four values are possible for *kind* including: *word*, *number*, *symbol*, and *punctuation*. The attribute *kind* belonging to *Token* has sub-category attributes. For instance, a *Token* that is a *word* (*kind==word*) has an attribute called *orth* for which four values are possible: *upperInitial*, *allCaps*, *lowerCase*, and *mixedCaps*. For *SpaceToken* two values are possible for *kind*: *space* and *control*. The sole preprocessing applied to all input text involves attaching two special characters, $\boxed{\texttt{\$b}}$ at the beginning and $\boxed{\texttt{\$e}}$ at the end.

For readability, the tokenizer rules are built on top of some simple grammar building blocks. Also, there are in fact many other character sets and control characters that have not been included in Table 4.2.

character sets:

$$\begin{array}{ll}
\gamma_u = [\boxed{\texttt{A}}, \boxed{\texttt{Z}}] & \gamma_l = [\boxed{\texttt{a}}, \boxed{\texttt{z}}] \\
\gamma_d = [\boxed{\texttt{0}}, \boxed{\texttt{9}}] & \gamma_c = \boxed{\texttt{␣}} \vee \boxed{\texttt{CR}} \vee \boxed{\texttt{LF}} \\
\gamma_{bSp} = (\gamma_l \vee \gamma_u \vee \gamma_d \vee \boxed{\texttt{\$b}} \vee \gamma_p \vee \boxed{\texttt{CR}} \vee \boxed{\texttt{LF}}) & \gamma_{aSp} = (\gamma_l \vee \gamma_u \vee \gamma_d \vee \boxed{\texttt{\$e}} \vee \gamma_p \vee \boxed{\texttt{CR}} \vee \boxed{\texttt{LF}}) \\
\gamma_{bDg} = (\gamma_l \vee \gamma_u \vee \boxed{\texttt{\$b}} \vee \gamma_c \vee \gamma_p) & \gamma_{aDg} = (\gamma_l \vee \gamma_u \vee \boxed{\texttt{\$e}} \vee \gamma_c \vee \gamma_p) \\
\gamma_{bW} = (\gamma_d \vee \boxed{\texttt{\$b}} \vee \gamma_c \vee \gamma_p) & \gamma_{aW} = (\gamma_d \vee \boxed{\texttt{\$e}} \vee \gamma_c \vee \gamma_p) \\
\gamma_p = (\boxed{\texttt{:}} \vee \boxed{\texttt{,}} \vee \boxed{\texttt{;}} \vee \boxed{\texttt{!}} \vee \boxed{\texttt{.}} \vee \boxed{\texttt{?}}) &
\end{array}$$

Table 4.2: Sample tokenizer rules.

$R_1 : \gamma_{bSp}(\boxed{\textvisiblespace}\boxed{\textvisiblespace}^*)\text{: } ST\gamma_{aSp} \rightarrow ST.SpaceToken = \{kind = space\}$
$R_2 : \gamma_{bDg}(\gamma_d\gamma_d^*)\text{: } T\gamma_{aDg} \rightarrow T.Token = \{kind = number\}$
$R_3 : \gamma_{bW}(\gamma_u\gamma_l^*)\text{: } T\gamma_{aW} \rightarrow T.Token = \{kind = word, orth = upperInitial\}$
$R_4 : \gamma_{bW}(\gamma_l\gamma_l^*)\text{: } T\gamma_{aW} \rightarrow T.Token = \{kind = word, orth = lowercase\}$
$R_5 : \gamma_{bW}(\gamma_u\gamma_u\gamma_u^*)\text{: } T\gamma_{aW} \rightarrow T.Token = \{kind = word, orth = uppercase\}$
$R_6 : \gamma_{bW}((\gamma_u \vee \gamma_l)(\gamma_l(\gamma_u \vee \gamma_l)^*\gamma_u(\gamma_u \vee \gamma_l)^* \vee \gamma_u(\gamma_u \vee \gamma_l)^*\gamma_l(\gamma_u \vee \gamma_l)^*))\text{: } T\gamma_{aW} \rightarrow$ $T.Token = \{kind = word, orth = mixedCaps\}$

After removing duplicate annotations for each span, the outcome of the tokenizer is a linear annotation graph over the input string.

4.1.3 Gazetteer

In a JAPE phase we have access to a gazetteer which represents a predefined set of entities such as names of cities, organizations, days of the week, among others. In ANNIE, the gazetteer consists of multiple files representing set of entities, i.e., *file_name.lst*. For instance, there can be a list for departments, departments.lst, that contains following strings:

departments.lst

```
Agriculture Department
Commerce Department
Department of Agriculture
Department of Health and Human Services
Department of Housing and Community Development
Department of Interior
```

There is a simple index file, i.e., *lists.def*, to access list files. In the index file, each file is associated with attributes including: *Annotation Type* (for all lists its value is *lookup*), *MajorType*, and *MinorType*. Based on the semantics of each list, meaningful values are assigned to attributes. For *departments.lst* three attributes are defined: *AnnotationType= lookup*, *MajorType=Organization*, and *MinorType=government*. We assume that the gazetteer used in our extraction environment is a special JAPE phase that is created before the program starts. For each *.lst* in *lists.def* a set of rules are created and added to the phase. The phase policy is All and its only output is *Lookup*. We follow the

wholeWordsOnly matching paradigm of ANNIE. Therefore, rules corresponding to the gazetteer are designed in such way that a *Lookup* annotation is created only if the entire gazetteer entry matches as in Table 4.3. Note that a span of text can be annotated by more that one *Lookup*.

Table 4.3: Sample Gazzetter rules created for *departments.lst*.

$$
\begin{aligned}
R_1 &: (\gamma_p \vee \gamma_c \vee \boxed{\texttt{\$b}})(\boxed{\texttt{Agriculture}}\boxed{\texttt{␣}}\boxed{\texttt{Department}}) : L(\gamma_p \vee \gamma_c \vee \boxed{\texttt{\$e}}) \rightarrow \\
&\quad L.Lookup = \{MajorType = Organization, MinorType = goverment\} \\
R_2 &: (\gamma_p \vee \gamma_c \vee \boxed{\texttt{\$b}})(\boxed{\texttt{Commerce}}\boxed{\texttt{␣}}\boxed{\texttt{Department}}): L(\gamma_p \vee \gamma_c \vee \boxed{\texttt{\$e}}) \rightarrow \\
&\quad L.Lookup = \{MajorType = Organization, MinorType = goverment\} \\
R_3 &: (\gamma_p \vee \gamma_c \vee \boxed{\texttt{\$b}})(\boxed{\texttt{Department}}\boxed{\texttt{␣}}\boxed{\texttt{of}}\boxed{\texttt{␣}}\boxed{\texttt{Agriculture}}): L(\gamma_p \vee \gamma_c \vee \boxed{\texttt{\$e}}) \rightarrow \\
&\quad L.Lookup = \{MajorType = Organization, MinorType = goverment\}
\end{aligned}
$$

4.1.4 Relation Generator

The relation generator is a special JAPE phase that runs after all regular phases and is denoted by P_E. Every JAPE rule has a JAVA function call on the right hand side. We denote the function as *makeRow(args)* where *args* are all the binding labels on the left hand side of the corresponding rule. The only action that is allowed by *makeRow(args)* is to form and output a row for every individual match of the rule without altering the annotation graph. Thus, the extracted relation has a column for each binding label. For every record in the output relation, attributes are populated with the values from input D which are matched by a JAPE rule in P_E or null if no span is bound to a corresponding label. Also, the output relation has a column that contains a unique id for every match.

Example 4.1.2

Consider the following phase as the relation generator in a JAPE program. As a result we have a relation with three columns: *urlAddress*, *countryCodeFar*, and *countryCodeClose*.

Phase: Url_Relation
Input: Lookup SpaceToken Token UrlPre
Options: control = appelt

```
Rule: Url1
```

```
Priority: 50
({UrlPre}({Token})[1,7]):urlAddress ({SpaceToken}) --> makeRow("urlAddress")

Rule: Url2
Priority: 100
({UrlPre} ({Token})[3]{Token.string=="."}
       ({Lookup.majorType==country_code}):countryCodeClose):urlAddress |
       ({UrlPre} ({Token})[5]{Token.string=="."}
       ({Lookup.majorType==country_code}):countryCodeFar):urlAddress
      --> makeRow("urlAddress", "countryCodeFar", "countryCodeClose")
```

4.2 Spanner Representation of JAPE Program

In this section, we overview how a Core JAPE program can be modelled by spanners.

4.2.1 Partially Functional Spanners

Transforming a JAPE program into a corresponding spanner might result in its execution producing a spanner relation in which some of the attributes have a null value. In this study, the interpretation of null values aligns with the notion of "nonexistent" [64]. That is, comparisons involving null are treated as if null were a specific value (i.e., NULL $==$ NULL is true and NULL $==$ *anything else* is false). In this section, first we generalize functional spanners defined in Section 3.1.3 to accommodate null values; second, we slightly modify the spanner algebra to support null values; finally, we provide a construction algorithm for the difference spanner that will be used in Algorithm 10.

Definition 13 *A document spanner is* partially functional *if its associated span relation i) is in first normal form , but possibly with null values ii) does not contain any record with null values for all attributes. Applying such a spanner to a document produces a span relation in which each row defines a non-empty partial function from attributes to the non-null values stored in that row.*

Partially Functional Spanner Algebra

Let D be an input document, and let E, E_1, and E_2 be partially functional spanners where the last two are union-compatible, i.e., $E_1 = \varnothing \lor E_2 = \varnothing \lor SVars(E_1) = SVars(E_2)$, and $S \subseteq SVars(E)$. The algebraic operators essential for our work are then defined as:

- The *projection* of E is a spanner represented as $\tilde{\pi}_S(E)$. For any arbitrary D, $\tilde{\pi}_S(E)(D)$ is obtained by i) restricting the domain of $E(D)$ to S and ii) eliminating any record for which all attributes have null value.
- For two union compatible partially functional spanners, the union spanner simply extends union as defined in Section 3.1.3 to partially functional spanners.
- The *join* spanner is represented as $E \tilde{\bowtie} E_1$ where $SVar(E \tilde{\bowtie} E_1) = SVars(E_1) \cup SVars(E)$. For an input document, the span relation associated with the join spanner $E \tilde{\bowtie} E_1(D)$ is defined similarly to $E \bowtie E_1(D)$ except it includes also tuples that have matching nulls for common variables.
- For two union compatible (partially) functional spanners, the *difference* spanner is represented as $E_1 \setminus E_2$ where $SVars(E_1 \setminus E_2) = SVars(E_1)$. For an input document $E_1 \setminus E_2(D) = E_1(D) \setminus E_2(D)$.

For the sake of simplicity, we drop the tilde ˜ from these algebraic operators when there is no possibility of confusion. Next, we revise Morciano's well-behaved property and the algorithms for operating on functional spanners to accommodate null values.

Definition 14 *An eVset-automaton A is* partially well-behaved *if, for every complete path p in A we have: 1) every variable opens and closes* ***at most once****, 2) each variable opens before it is closed , and 3) at least one variable opens and closes in p.*

Proposition 5 *A spanner represented by a partially well-behaved eVset-automaton is partially functional.*

Proof. By definition. □

Join: Construction proposed by Morciano for join works for partially well-behaved eVset-automaton, the resulting eVset-automaton is partially well-behaved.

Union: The construction proposed by Morciano works and the output is partially well-behaved.

Projection: We develop Algorithm 7 below based on Morciano's construction algorithm, Theorem 4.1 [72].

Proposition 6 *Given a partially well-behaved eVset-automaton A and $Y = \{y_1, \cdots, y_n\} \subseteq SVars(A)$, Algorithm 7 produces a partially functional spanner that represents $\pi_Y[\![A]\!]$.*

Proof. Applying projection to a partially well-behaved eVset-automaton might result in an all-null record. The join in Line 4 keeps only those records that have at least one none-null value. □

Algorithm 7 Projection for Partially functional Spanners.

```
   Input: eVset-automaton $A$, Projected Variables $Y \subseteq SVars(A)$
   Output: Spanner $S$
   Precondition: $A$ is partially well-behaved.
1: $S \leftarrow \emptyset$
2: $A' \leftarrow MCons(A, Y)$                    ▷ Call Morciano's projection algorithm
3: for all $y_i \in Y$ do
4:     $S \leftarrow S \cup [\![\Sigma^* y_i\{\Sigma^*\}\Sigma^*]\!] \bowtie [\![A']\!]$
5: end for
6: if $VOp(S) == \emptyset$ then
7:     return $\emptyset$                        ▷ There is no variable operation in $S$
8: end if
9: return $S$
```

Difference: Morciano [72] does not include an implementation of difference, even though Fagin et al. [37] proved that the operation is supported by variable-set automata. We follow Fagin et al.'s proof that variable-set automata are closed under difference and Morciano's construction for joins to show how difference can be supported by automata. We first define a variant type of transition, adapted from the definition of *vset-path* introduced by Fagin et al. [37]:

Definition 15 *Given a set of states Q, a set of symbols Σ, and a set of variables V, a* regex-transition *t is of the form (q_i, rS, q_j) where $\{q_i, q_j\} \subseteq Q$, r is a regular expression over Σ, and S is a (possibly empty) set of open and close operations over V. If $x \vdash \in S$, we* *say that t* opens *x and if $\dashv x \in S$, we say that t* closes *x; also, if T is a set of transitions, we say that T opens/closes x if any transition $t \in T$ opens/closes x. A* regex-transition graph *is a sextuple $(\Sigma, V, Q, q_0, q_f, \delta)$ where $q_0 \in Q$ is the initial state, $q_f \in Q$ is the final state, and δ is a set of regex-transitions over Σ and V.*

Next, given an automaton $A = (Q, q_0, q_f, \delta)$ defined over alphabet Σ and $V = SVars(A)$, and continuing to adapt the construction of vset-paths, we define a similar set of paths $\Pi(A)$ as follows:

1. Initialize the set $Q' = Q$ and $\delta' = \delta$. Then,
   - If q_0 has any incoming transitions, create a new state q'_0, add it Q', and add (q'_0, ϵ, q_0) to δ'; otherwise, let $q'_0 = q_0$.
   - Replace any empty operator transition $(q_i, \varnothing, q_j) \in \delta'$ by (q_i, ϵ, q_j).
   - For each state $q_j \in Q$ such that there is at least one incoming non-empty operator transition and at least one incoming character or ϵ transition, create a new state q'_j, add it to Q', add the transition (q'_j, ϵ, q_j) and replace all operator transitions (q_i, S, q_j) by (q_i, S, q'_j).
   - Finally create a new state q'_f, add it Q', and add $(q_f, \varnothing, q'_f)$ to δ'.

   Now we can split the set Q' into two subsets: Q'_s for which all incoming transitions are character transitions and Q'_v for which all incoming transitions are operator transitions.

2. Using the state-removal approach described by Linz [65] for converting finite state automata into regular expressions, create the regex-transition graph $rx(A) = (\Sigma, V, Q'', q'_0, q'_f, \delta'')$ by removing all states in $Q'_s \setminus \{q'_0, q'_f\}$ and modifying the transitions to and from those states accordingly.

3. Finally, define $\Pi(A) = \{(\Sigma, V, Q''_p, q'_0, q'_f, \delta''_p) \mid p$ is a path in $rx(A)$ from q'_0 to $q'_f\}$ [4].

Proposition 7 *Given a spanner defined by a partially well-behaved automaton A, on each path $p \in \Pi(A)$, at least one variable is opened and closed, each variable in $SVars(A)$ is opened and closed at most once, and a variable is always opened before or at the same time as it is closed. Thus each path has at most $2|SVars(A)| + 1$ edges. Furthermore, only the final transition in p opens no variables and closes no variables.*

[4]It is easy to show that $|\Pi(A)|$ can be exponential in the number of capture variables.

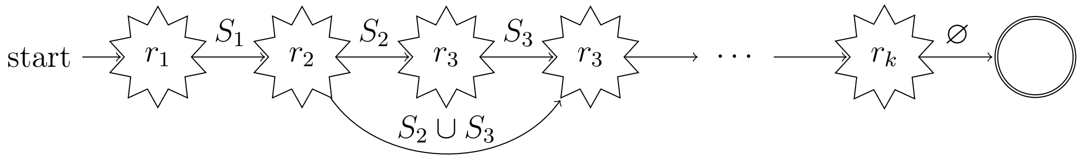

Figure 4.6: The automaton $\Lambda(p)$ corresponding to a path $p \in \Pi(A)$ that has k transitions: a star labelled r_i corresponds to a sub-automaton matching regular expression r_i; the edge labelled $S_2 \cup S_3$ is added because the regular expression r_3 matches ϵ.

Proof. The input automaton is partially well-behaved, and since the construction does not modify the order of variable transitions and creates a transition with the same operations, if we need to add a new state the proposition holds. □

Given a regex-transition graph $g = (\Sigma, V, Q_g, q_g^0, q_g^f, \delta_g)$ with labels of the form rS where r is a regular expression over Σ and S is a set of operations over V, let $\Lambda(g)$ be the automaton that results from converting regex-transitions in g into corresponding sub-automata (first matching the regular expression and then the set of open and close operations) followed by state pruning and operation closure (Figure 4.6).

Proposition 8 *Given automaton A, $[\![A]\!] = [\![\bigcup_{p\in\Pi(A)} \Lambda(p)]\!]$.*

Proof. This corresponds to vset-path unions (Lemma 4.2) as described in the paper by Fagin et al. [37]. □

Given automaton E and a path $p = (\Sigma, V, Q_p, q_p^0, q_p^f, \delta_p)$ in $\Pi(E)$, let p_i be a subpath in p on which the regular expression for every transition matches ϵ, where q_i^0 is the first state in p_i, q_i^f is the last state in p_i, δ_i is the set of transitions in p_i, and S_i is the union of all open and close operations on transitions in p_i. Define $\mathrm{cl}(p, p_i) = \delta_p \cup \{(q, rS_\cup, q_i^f) \mid (\exists S_0)(q, rS_0, q_i^0) \in \delta_p \wedge S_\cup = S_0 \cup S_i\}$ and $\mathrm{cl}(p) = \bigcup_{p_i} \mathrm{cl}(p, p_i)$.

Lemma 8 *Given two union-compatible automata E_1 and E_2 and two paths $p_1 \in \Pi(E_1)$ and $p_2 \in \Pi(E_2)$, there is an automaton $\ominus(p_1, p_2)$ such that for any document D, $[\![\ominus(p_1,p_2)]\!](D) = [\![\Lambda(p_1)]\!](D) \setminus [\![\Lambda(p_2)]\!](D)$. Moreover, this automaton can be created in polynomial time.*

Proof. Let $p_1 = (\Sigma, V, Q_1, q_0^1, q_m^1, \delta_1)$ and $p_2 = (\Sigma, V, Q_2, q_0^2, q_n^2, \delta_2)$. Denote a path passing through states $Q_1 = \{q_j^1 \mid 0 \leq j \leq m\}$ in increasing numerical order by $p_\delta^1 = \{(q_{j-1}^1, r_j^1 S_j^1, q_j^1) \mid 1 \leq j \leq m\}$, and let $p_\delta^2 = \{(q_{j-1}^2, r_j^2 S_j^2, q_j^2) \mid 1 \leq j \leq n\}$ be a path passing through states $Q_2 = \{q_j^2 \mid 0 \leq j \leq n\}$ in increasing numerical order. Let q_f' be distinct from all $q \in Q_2$ and $Q = Q_1 \times (Q_2 \cup \{q_f'\})$. Create $A = (\Sigma, V, Q, (q_0^1, q_0^2), (q_m^1, q_f'), \delta)$ where δ is the union of the following sets of transitions over Q:

$$\begin{aligned}
&\{((q_i^1, q_i^2), rS, (q_j^1, q_j^2)) \mid (\exists r_1, r_2) \\
&\qquad [(q_i^1, r_1 S, q_j^1) \in \mathrm{cl}(p_\delta^1) \wedge (q_i^2, r_2 S, q_j^2) \in \mathrm{cl}(p_\delta^2) \wedge L(r) = L(r_1) \cap L(r_2)]\} \qquad (4.2) \\
&\{((q_i^1, q_i^2), rS, (q_j^1, q_f')) \mid (\exists r_1, r_2) \\
&\qquad [(q_i^1, r_1 S, q_j^1) \in \mathrm{cl}(p_\delta^1) \wedge (q_i^2, r_2 S, q_j^2) \in \mathrm{cl}(p_\delta^2) \wedge L(r) = L(r_1) \setminus L(r_2)]\} \qquad (4.3)
\end{aligned}$$

$$\begin{aligned}
&\{((q_i^1, q_i^2), r_1 S, (q_j^1, q_f')) \mid (q_i^1, r_1 S, q_j^1) \in \mathrm{cl}(p_\delta^1) \wedge (\nexists q \in Q_2)(\nexists r_2)[(q_i^2, r_2 S, q) \in \mathrm{cl}(p_\delta^2)]\} \qquad (4.4) \\
&\{((q_i^1, q_f'), r_1 S, (q_j^1, q_f')) \mid (q_i^1, r_1 S, q_j^1) \in \mathrm{cl}(p_\delta^1)\} \qquad (4.5)
\end{aligned}$$

Finally $\ominus(p_1, p_2) = \Lambda(A)$.

Given an arbitrary input string $D = s_1 s_2 ... s_k$, we need to show (i) $[\![\ominus(p_1, p_2)]\!](D) \subseteq [\![\Lambda(p_1)]\!](D) \setminus [\![\Lambda(p_2)]\!](D)$ and (ii) $[\![\Lambda(p_1)]\!](D) \setminus [\![\Lambda(p_2)]\!](D) \subseteq [\![\ominus(p_1, p_2)]\!](D)$.

- *if*: Given a row in $[\![\ominus(p_1, p_2)]\!](D)$, let the sequence of configurations $\rho =< ((q_0^1, q_0^2), 1), \cdots, ((q_i^1, q_j^2), k), \cdots, ((q_m^1, q_f'), |D| + 1) >$ represent the corresponding accepting run of A for D. ρ can be decomposed into two parallel runs, namely, $\rho_1 =< (q_0^1, 1), \cdots, (q_i^1, k), \cdots, (q_m^1, |D| + 1) >$ and $\rho_2 =< (q_0^2, 1), \cdots, (q_j^2, k), \cdots, (q_f', |D| + 1) >$. By construction, ρ_1 is an accepting run of $\Lambda(p_1)$ consuming D with an identical sequence and positions for all variable operations. Thus an identical row will occur in $[\![\Lambda(p_1)]\!](D)$. However, by construction q_f' is an error state for $\Lambda(p_2)$ thus there is no row in $[\![\Lambda(p_2)]\!](D)$ with the identical sequence and positions for all variable operations.

- *only if*: Given an arbitrary input string D we show that $[\![\Lambda(p_1)]\!](D) \setminus [\![\Lambda(p_2)]\!](D) \subseteq [\![A]\!](D)$. Thus if a row r shows up in $[\![\Lambda(p_1)]\!](D)$ and not in $[\![\Lambda(p_2)]\!](D)$ it shows up in $[\![A]\!](D)$, otherwise r does not show up in $[\![A]\!](D)$. There are multiple cases:
  - $\Lambda(p_1)$ recognizes the input string D with the corresponding run $\rho_1 = (q_0^1, 1), \cdots, (q_i^1, k), (q_{i+1}^1, k + 1), \cdots, (q_m^1, |D| + 1)$ but $\Lambda(p_2)$ does not recognize D and

cannot proceed starting from offset k on D, with the corresponding run $\rho_2 = (q_0^2, 1), \cdots, (q_j^2, k)$. Based on the construction (case 2) these two runs represent an accepting run of A $\rho = ((q_0^1, q_0^2), 1), \cdots, ((q_i^1, q_j^2), k), ((q_{i+1}^1, q'_f), k+1), \cdots, ((q_m^1, q'_f), |D|+1)$. Furthermore, all variable operations of ρ are consistent with ρ_1. Thus ρ_1 represents a row in $[\![A]\!]$.

– $\Lambda(p_1)$ recognizes the input string D with the corresponding run $\rho_1 = (q_0^1, 1), \cdots, (q_i^1, k), (q_{i+1}^1, k), \cdots, (q_m^1, |D|+1)$, which includes some variable operations S at offset k of D. Now assume $\Lambda(p_2)$ cannot proceed at offset k on D doing the same variable operations S, and let the corresponding run be $\rho_2 = (q_0^2, 1), \cdots, (q_j^2, k)$. Based on the construction (case 3) these two runs represent an accepting run of A $\rho = ((q_0^1, q_0^2), 1), \cdots, ((q_i^1, q_j^2), k), ((q_{i+1}^1, q'_f), k), \cdots, ((q_m^1, q'_f), |D|+1)$. Furthermore, all variable operations of ρ are consistent with ρ_1. Thus ρ_1 represents a row in $[\![A]\!]$ with no corresponding run of $\Lambda(p_2)$.

– $\Lambda(p_1)$ recognizes the input string D with the corresponding run $\rho_1 = (q_0^1, 1), \cdots, (q_i^1, k), (q_{i+1}^1, k+1), \cdots, (q_m^1, |D|+1)$, and $\Lambda(p_2)$ recognizes D with the corresponding run $\rho_2 = (q_0^2, 1), \cdots, (q_j^2, k) \cdots, (q_n^2, |D|+1)$. Let all the variable operation of ρ_2 be consistent with ρ_1 so both runs represent an identical row in $[\![\Lambda(p1)]\!]$ and $[\![\Lambda(p_1)]\!]$. Based on the construction (case 1) these two runs represent a failed run of A which starts at initial state (q_0^1, q_0^2), consumes the input, and ends up in a non-final state (q_m^1, q_n^2) and by the construction of the regex-transition graph, there is no outgoing edges from q_n and q_m, thus this row will not appear in $[\![A]\!](D)$.

Complexity Analysis: In the worst case, there is a path between each pair of states with ϵ on all its transitions. Thus, we add $O(m)$ transitions to each state in p_1 and $O(n)$ transitions to each state in p_2. There are $O(m^2n^2)$ of $[(q_i^1, q_j^2), (q_{i'}^1, q_{j'}^2)]$ thus we do $2 \times O(m^2n^2)$ comparisons to construct $\ominus(p_1, p_2)$. □

Proposition 9 *Given two union-compatible automata E_1 and E_2, for any document D, $[\![E_1 \setminus E_2]\!](D) = [\![E_1]\!](D) \setminus [\![E_2]\!](D)$ where*

$$E_1 \setminus E_2 = \bigcap_{p_j \in \Pi(E_2)} \bigcup_{p_i \in \Pi(E_1)} \ominus(p_i, p_j)$$

Proof. Let Υ_V^Σ represent the universal spanner [37] where Σ is the alphabet and V represents the set of variables.

$\forall \rho \in \Upsilon_V^\Sigma$ where $V = SVars(E_1), \rho \in [\![E_1 \setminus E_2]\!] \iff \rho \in [\![E_1]\!] \wedge \rho \notin [\![E_2]\!]$ (by definition),

$$[\![E_1]\!] = [\![\bigcup_{p_i^1 \in \Pi(E_1)} \Lambda(p_i^1)]\!] = \bigcup_{p_i^1 \in \Pi(E_1)} [\![\Lambda(p_i^1)]\!] \text{ where } 1 \leq i \leq m, \text{ (proposition 8)},$$

$$\rho \in [\![E_1]\!] \iff \rho \in \bigcup_{p_i^1 \in \Pi(E_1)} [\![\Lambda(p_i^1)]\!] \iff \rho \in [\![\Lambda(p_1^1)]\!] \vee \rho \in [\![\Lambda(p_2^1)]\!] \vee \cdots \vee \rho \in [\![\Lambda(p_m^1)]\!],$$

$$[\![E_2]\!] = [\![\bigcup_{p_j^2 \in \Pi(E_2)} \Lambda(p_j^2)]\!] = \bigcup_{p_j^2 \in \Pi(E_2)} [\![\Lambda(p_j^2)]\!] \text{ where } 1 \leq j \leq n, \text{ (proposition 8)},$$

$$\rho \notin [\![E_2]\!] \iff \rho \notin \bigcup_{p_j^2 \in \Pi(E_2)} [\![\Lambda(p_j^2)]\!] \iff \rho \notin [\![\Lambda(p_1^2)]\!] \wedge \rho \notin [\![\Lambda(p_2^2)]\!] \wedge \cdots \wedge \rho \notin [\![\Lambda(p_n^2)]\!],$$

$$\rho \in [\![E_1]\!] \wedge \rho \notin [\![E_2]\!] \iff \Big(\rho \in [\![\Lambda(p_1^1)]\!] \vee \rho \in [\![\Lambda(p_2^1)]\!] \vee \cdots \vee \rho \in [\![\Lambda(p_m^1)]\!]\Big) \wedge \Big(\rho \notin [\![\Lambda(p_1^2)]\!]$$
$$\wedge \rho \notin [\![\Lambda(p_2^2)]\!] \wedge \cdots \wedge \rho \notin [\![\Lambda(p_n^2)]\!]\Big) \iff$$
$$\Big((\rho \in [\![\Lambda(p_1^1)]\!]) \wedge \rho \notin [\![\Lambda(p_1^2]\!]) \vee \cdots \vee (\rho \in [\![\Lambda(p_m^1)]\!] \wedge \rho \notin [\![\Lambda(p_1^2)]\!])\Big) \wedge \cdots \wedge$$
$$\Big((\rho \in [\![\Lambda(p_1^1)]\!] \wedge \rho \notin [\![\Lambda(p_n^2)]\!]) \vee \cdots \vee (\rho \in [\![\Lambda(p_m^1)]\!] \wedge \rho \notin [\![\Lambda(p_n^2)]\!])\Big) \iff$$
$$\Big(\rho \in \ominus(p_1^1, p_1^2) \vee \cdots \vee \rho \in \ominus(p_m^1, p_1^2)\Big) \wedge \cdots \wedge \Big(\rho \in \ominus(p_1^1, p_n^2) \vee \cdots \vee \rho \in \ominus(p_m^1, p_n^2)\Big) \iff$$
$$\rho \in \bigcap_{p_j^2 \in \Pi(E_2)} \bigcup_{p_i^1 \in \Pi(E_1)} \ominus(p_i^1, p_j^2).$$

□

Handling of Spanner's Capture Variables

JAPE programs use annotations and binding labels to distinguish and mark specific parts of the input text which conform to predefined rules. These parts are then available for further processing on the right-hand side of the rule or in the next phases. In a Core JAPE program, binding labels and annotations serve three roles that are essential for our work: in a regular phase 1) annotations represent constraints on the annotation graph 2) they represent the new annotations that are going to be added to the annotation graph once the rule is fired; in an extraction phase 3) binding labels specify parts of the text to be extracted. Similarly, document spanners rely on capture variables to mark spans of input text. We rely on creating, retrieving and manipulating capture variables to represent binding variables, annotations and corresponding features that appear in JAPE phases. In converting JAPE programs to document spanners, whenever an annotation and its attributes appear in a rule, we represent it by a specific type of capture variable in the corresponding spanner. Four classes of capture variables are defined for a corresponding spanner that is denoted by S:

Created by reference: whenever an annotation (or attribute expression) is used in a constraint, a *reference variable* represents it in the associated spanner. The set containing all reference variables of a spanner in denoted by $V_r(S)$.

Created by addition: a variable that represents an annotation or an annotation-attribute-value triple that is added by a rule. The corresponding set is denoted by $V_a(S)$.

Created by extraction: in the relation generator phase a variable that represents a binding label extracted as a column is an *extraction variable*. The corresponding set is denoted by $V_e(S)$.

Created to mark cover: *cover variables* are used to mark the whole region of text that is consumed by a rule, in the associated spanner (see Line 37 of Algorithm 9). These variables are used for overlap analysis. The corresponding set is denoted by $V_c(S)$.

Thus, for a given spanner S representing a JAPE program, the set of variables is $V(S) = V_r(S) \cup V_c(S) \cup V_a(S) \cup V_e(S)$.

We use two global tables, accessible in the scope of the JAPE program, to manage variables and their metadata 1) Throughout a JAPE program, a repeated capture variable is assigned to identical annotations. However, two identical attribute expressions in two different annotations have different variables assigned. The first table, denoted as *MTable*, maintains a bidirectional mapping from annotations and attributes to their equivalent variables; 2) Every capture variable has a canonical name and zero or more aliases. For each variable, *VTable* records the corresponding spanner, name, category (above classes), and whether the label is an alias. If it belongs to the cover class, we keep the associated rule identifier and disjunct dentifier too. If it is an alias, its associated canonical name is recorded. Various primitive functions are designed to work with variables and their counterparts.

- $toVar(A)$: For the input set of annotation(s) or attribute expression(s), A, returns corresponding variables stored in *MTable*, where the variables are unique for each annotation or annotation-attribute-value triple.

  Example 4.2.1

In Figure 4.3b a capture variable will be defined for *Url* which is added to the annotation graph by all the rules in the phase. *Url* determines scope for attributes such as *kind="urlAddress"*, *rule="UrlContext"*, or *rule="UrlGuess"*. A single capture variable is defined for all instances of *kind="urlAddress"* of *Url* and distinct capture variables are also defined for each of *rule="UrlContext"* and *rule="UrlGuess"*.

- $getAnnots(I)$: extracts the set of annotations from the input and materialize the required mappings.
- $getAttrExprs(I)$: extracts the set of attribute expressions (possibly empty) along with corresponding annotations from the input.
- $getVar(A)$: returns a variable associated to input annotation A.
- $VarConf(regex1, regex2)$: renames all variables in $regex2$ that occur in $regex1$ and return the modified $regex2$. The new variables are given alias names. Finally, *VTable* is updated to add the new variables and their metadata.

Tokenizer and Gazetteer as Spanner

Proposition 10 *Given the tokenizer or gazetter phase $\mathcal{I}$, Algorithm 8 constructs a spanner S that 1) is partially functional 2) marks exactly the same spans marked by $\mathcal{I}$ for all input documents where the capture variables marking the spans represent annotations and corresponding attributes.*

Proof. 1) By construction every disjunct in the resulting regular expression is partially functional because every rule in $\mathcal{I}$ has exactly one regular expression and one annotation.

2) All substrings of the input document that are matched by $\mathcal{I}$ are matched by the spanner because the regular expression of every rule in $\mathcal{I}$ shows up as one disjunct of the output spanner, and concatenating Σ^* in Line 9 models the unanchored matching of JAPE. Annotations and their associated attributes can be restored from capture variables because, for each rule, Line 6 encloses the left-hand side subexpression bound to the binding label with variables that correspond to the annotation and attributes on the right-hand side. Consider the left-hand side expression of R_3 in Table 4.2, the formed string is $\gamma_{bW} v_1 v_2 v_3 \{\{\{\gamma_u(\gamma_l)^*\}\}\} \gamma_{aW}$ where $\{v_1\} \leftarrow toVar(\{Token\})$, $\{v_2\} \leftarrow toVar(\{orth = upperInitial\})$, and $\{v_3\} \leftarrow toVar(\{kind = word\})$. $\square$

Algorithm 8 Tokenizer/Gazetteer to Spanner Algorithm.

```
    Input: Tokenizer/Gazetteer Phase $\mathcal{I}$
    Output: Spanner $\mathcal{S}$
    Precondition: alphabet of Token and SpaceToken is disjoint in Tokenizer.
 1: $\mathcal{S} \leftarrow \emptyset$                                  ▷ define an empty spanner
 2: $rgxStr \leftarrow$ “”                                  ▷ define an empty string
 3: for all $R \in \mathcal{I}$ do                                  ▷ for each $\mathcal{I}$'s rule
 4:     $V_a \leftarrow toVar(getAnnts(R)) \cup toVar(getAttrExps(R))$
 5:     $L \leftarrow LHS(R)$                                  ▷ retrieve left-hand side string
        ▷ enclose the subexpression covered by the binding label in $L$ by capture variables
 6:     $L \leftarrow encloseBoundExp(L, V_a)$
                    ▷ append $L$, with a valid disjunction symbol to the resulting string
 7:     $rgxStr \leftarrow append(L, \text{“} \vee \text{”}, rgxStr)$
 8: end for
         ▷ append $\Sigma^*$ with a valid concatenation symbol to beginning and end of $rgxStr$
 9: $rgxStr \leftarrow append(\text{“}\Sigma^*\text{”}, \text{“} \bullet \text{”}, rgxStr, \text{“} \bullet \text{”}, \text{“}\Sigma^*\text{”})$
10: $\mathcal{S} \leftarrow [\![toRegExp(rgxStr)]\!]$
11: return $\mathcal{S}$
```

4.2.2 JAPE Rule Spanner Representation

Syntactically, the left-hand side of a JAPE rule can be viewed as a regular expression with capture variables. The use of a sub-expression of the form $(e) : A$ in a JAPE rule, where e is a regular expression over constraints, is similar to $A\{e\}$ in a variable regex, where e is a regular expression over input characters. However, in JAPE whenever the enclosed regular expression matches a sub-string, that sub-string is made available through the associated binding variable to undergo further processing on the right-hand side, whereas in the spanner formalism the substring's start and end offsets are merely assigned to the capture variable.

We rely on phases' input and policies, along with the rules' structures, to construct equivalent spanner representations for a JAPE program. A phase relies on the output of some preceding phases and perhaps the tokenizer and/or gazetteer.

Definition 16 *For a given JAPE phase P with input $I = \{A_i | A_i$ is an annotation$\}$, a preceding phase P' is called a* relevant phase *with* relevant annotations *I_r where P' outputs I' and $I_r = I' \cap I \neq \emptyset$. Furthermore, the spanner representation of a relevant phase is called a* relevant spanner*.*

Example 4.2.2

Consider the phase ***NameAndURL*** in Figure 4.3d: ***Name*** and ***Ulr*** are relevant phases with relevant annotations of *Capitalized* and *Url* respectively.

We design Algorithm 9 to create an equivalent spanner for a JAPE phase. The important steps are explained below:

- Line 10 calls a procedure to transform the left-hand side of the rule into a spanner using recursive descent based on the Core JAPE grammar (Figure 4.2).
- Line 119: if a variable is already used, we rename the repeated variables and both global tables are modified accordingly.
- Line 81: $allComb(E, \Phi)$ generates all valid combinations of variables in $V_r(E)$ and corresponding relevant spanners: For each element $\beta = \{< v_r, v_a, S' >\}$, returned by the function, $|\beta| = |V_r(E)|$ and β describes a total function from $V_r(E)$ to $V_a(S')$ and spanner S' corresponds to previous phases that could have produced values for $V_r(E)$.
- Line 92: $Filler(annots : List)$ constructs a regular expression that recognizes any strings that might appear between two consecutive annotations.

Remark 1 *The left outer join ($⟕$ in Line 86) is accomplished with a minor adjustment to Morciano's join algorithm.*

Proposition 11 *For a JAPE phase P_i, Algorithm 9 constructs a spanner $\mathcal{S}_i$ in such a way that for each binding label l in P_i, there is a corresponding capture variable in S_i that marks a region if that region is marked by l.*

Proof. Consider P_i to be a phase in a sequence: $\{P_T, P_G\}, P_0, \cdots, P_{i-1}, P_i, \cdots, P_n$ where P_T and P_G stand for the Tokenizer and the Gazetteer, respectively. At least one of $\{P_T, P_G\}$ must be executed before the user-provided program runs. In any phase four conditions need to be met for a rule to fire:

1. There must be an ordering of the phase's visible annotations in the annotation graph consistent with the order in which those annotations are used on the left-hand side of the rule.

2. If two visible annotations are considered adjacent by the rule, based on the phase's visible annotations, they might not be adjacent over input characters. For example, if *SpaceToken* is invisible, two adjacent annotations might be separated by any number of $\boxed{\llcorner\!\lrcorner}$ characters.

3. The constraints over annotations must match the annotations present in the graph.

4. The constraints over input characters (dynamic attribute expressions) must match the input.

After these conditions are fulfilled, the corresponding annotations from the right-hand side of the rule are appended to a section of the annotation graph that is bound to the associated label. We reason that every rule in P_i is simulated by a spanner, generated by Algorithm 9, which fulfills equivalent conditions to match the input.

Base cases: P_T and P_G are correctly converted to spanners S_T and S_G, respectively (Proposition 10).

Inductive step: Assume that every phase before P_i is correctly converted to a spanner. This implies that every span marked by a capture variable in any previous phase has a binding label, consequently an associated annotation on the annotation graph with the same offsets in the input text. Therefore the input to S_i matches the input to P_i. For every rule a regular expression with capture variables is created (Line 10) in such a way that every annotation on the left side of the rule is represented by a variable that encloses Σ^*. By construction, the ordering of visible annotations and corrsponding variables is identical. An additional variable corresponding to each binding variable encloses those variables to match the rule. If two variables are supposed to be adjacent in a phase where the annotation graph is partially visible, they might not be adjacent over the complete annotation graph. *FILLER* creates a regular expression that recognizes strings (possibly some invalid strings too) that might show up between those variables. Therefore if two annotations are considered adjacent in the annotation graph, their corresponding variables after matching a string will not be separated by any unwanted substrings. By joining the created spanner with the previously defined spanner (Line 86), we ensure that the subexpressions enclosed by the variables match the constraints expressed in terms of regular expressions in the earlier phases. It is important to note that if a constraint is defined on a dynamic attribute expression, the corresponding regular expression language is constructed and explicitly enclosed by an associated variable in Line 54. This ensures that the dynamic attribute expression is properly represented and considered during the matching process. The spanner that results meets the conditions of the proposition. □

Corollary 1 *Given a JAPE program comprising n standard phases $P_0, \cdots P_{n-1}$, a tokenizer P_T, a gazetteer P_G, and a relation generator P_E with associated spanners generated by Algorithm 9 as $S_0, \cdots S_{n-1}$,S_E, S_T, and S_G, S_E extracts all spans extracted by P_E.*

Proof. The extracted regions marked by P_E are associated with regions marked by S_E using extraction variables (Lines 6 and 68). Subsequently, S_E are projected on cover and extraction variables (Line 16). □

Algorithm 9 JAPE Phase to Spanner Algorithm.

```
    Input: Phase $\mathcal{P}_i$
    Output: Spanner $\mathcal{S}_i$
    Precondition: Tokenizer, gazetteer, and all preceding phases are transformed
 1: $\mathcal{S}_i \leftarrow \emptyset$, $V(\mathcal{S}_i) \leftarrow \emptyset$        ▷ define an empty spanner
 2: $rgxStr \leftarrow$ “”
 3: for all $R \in \mathcal{P}_i$ do
 4:     $\mathcal{L} \leftarrow LHS(R)$        ▷ get expression on left-hand side of rule
 5:     if $\mathcal{P}_i$ is ordinary phase then
 6:         $V_a(E) \leftarrow toVar(getAnnots(R)) \cup toVar(getAttrExprs(R))$
 7:     else
 8:         $V_e(E) \leftarrow toVar(\mathcal{L}.getBindingLabels())$        ▷ relation generator phase
 9:     end if
                ▷ produce the string representation of the regex of the lefthand side
10:     $E \leftarrow$ TRANSFORM($\mathcal{L}$)
                ▷ append $E$, with a valid disjunction symbol to the resulting string
11:     $rgxStr \leftarrow append(E, “\vee”, rgxStr)$
12: end for
        ▷ append $\Sigma^*$ with a valid concatenation symbol to beginning and end of $rgxStr$
13: $rgxStr \leftarrow append(“\Sigma^* \bullet”, rgxStr, “\bullet \Sigma^*”)$
14: $\mathcal{S}_i \leftarrow [\![toRegExp(rgxStr)]\!]$
15: $\mathcal{S}_i \leftarrow$ PUSHDOWN($\mathcal{S}_i$)
                ▷ keep cover and addition or extraction variables of the current phase
16: $\mathcal{S}_i \leftarrow \pi_{\{V_c(\mathcal{S}_i) \cup V_a(\mathcal{S}_i) \cup V_e(\mathcal{S}_i)\}} \mathcal{S}_i$
17: return $\mathcal{S}_i$

18: procedure TRANSFORM($\mathcal{L} : String$)
19:     $S \leftarrow$ CONSTRAINTGROUP($\mathcal{L}$)
20:     $S \leftarrow S.kpExpdVars(getCoverVar(S))$        ▷ keep exposed cover variables
21:     return $S$
22: end procedure
                                                            ▷ cont. in the next page
```

Algorithm 9 JAPE Phase to Spanner Algorithm. cont.

```
23: procedure ConstraintGroup($cg : String$)
24:     $rgxCGstr \leftarrow$ “”
25:     $V_c \leftarrow \emptyset$                                        ▷ def. an empty set of cover variables
26:     $ptrnElms \leftarrow getPtnElms(cg)$                               ▷ split cg by highest level <bar>
27:     for all $pes \in PtnElms$ do
28:         $ps[1 \cdots n] \leftarrow getPtnElms(pes)$                    ▷ sep. consecutive pattern elements
29:         $rgxStr \leftarrow$ “”
30:         for all $p \in ps[1 \cdots n]$ do
31:             $tmpRgxStr \leftarrow$ PatternElement($p$)
32:             $annots \leftarrow \mathcal{P}_i.getVisibleAnnots()$
33:             $F \leftarrow toString($Filler($annots$)$)$
34:             $rgxStr \leftarrow append(rgxStr,$ “ $\bullet$ ”$, F,$ “ $\bullet$ ”, VarConf($rgxStr, tmpRgxStr$)$)$
35:         end for
36:         $V_c \leftarrow V_c \cup \{W_{pes}\}$                          ▷ def. a cover var.
37:         $rgxStr \leftarrow enclose(rgxStr, \{W_{pes}\})$                ▷ enclose with cover var.
38:         $rgxStr \leftarrow append(rgxStr,$ “ $\vee$ ”$, rgxCGstr)$
39:     end for
40:     return $rgxCGstr$
41: end procedure

42: procedure PatternElement($bpe : String$)
43:     if IsBasicPatternElement(bpe) then
44:         returnBasicPatternElement($bpe$)
45:     else
46:         returnComplexPatternElement($bpe$)
47:     end if
48: end procedure                                                      ▷ cont. in the next page
```

Algorithm 9 JAPE Phase to Spanner Algorithm. cont.

```
49: procedure BasicPatternElement($bpe : String$)
50:     $V \leftarrow \emptyset$
51:     $Cs \leftarrow getConstraints(bpe)$                              ▷ split by $< comma >$
52:     if $Cs$ has dyn. attr. then                        ▷ e.g. {Token.string=="sample"}
53:         $dynCs \leftarrow getDynAttr(CS)$
                                ▷ conjunction of corresponding regular expressions
54:         $rgxStr \leftarrow toRegexStr(dynCs)$
55:     else
56:         $rgxStr \leftarrow$ “$\Sigma^*$”
57:     end if
58:     $V' \leftarrow getVars(Cs)$
59:     $rgxStr \leftarrow enclose(rgxStr, V')$
60:     $V_r \leftarrow V_r \cup V'$
61:     return $rgxStr$
62: end procedure

63: procedure ComplexPatternElement($cpe : String$)
64:     $CG \leftarrow getCGroup(cpe)$
65:     $rept \leftarrow cpe.getRep()$         ▷ get corresponding repetition if exists 0 otherwise
66:     $labels \leftarrow cpe.getBindingLables()$         ▷ get corresponding binding labels if exist
67:     if phase is relation generator then
68:         $V_{tmp} \leftarrow getVar(labels)$
69:     else                                                      ▷ an ordinary phase
70:         $V_{tmp} \leftarrow getVar(getAnnots(labels)) \cup getVar(getAttrExprs(labels))$
71:     end if
72:     $rgxStr \leftarrow$ (ConstraintGroup($CG$))
73:     $i \leftarrow 1$, $exStr \leftarrow rgxStr$
74:     while $i < rept$ do                                    ▷ apply repetition
75:         $exStr \leftarrow append(exStr,$ “ $\bullet$ ”, VarConf($exStr, rgxStr$))
76:         $i \leftarrow i + 1$
77:     end while
78:     return $enclose($VarConf($V_{tmp}, exStr$)$, V_{tmp})$
79: end procedure
```

Algorithm 9 JAPE Phase to Spanner Algorithm. cont.

```
80: procedure PushDown($E : rgx$)
81:     $\mathcal{B} \leftarrow$ allComb($E$)                                    ▷ $< v, v_a, spanner >$
82:     $\mathcal{H} \leftarrow \emptyset$                                           ▷ define an empty spanner
83:     for all $\beta \in \mathcal{B}$ do
84:         $\mathcal{G} \leftarrow [\![E]\!]$
85:         for all $< v, v_a, \mathcal{S}' > \in \beta$ do          ▷ $v$ is refered by $E$; $v_a$ is added by $\mathcal{S}'$
86:             $\mathcal{G} \leftarrow \mathcal{G} ⟕ \rho_{v_a \rightarrow v} \pi_{v_a}(\mathcal{S}')$                ▷ perform a left outer join
87:         end for
88:         $\mathcal{H} \leftarrow \mathcal{G} \cup \mathcal{H}$
89:     end for
90:     return $\mathcal{H}$
91: end procedure

92: procedure Filler($annots : set$)
93:     if $\{Token, spaceToken\} \subseteq annots$ then
94:         return $\emptyset$
95:     end if
96:     $c \leftarrow \{Token, spaceToken\} \cap annots$
97:     if $c \neq \emptyset$ then                          ▷ either $Token$ or $spaceToken$ is missing
98:         $tmp \leftarrow \{Token, spaceToken\} \setminus c$
99:         $v \leftarrow getVar(tmp)$
100:        $sp \leftarrow VTable.getSp(v)$                              ▷ get spanner having $v$
101:        $exps[] \leftarrow sp.enclosedBy(v)$
102:        $resultStr \leftarrow append(toString(exps[]), " \vee ")$
103:        $rgx \leftarrow toRegExp(resultStr)$
104:        ▷ only the expression accepting arbitrary number of missing initial annotation
    can fill the gap
105:        return $rgx^*$
106:    else
107:        return $\Sigma^*$    ▷ we do not know what can appear in between in terms of initial
    annotations
108:    end if
109: end procedure
```

Algorithm 9 JAPE Phase to Spanner Algorithm. cont.

```
110: procedure VarConf($rgxStr : String, tmpRgxStr : String$)
111:     if $rgxStr$ represents a $Regex$ then
112:         $CurrVars \leftarrow getVars(rgxStr)$
113:     else
114:         $CurrVars \leftarrow rgxStr$
115:     end if
116:     $CurrVars \leftarrow getVars(rgxStr)$
117:     $tmpVars \leftarrow getVars(tmpRgxStr)$
118:     if $tempVars \cap CurrVars \neq \emptyset$ then
119:         $tmpRgxStr, Cfs \leftarrow renVars(tmpRgxStr, tmpVars \cap CurrVars)$
120:     end if
121:     $update(MTable, Cfs)$
122:     $update(VTable, Cfs)$
123:     return $tmpRgxStr$
124: end procedure

125: procedure AllComb($e : Regex$)
126:     $i \leftarrow 1$
127:     for all $v_r \in e.V_r$ do                    ▷ for every variable that is used in $e$
128:         $l_i \leftarrow newList[< var, var, spanner >]()$
129:         $v_{rc} \leftarrow VTable.getCanonicalLabel(v_r)$
130:         for all $r \in VTable$ do                  ▷ for all records
131:             if $r.var == v_{rc} \wedge r.var \in V_a$ then
132:                 $l_i.add(< v_r, r.var, r.spanner >)$
133:             end if
134:         end for
135:         $i \leftarrow i + 1$
136:     end for
137:     return $l_1 \times l_2 \cdots \times l_i$      ▷ returns all combinations as a list of lists
138: end procedure
```

Example

We present a Core JAPE program and outline the essential steps involved in converting the program into spanners. The simplistic program extracts instances of forenames, middle names, and last names from the input text using a simple tokenizer, a JAPE phase, and a relation generator. Figure 4.7 is the resulting relation if the extractor is applied to `Two notable musicians from Germany are Johann Sebastian Bach and Till Lindemann who ...`.

FN	MN	LN
Johann	Sebastian	Bach
Till	null	Lindemann

Figure 4.7: extracted records for the sample text.

For simplicity, we consider a tokenizer with only two rules that add only one type of annotation to the graph, i.e., *Token*.

```
Tokenizer
Rule: $R_3$
$\gamma_{bW}(\gamma_u(\gamma_l)^*):T\gamma_{aW}$ --> T.Token={orth=upperInitial,kind=word}
Rule: rule2
$\gamma_{bW}(\gamma_l(\gamma_l)^*):T\gamma_{aW}$ --> T.Token={orth=lowercase,kind=word}
```

```
Phase: tag_names
Input: Token
Options: control = appelt
Rule: Name1
{Token.orth = lowercase}(({Token.orth = upperInitial}):fname
      ({Token.orth = upperInitial}):lname){Token.orth = lowercase}
      --> :fname.Fname = {rule="Name1"}, :lname.Lname = {rule="Name1"}
Rule: Name2
{Token.orth = lowercase}(({Token.orth = upperInitial}):fname
      ({Token.orth = upperInitial}):mname ({Token.orth = upperInitial}):lname)
      {Token.orth = lowercase}
      --> :fname.Fname = {rule="Name2"},mname.Mname={rule=Name2},
      :lname.Lname = {rule="Name2"}
```

```
Phase: extract_names
Input: Fname Mname Lname
Options: control = appelt
Rule: Two-part-Name
({Fname.rule="Name1"}):FN ({Lname}):LN --> makeRow("FN","LN")
Rule: Three-part-Name
({Fname.rule="Name2"}):FN ({Mname}):MN ({Lname}):LN--> makeRow("FN", "MN","LN")
```

The spanner representing the tokenizer is:

$$S_T = \Sigma^*(\gamma_{bW} V_1 V_2 V_3\{\{\{\gamma_u(\gamma_l)^*\}\}\}\gamma_{aW} \cup \gamma_{bW} V_1 V_3 V_4\{\{\{\gamma_l(\gamma_l)^*\}\}\}\gamma_{aW})\Sigma^*$$

where $V = \{V_1, V_2, V_3, V_4\}$. Table 4.4 shows the variables that are created while converting the tokenizer to S_T.

Table 4.4: Available Variables created for S_T.

Variable	Annot./Attr.	Var. Type
V_1	$Token$	addition
V_2	$Token.orth = upperInitial$	addition
V_3	$Token.kind = word$	addition
V_4	$Token.orth = lowercase$	addition

Next, we convert the JAPE phase to a spanner. The initial spanner created for the phase *tag_names* and before pushdown is:

$$S'_{tn} = S_{Name1} \cup S_{Name2}$$

where S_{Name1} and S_{Name2} are spanners created for rules $Name1$ and $Name2$ respectively[5]:

$$\begin{aligned} S_{Name1} =& \Sigma^* V_7\{V_4\{\Sigma^*\}V_5 V_{51}\{\{V_2\{\Sigma^*\}\}\}V_6 V_{61}\{\{V'_2\{\Sigma^*\}\}\}V'_4\{\Sigma^*\}\}\Sigma^* \\ S_{Name2} =& \Sigma^* V_9\{V_4\{\Sigma^*\}V_5 V_{52}\{\{V_2\{\Sigma^*\}\}\}V_8 V_{82}\{\{V''_2\{\Sigma^*\}\}\}V_6 V_{62}\{\{V'''_2\{\Sigma^*\}\}\}V''_4\{\Sigma^*\}\}\Sigma^* \end{aligned}$$

After pushdown we end up with the following spanner:

$$\begin{aligned} S_{tn} =& \pi_V\big(S'_{tn} \bowtie \pi_{V_2} S_T \bowtie \pi_{V_4} S_T \bowtie (\rho_{V_2 \to V'_2} \pi_{V_2} S_T) \bowtie (\rho_{V_4 \to V'_4} \pi_{V_4} S_T) \\ & \bowtie (\rho_{V_2 \to V''_2} \pi_{V_2} S_T) \bowtie (\rho_{V_2 \to V'''_2} \pi_{V_2} S_T) \bowtie (\rho_{V_4 \to V''_4} \pi_{V_4} S_T)\big) \end{aligned}$$

where $V = \{V_5, V_{51}, V_{52}, V_6, V_{61}, V_{62}, V_7, V_8, V_{82}, V_9\}$. Note that, in our simplified approach, the phase policy is not applied to the spanner representation. As a result, the corresponding spanner represents each phase under the assumption that all phase rules have the opportunity to fire and mark the input text.

Table 4.5 shows the variables that are available or created when converting *tag_names*. Next, the relation generator phase is converted to its spanner representation. The initial

[5]We use color coding to differentiate variable types: red for cover variables, gray for variables with alias names, and black for all other variables.

Table 4.5: Available variables while/after creating S'_{tn}.

Variable	Annot./Attr.	Var.Type	Rule	Alias?	Canon.Name	Origin Phase
V_1	$Token$	addition	$---$	no	$---$	Tokenizer
V_2	$Token.orth = upperInitial$	addition	$---$	no	$---$	Tokenizer
V_3	$Token.kind = word$	addition	$---$	no	$---$	Tokenizer
V_4	$Token.orth = lowercase$	addition	$---$	no	$---$	Tokenizer
V_5	$Fname$	addition	$---$	no	$---$	tag-names
V_{51}	$Fname.rule = "Name1"$	addition	$---$	no	$---$	tag-names
V_{52}	$Fname.rule = "Name2"$	addition	$---$	no	$---$	tag-names
V_6	$Lname$	addition	$---$	no	$---$	tag-names
V_{61}	$Fname.rule = "Name1"$	addition	$---$	no	$---$	tag-names
V_{62}	$Fname.rule = "Name2"$	addition	$---$	no	$---$	tag-names
V_8	$Mname$	addition	$---$	no	$---$	tag-names
V_{82}	$Mname.rule = "Name2"$	addition	$---$	no	$---$	tag-names
V_2	$---$	reference	$---$	no	$---$	tag-names
V_4	$---$	reference	$---$	no	$---$	tag-names
V'_2	$---$	reference	$---$	yes	V_2	tag-names
V''_2	$---$	reference	$---$	yes	V_2	tag-names
V'''_2	$---$	reference	$---$	yes	V_2	tag-names
V'_4	$---$	reference	$---$	yes	V_4	tag-names
V''_4	$---$	reference	$---$	yes	V_4	tag-names
V_7	$---$	cover	*Name1*	no	$---$	tag-names
V_9	$---$	cover	*Name2*	no	$---$	tag-names

spanner for the relation generator before pushdown is:

$$S'_{en} = S_{2P} \cup S_{3P}$$

where

$$\begin{aligned} S_{2P} =& \Sigma^* V_{10}\{FN\{\Sigma^* V_{51}\{\Sigma^*\}\Sigma^*\}LN\{\Sigma^* V_6\{\Sigma^*\}\Sigma^*\}\}\Sigma^* \\ S_{3P} =& \Sigma^* V_{11}\{FN\{\Sigma^* V_{52}\{\Sigma^*\}\Sigma^*\}MN\{\Sigma^* V_8\{\Sigma^*\}\Sigma^*\}LN\{\Sigma^* V_6\{\Sigma^*\}\Sigma^*\}\}\Sigma^* \end{aligned}$$

After pushdown the final spanner is:

$$S_{en} = \pi_V \left(S'_{en} \bowtie \pi_{V_{51}} S_{tn} \bowtie \pi_{V_6} S_{tn} \bowtie \pi_{V_8} S_{tn} \bowtie \pi_{V_{52}} S_{tn}\right)$$

where $V = \{FN, MN, LN, V_{10}, V_{11}\}$. The final set of variables are listed in Table 4.6.

Table 4.6: Available variables while/after creating S'_{en}.

Variable	Annot./Attr./BLabel	Var.Type	Rule	Alias?	Canon.Name	Origin Phase
V_1	$Token$	addition	$---$	no	$---$	Tokenizer
V_2	$Token.orth = upperInitial$	addition	$---$	no	$---$	Tokenizer
V_3	$Token.kind = word$	addition	$---$	no	$---$	Tokenizer
V_4	$Token.orth = lowercase$	addition	$---$	no	$---$	Tokenizer
V_5	$Fname$	addition	$---$	no	$---$	tag-names
V_{51}	$Fname.rule = "Name1"$	addition	$---$	no	$---$	tag-names
V_{52}	$Fname.rule = "Name2"$	addition	$---$	no	$---$	tag-names
V_6	$Lname$	addition	$---$	no	$---$	tag-names
V_{61}	$Fname.rule = "Name1"$	addition	$---$	no	$---$	tag-names
V_{62}	$Fname.rule = "Name2"$	addition	$---$	no	$---$	tag-names
V_8	$Mname$	addition	$---$	no	$---$	tag-names
V_{82}	$Mname.rule = "Name2"$	addition	$---$	no	$---$	tag-names
V_7	$---$	cover	*Name1*	no	$---$	tag-names
V_9	$---$	cover	*Name2*	no	$---$	tag-names
V_{51}	$---$	reference	$---$	no	$---$	extract-names
V_6	$---$	reference	$---$	no	$---$	extract-names
V_{52}	$---$	reference	$---$	no	$---$	extract-names
V_8	$---$	reference	$---$	no	$---$	extract-names
FN	FN	extraction	$---$	no	$---$	extract-names
MN	MN	extraction	$---$	no	$---$	extract-names
LN	LN	extraction	$---$	no	$---$	extract-names
V_{10}	$---$	cover	*Two-part-Name*	no	$---$	extract-names
V_{11}	$---$	cover	*Three-part-Name*	no	$---$	extract-names

4.3 Characterization of Stable Information Extraction Programs

In this section, we define stable information extraction programs. We show that if an information extraction program is stable, we can alter the document in such a way that the synthetic version of an extracted table can be extracted directly from the altered text using the same extraction program. We propose a verifier that tests sufficient conditions on core JAPE programs to determine whether the program is stable.

4.3.1 Extracted View Update Model

In addition to the extracted span relation we assume that an *extracted string relation* is produced and is available for update. An extracted string relation contains string values associated with cells in the extracted span relation . In the string relation, $\mathcal{R}$, we denote the domain of the i^{th} attribute A_i by W_i. Our update model enables the replacement of a cell's value with another valid value from the domain associated with its corresponding attribute in the extracted string relation. Therefore, we define the update function as follows. Let $\mathcal{F}$ be an indexed set of *domain preserving* functions so that $\mathcal{F} = \{f_i | f_i : W_i \rightarrow W_i\}$ where $i \in [1 \ldots \mathcal{T}]$ and W_i is the domain for attribute A_i. If $r = \langle v_1, \ldots, v_{\mathcal{T}} \rangle$ is a record in $\mathcal{R}$, we denote the corresponding record in which the j^{th} attribute is modified as $F(r, j) = \langle v'_1, \ldots, v'_{\mathcal{T}} \rangle$, where

$$v'_k = \begin{cases} f_k(v_k) & \text{if } k = j, \\ v_k & \text{otherwise.} \end{cases}$$

In our update model, it is possible to update a whole record by repeatedly updating its cells. Therefore, we extend this notation to let $F(r) = \langle f_1(v_1), \ldots, f_{\mathcal{T}}(v_{\mathcal{T}}) \rangle$.

4.3.2 Update Translation Mechanism

In order to ensure consistency between an extracted view and the source document, it is essential to have a mechanism for translating updates made to the view back to the source document, similar to relational databases. In this work, we propose a straightforward and intuitive translation mechanism: we replace the old value corresponding to the modified cell with the new value in the document.

Let $\mathcal{X}$ be a strict and computable extractor (as defined in Chapter 1), D be a document, A_j be an extracted attribute with value v_j, and $[a_j, b_j\rangle$ denote a span in D from which v_j of record r is extracted. We define $g_r(D, j)$ to be the result of substituting $f_j(v_j)$ for v_j in the span identified by $[a_j, b_j\rangle$. Formally, $g_r(D, j) = D_{[1, a_j+1\rangle} \bullet f_j(v_j) \bullet D_{[b_j, |D|+1\rangle}$.

4.3.3 Stable Extractors

Definition 17 *An information extraction algorithm $\mathcal{X}$ is* stable *if for any document D, $\forall j \in [1 \ldots \mathcal{T}]$, and $\forall r \in R$ we have*

$$\mathcal{X}(D) = R \implies \mathcal{X}(g_r(D, j)) = \{r' | r' \in R \wedge r' \neq r\} \cup F(r, j)$$

where R is the extracted string relation.

Thus, with a stable extractor, changing a value in the appropriate position in a document affects only the expected cell in the extracted table. Our extraction process operates on a document-at-a-time basis, guaranteeing that other records extracted from other documents are not affected due to the update to an individual cell. Therefore, we focus on studying possible effects on the records extracted from the source document corresponding to an updated cell.

Theorem 4 *Consider a stable extractor $\mathcal{X}$, any indexed set of domain preserving functions $\mathcal{F} = \{f_i | f_i : W_i \rightarrow W_i$, where $i \in [1 \ldots \mathcal{T}]\}$, and any document D. For all $j \in [1 \ldots \mathcal{T}]$ and $r \in R$, substituting $f_j(v_j)$ for v_j in $[a_j, b_j\rangle$ produces $D^{\mathcal{P}}_{\mathcal{F}}$ in such a way that $F(\mathcal{X}(D)) = \mathcal{X}(D^{\mathcal{P}}_{\mathcal{F}})$.*

Proof. $\mathcal{X}$ is strict, therefore all v_j in r occur in D. Being a computable extractor, $[a_j, b_j\rangle$ is known for every v_j in r. So the locations of all spans in D that need to be modified is accessible to the procedure. Finally, $\mathcal{X}$ is stable, so any substitutions corresponding to each attribute v_j affect only the j^{th} attribute in r. □

If the extractor is stable, then the program outputs a string relation that allows us to translate updates made to the extracted relation back to the source documents.

4.3.4 Verification

Given a JAPE program and a collection of domain preserving functions, the verification process determines whether the JAPE program satisfies the proposed properties, i.e., whether the extractor is stable. As was true for the previous chapter, our verification process studies sufficient conditions for a stable extractor.

When updating a cell's value in the extracted view, we replace the old value with the new value in the corresponding position of the source document. We expect the previously extracted view to remain the same after re-extraction, except for the updated cell which should contain the new value. Hence, if modifying an individual cell results in changes in the values of other cells (either within the record associated with the updated cell or other records) when the extractor is re-executed, it indicates that the extractor lacks stability.

An extractor may not be stable for any of four reasons. First, instability may arise when the extraction program itself is inherently unpredictable, resulting in random outcomes. Second, if the new value does not fall within the domain defined by the extractor for the associated attribute, instability can occur. Third, instability may arise when the updated region overlaps with other extracted or marked regions or when the updated region overlaps with its own or other marked regions' context.

In the remainder of this section, we thoroughly examine each of these reasons and express them as conditions within a Core JAPE program. Additionally, we develop a diagnostic tool capable of analyzing an extraction program to verify its stability. We denote the input Core JAPE program by $\mathcal{P}$, its spanner representation by $\mathcal{S}$, the relation generator phase by P_E and its spanner representation by S_E, the binding label of P_E corresponding to the updated cell by L_j, and its corresponding variable in S_E by V_j.

Deterministic JAPE program

Definition 18 *A JAPE phase P is* deterministic *if for any arbitrary input document annotated regions by P stay unchanged across all possible correct implementations of a JAPE processor engine.*

Example 4.3.1

Consider the following JAPE phase:

```
Phase: phase1
Input: Token
Options: control = appelt
Rule: R1
(({Token.string == "thing"}):f{Token}|
      {Token.string == "thing"}({Token}):s) -> :f.F={rule=R1},:s.S={rule=R1}
```

This phase is not deterministic. If *R1* matches a region of the input text there are always two possible annotations, one of which will be selected based on a factor that is implementation dependent.

Definition 19 *A JAPE program is* deterministic *if all of its phases are deterministic.*

A JAPE phase with either All or Brill policy is deterministic since there is no need for arbitrary choosing an alternative match. For simplicity we consider JAPE phases which deal with ambiguity based on *explicit priority declarations* or *order of rules* not deterministic.

To study whether a JAPE phase, with policies other than All or Brill, is deterministic we generalize $ambig(g, Z)$ constructed in the proof of Theorem 1 in Chapter 3. Given a spanner specified by S representing a JAPE phase, symbols $X \notin V(S)$, $Y \notin V(S)$, $Z \in V_a(S) \cup V_e(S)$, and $W_i, W_j \in V_c(S)$ we create the spanner:

$$ambig(S, Z, W_i, W_j) = \pi_{\{W,X\}}(\rho_{Z\to X, W_i \to W}(S)) \bowtie [\![\Gamma_{(X \neq Y)}]\!] \bowtie \pi_{\{W,Y\}}(\rho_{Z \to Y, W_j \to W}(S))$$

where $\Gamma_{(X \neq Y)}$ is the disjunction of the first 12 basic relationships in Table 3.1; that is, spans in $S(D)$ could include two rows that match on cover variables but do not match on Z.

Proposition 12 *If for all $Z \in V_a(S) \cup V_e(S)$ and for all $W_i, W_j \in V_c(S)$, $ambig(S, Z, W_i, W_j) = \emptyset \vee \pi_{\{W_i\}} ambig(S, Z, W_i, W_j) = \emptyset$[6], then S represents a deterministic JAPE phase.*

Proof. Similar to the proof of Theorem 1 in Section 3.1.7. □

Domain Consistency

The view update function replaces the value in a particular cell with another value chosen from the domain of its associated attribute. Generally, the set of possible values for an extracted binding label implicitly is defined by the extraction program. We require that, for each domain-preserving function in $\mathcal{F}$, the domain of f_j is a subset of the domain formed by the rule constraints associated with the binding variable L_j in P_E. Specifically, in a JAPE rule a *ConstraintsGroup* of a *ComplexPatternElements* is bound to L_j which determines the possible values for L_j. However, we might have multiple instances of such *ConstraintsGroup* in P_E, since multiple rules in P_E might output the same label. We require that all instances of *ConstraintsGroups* specify the same set of possible values or simply define the same domain, so that changing the value cannot affect which rule is matched. By representing a JAPE program as a spanner, we express *ConstraintsGroup* using regular expressions over input characters. In other words, all regular languages that are designated as L_j must be identical. Algorithm 10 determines whether *ConstraintsGroups* associated with L_j all specify an identical domain.

Non-Conflicting Extractor

We need to find whether re-running a JAPE program over updated document, $g_r(D, j)$, for all $j \in [1, \dots, \mathcal{T}]$, extracts *correctly modified* records. Unwanted side-effects include extracting an unexpected value for the updated cell and extracting a modified value for

[6]Testing instead for $\pi_{\{W_j\}} ambig(S, Z, W_i, W_j) = \emptyset$ is equally valid.

Algorithm 10 Algorithm for Domain Consistency Test.

```
    Input: Relation generator $P_E$
    Output: Boolean
    Precondition: $\mathcal{P}$ is deterministic
 1: $[\![S_E]\!] \leftarrow toSpanner(P_E)$                          ▷ use Algorithm 9
 2: for all $V_j \in V_e([\![S_E]\!])$ do
 3:     $[\![S']\!] \leftarrow \pi_{V_j}[\![S_E]\!]$
 4:     $rgxL \leftarrow emptyList()$
 5:     for all $p \in \Pi(S')$ do                                   ▷ see Section 4.2.1
 6:         $rgxL.add(getEnclosedRegEx(p, V_j))$   ▷ retrieve regular expression enclosed by
    $V_j$ in $p$
 7:     end for
 8:     for all $r \in rgxL$ do
 9:         for all $r' \in rgxL$ do
10:             if $r \setminus r' \neq \emptyset$ then
11:                 return $False$
12:             end if
13:         end for
14:     end for
15: end for
16: return $True$
```

any other cell in the extracted relation including gaining or losing rows in the table. The reason that such side-effects might occur is overlaps between extracted spans of the input text with other determining pieces of the text. In this section, we categorize the problematic overlaps and propose a mechanism to verify a sufficient condition for a given JAPE program to be conflict-free.

Assume the entire program has been converted to its spanner representation. Given a JAPE program, let P_E represent the relation generator phase and S_E represent its equivalent spanner. The final spanner S_E has various categories of variables:

1. The set of variables associated with extracted labels, denoted by $V_e(S_E)$. In the example on page 99, $V_e(S_E) = \{FN, MN, LN\}$.
2. For each column C_j in the extracted relation and its corresponding variable in S_E, V_j, the set of all cover variables for rules that extract C_j, denoted by $V_c(S_E, V_j)$. In the example on page 99, $V_c(S_E, MN) = \{V_{11}\}$. Note that in spanner S_E each cover variable in $V_c(S_E, V_j)$ encloses one V_j.
3. The set of cover variables that are created to mark the cover of rules in the extraction phase P_E denoted by $V_c(S_E) = \{V_c(S_E, V_j) | V_j \in V_e(S_E)\}$; In the example on page 99, $V_c(S_E) = \{V_{10}, V_{11}\}$.
4. The set of all cover and extraction variables that are created for P_E when transforming P_E to S_E, denoted by $V(S_E) = V_c(S_E) \cup V_e(S_E)$; In the example on page 99, $V(S_E) = \{FN, MN, LN, V_{10}, V_{11}\}$.

Two classes of conflicts are defined based on the column that is affected by them.

Local Conflicts: Local conflicts occur when, within a given column, C_j, of the extracted relation, there are two distinct cells, c and c', in which the spans associated with their values overlap but are not equal, are equal, or are disjoint but c overlaps the cover of a rule in P_E that is marked c'.

We construct three spanners to examine whether a deterministic and domain consistent JAPE program might have local conflicts. For all three spanners we first remove records that have null for V_j:

$$\bar{S} = \pi_{V_j} S_E \bowtie S_E$$

Case I

Sometimes an extracted cell shares some data in the underlying document with another cell from the same column. Consider the following very simple relation generator with two rules that aim to extract names that comprise two or three words.

Example 4.3.2

```
Phase: RelationGenerator
Input: Token
Options: control = Brill
Rule: Two-Part
({Token.orth = upperInitial}{Token.orth = upperInitial}):name -> makeRow("name")
Rule: Three-Part
({Token.orth = upperInitial}{Token.orth = upperInitial}
      {Token.orth = upperInitial}):name -> makeRow("name")
```

Running the extractor over `Ken Michael Johnson conducted the research study.` results in a relation with overlapping spans in the column *name*.

Given an extraction variable V_j, we construct $CaseI(S_E, V_j)$:

$$CaseI(S_E, V_j) = \pi_X(\rho_{V_j \to X}\bar{S}) \bowtie [\![\Gamma_{(X \circledcirc Y)}]\!] \bowtie \pi_Y(\rho_{V_j \to Y}\bar{S})$$

where $\Gamma_{(X \circledcirc Y)}$ is the disjunction of the fifth through the twelfth basic relationships in Table 3.1.

Lemma 9 *Given a deteministic JAPE program with consistent domain and $V_j \in V_e(S_E)$ if $CaseI(S_E, V_j) = \emptyset$ then spans extracted as V_j are non-overlapping unless they are equal.*

Proof. Similar to the proof of Theorem 1 in Section 3.1.7. □

Case II A value might appear in more that one record as V_j. Consider the following example:

Example 4.3.3

```
Phase: RelationGenerator
Input: Token
Options: control = Brill
Rule: Two-Part
({Token.orth = upperInitial}{Token.orth = upperInitial}):name -> makeRow("name")
Rule: name-lastname
({Token.orth = upperInitial}{Token.orth = upperInitial}):name
      ({Token.orth = upperInitial}):lastname -> makeRow("name","lastname")
```

Running the extractor over `Ken Michael Johnson conducted the research study.` results in a relation with two rows in which `Ken Michael` shows up in the column *name*, once with null and once with `Johnson` under lastname. We construct the following spanners to detect similar situations:

$$CaseII(S_E, V_j, Z) = \pi_{X,V_j}(\rho_{Z\to X}\bar{S}) \bowtie [\![\Gamma_{(X \neq Y)}]\!] \bowtie \pi_{Y,V_j}(\rho_{Z\to Y}\bar{S})$$

$$C^{\emptyset}(S_E, V_j, Z) = \pi_{X,V_j}(\rho_{Z\to X}\bar{S}) \bowtie \pi_{Y,V_j}(\rho_{Z\to Y}\bar{S})$$

where $\Gamma_{(X\neq Y)}$ is the disjunction of the first 12 basic relationships and $Z \in V(S_E) \setminus V_j$.

Lemma 10 *Given a deteministic JAPE program with consistent domain and* $V_j \in V_e(S_E)$ *if* $\forall Z \in V(S_E) \setminus V_j$ $CaseII(S_E, V_j, Z) = \emptyset$ *and* $\left(\pi_X C^{\emptyset}(S_E, V_j, Z) \bowtie C^{\emptyset}(S_E, V_j, Z)\right) = \left(\pi_Y C^{\emptyset}(S_E, V_j, Z) \bowtie C^{\emptyset}(S_E, V_j, Z)\right)$ *then there exist no two distinct rows of the extracted relation in which the spans associated with values extracted as* V_j *are identical.*

Proof. Similar to the proof of Theorem 1 in Section 3.1.7. □

Case III

An extracted value as V_j might have problematic overlap with its own cover or the cover of other values extracted as V_j. Consider the following program with two phases that aims to extract items similar to $Q1$ of the blog corpus (Table 3.2):

Example 4.3.4

```
Phase: action-movie
Input: Token
Options: control = All
Rule: tag-action
({Token.string = "see"}|{Token.string = "See"}|{Token.string = "watch"}|
      {Token.string = "Watch"}):action -> :action.Action={}
Rule: tag-movie
{Token.string="}({Token.orth = upperInitial}|({Token.orth = upperInitial}
      {Token.orth = upperInitial})):movie {Token.string="} -> :movie.Movie={}


Phase: RelationGenerator
Input: Action Movie
Options: control = All
Rule: movie-review
({Action}):ac({Movie}):mv -> makeRow("ac","mv")
```

Running the extractor on `I might see "The Watch" and "Inception" today!!` the span for `Watch` is marked as *ac* in one record and overlaps with the cover of the span for `see` marked as *ac* in a different record. Let S' represent a span relation in which each row corresponds to two firings of the rules extracting disjoint spans as V_j:

$$S' = (\rho_{V_j \to X} \pi_{V_j} \bar{S}) \bowtie [\![\Gamma_{(X\ dis\ Y)}]\!] \bowtie (\rho_{V_j \to Y} \bar{S})$$

where $\Gamma_{(X\ dis\ Y)}$ is the disjunction of the first four basic relationships in Table 3.13.1. The following spanner finds any overlap such as the one in Example 4.3.4:

$$CaseIII(S_E, V_j, Q) = \pi_X S' \bowtie [\![\Gamma_{(X \pitchfork Y)}]\!] \bowtie (\rho_{Q \to Y}(\rho_{Y \to W} S')).$$

$\Gamma_{(X \pitchfork Y)}$ is the disjunction of the fifth through thirteenth basic relationships in Table 3.13.1 and $Q \in V_c(S_E, V_j)$.

Lemma 11 *For a given deterministic and domain consistent JAPE program, if* $\forall Q \in V_c(S_E, V_j)$, $CaseIII(S_E, V_j, Q) = \emptyset$, *then there is no problematic overlap between the spans extracted as* V_j *and covers associated with* V_j.

Proof. Similar to the proof of Theorem 1 in Section 3.1.7. □

Corollary 2 *An input JAPE program is free of the local conflicts if* $\forall V_j \in V_e(S_E)$, $\forall Z \in V(S_E)\backslash V_j$, *and* $\forall Q \in V_c(S_E, V_j)$ $CaseI(S_E, V_j) = CaseII(S_E, V_j, Z) = CaseIII(S_E, V_j, Q) = \emptyset$ *and* $\big(\pi_X C^{\emptyset}(S_E, V_j, Z) \bowtie C^{\emptyset}(S_E, V_j, Z)\big) = \big(\pi_Y C^{\emptyset}(S_E, V_j, Z) \bowtie C^{\emptyset}(S_E, V_j, Z)\big)$.

Proof. The proof can be derived according to Lemmas 9, 10, and 11. □

Global Conflicts: Global conflicts occur when the value of a cell, c, in the extracted relation overlaps or is equal to the underlying text that is consumed by a rule in P_E except for the rules that mark the column associated with c.

We construct the following spanner:

$$conflict(S_E, V_j, V) = \pi_X(\rho_{V_j \to X}\bar{S}) \bowtie [\![\Gamma_{(X \pitchfork Y)}]\!] \bowtie \pi_Y(\rho_{V \to Y}\bar{S})$$

where $V \in V_c(S_E) \setminus V_c(S_E, V_j)$ and $\bar{S}$ is the same as defined for the local conflicts.

Lemma 12 *If* $\forall V \in V_c(S_E) \setminus V_c(S_E, V_j)$, $conflict(S_E, V_j, V) = \emptyset$ *then the input program does not have any global conflicts w.r.t* V_j.

Proof. Similar to the proof of Theorem 1 in Section 3.1.7. □

Corollary 3 *A JAPE program is global conflict-free if* $\forall V_j \in V_e(S_E)$ *and* $\forall V \in V_c(S_E) \setminus V_c(S_E, V_j)$, $conflict(S_E, V_j, V) = \emptyset$.

Proof. The proof can be derived according to Lemma 12. □

Theorem 5 *If a deterministic JAPE program* $\mathcal{P}$ *with a consistent domain is free of local and global conflicts, then* $\mathcal{P}$ *is stable.*

Proof. The proof is by contradiction. Assume that the value of a cell, c, within column C_j and row r is updated from v to v' and the source document is updated accordingly. Let R denote the rule responsible for extracting v. Assume that $\mathcal{P}$ is not stable. If we execute the extraction program over $g_r(D, j)$ two cases might occur:

1. r has changed in an unexpected manner. Unexpected effects might be one or more of the following possibilities:

   I) At least one cell other than c, say c', has changed in r. This implies that update to c has modified the context or value of c' both of which are in the cover of C_j. But $\mathcal{P}$ is free of local conflicts ($CaseIII$). This case is not possible.

   II) c has a value other than v'. Since v' has the same domain as v and $\mathcal{P}$ has a consistent domain, the match associated with r before the update needs to match after the update. This case is not possible.

   III) r has disappeared. Possibilities are: 1) The cover of R does not match the region when v' is in place of v, but as in Case 1.II, having a consistent domain makes this impossible. 2) some other match takes precedence. Since $\mathcal{P}$ is deterministic it must have been the policy causing this, but the priority of the match could not have changed after the update. Because the update stays in the domain, $\mathcal{P}$ has a consistent domain, and $\mathcal{P}$ is free of local and global conflicts, the update could not have changed the extent of other matches in D.

2. A row other than r, say r', has changed after the update. There are three possibilities:

   (a) r' is a new row that has appeared as a result of the update. There are two possible causes:

   I) The match corresponding to r' existed in D but was ignored. As for Case 1.III.2 the priority of matches cannot have changed, so this is impossible.

   II) The value of one or more extracted cells in r' or the underlying value of the cells' covers essential in forming the match of r' did not form a match in D but does form one in $g_r(D, j)$. Because the extractor is free of local and global conflicts v' could not have modified the values of those underlying values, thus this scenario is not possible.

   (b) r' is a row that has disappeared due to the update:

   I) The match corresponding to r' exists in $g_r(D, j)$ but is ignored. A similar argument to Case 1.III.2 can be made to discard this case. II) The value of one or more extracted cells in r', the underlying value of the cells' covers, essential in forming a match of r' did form a match in D but does not form one in $g_r(D, j)$. A similar argument to Case 2a.II can be made to rule out this possibility.

   (c) r' is a row that has at least one extracted value changed due to the update. Because the extractor is free of local and global conflicts v' could not have modified the extracted values.

□

The complete verification process including JAPE to Spanner conversion is shown in Algorithm 11 and Figure 4.8.

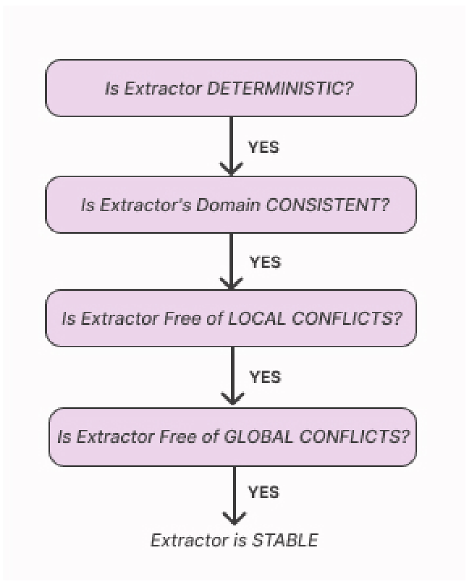
Is Extractor DETERMINISTIC?
YES
Is Extractor's Domain CONSISTENT?
YES
Is Extractor Free of LOCAL CONFLICTS?
YES
Is Extractor Free of GLOBAL CONFLICTS?
YES
Extractor is STABLE

Figure 4.8: The proposed verification process for updatable extracted views is realized through four distinct tests on the extractor's spanner representation.

4.4 Implementation Details

We developed a verification system, named U2V (**U**pdate-**to**-**V**iew), mainly to ensure the correctness of the proposed algorithms. Our validation process involves two key factors: I) annotated regions: For each input document, the set of annotated regions within the input text must be a subset of the regions annotated by the corresponding spanner representations; II) stability: the verifier must identify all unstable programs as unstable.

U2V comprises three main components:

- **Convertor**: The convertor converts an input program, written in Core JAPE, into its spanner representation. Each program consists of multiple files, including a tokenizer, gazetteer's files, and a file containing user-defined phases. The convertor generates an

Algorithm 11 Algorithm to determine Updatablity of extracted view

```
    Input: Core JAPE Program $[P_1, \cdots, P_n]$, Tokenizer $T$, Gazetteer $G$, relation gen. $P_E$
    Output: Boolean
    Precondition: JAPE program is deterministic with consistent domain
 1: $S_G \leftarrow GzToSpn(G)$                                  ▷ convert gazetteer to spanner (Alg. 8)
 2: $S_T \leftarrow TkrToSpn(T)$                                 ▷ convert tokenizer to spanner (Alg. 8)
 3: $S \leftarrow list()$
 4: for all $P_i \in [P_1, \cdots, P_n]$ do
 5:     $S.append(phToSpn(P_i, S, S_G, S_T))$                     ▷ convert phase to spanner (Alg. 9)
 6: end for
 7: $S_E \leftarrow relGenToSpn(P_E, S, S_G, S_T)$              ▷ convert relation gen to spanner (Alg. 9)
 8: for all $V_j \in V_e(S_E)$ do
 9:     if $CaseI(S_E, V_j) \neq \emptyset$ then
10:         return False
11:     end if
12:     for all $Z \in V(S_E) \setminus V_j$ do
13:         if $CaseII(S_E, V_j, Z) \neq \emptyset$ then
14:             return False
15:         end if
16:         if $\left(\pi_X C^{\emptyset}(S_E, V_j, Z) \bowtie C^{\emptyset}(S_E, V_j, Z)\right) \neq$
17:           $\left(\pi_Y C^{\emptyset}(S_E, V_j, Z) \bowtie C^{\emptyset}(S_E, V_j, Z)\right)$ then
18:             return False
19:         end if
20:     end for
21:     for all $Q \in V_c(S_E, V_j)$ do
22:         if $CaseIII(S_E, V_j, Q) \neq \emptyset$ then
23:             return False
24:         end if
25:     end for
26: end for                                                         ▷ free of local conflict
27: for all $V_j \in V_e(S_E)$ do
28:     for all $V \in V_c(S_E) \setminus V_c(S_E, V_j)$ do
29:         if $conflict(S_E, V_j, V) \neq \emptyset$ then
30:             return False
31:         end if
32:     end for
33: end for                                                         ▷ free of global conflict
34: return True                                                     ▷ program is updatable
```

individual spanner for each phase, tokenizer, and gazetteer, as well as one spanner for the entire program. Furthermore, the convertor is responsible for creating, updating, and persisting the global tables, i.e. MTable and VTable.

- **Operation Engine**: The operation engine performs all spanner algebraic operators used by the convertor and verifier. The implementation is based on Marciano's engine [72] engine. Because that engine supports operators on functional spanners, and our task involves partially functional spanners, we have added an additional layer on top to make the necessary adjustments.
- **Verifier Engine**: The verifier determines the stability of the input JAPE program through four tests (Section 4.3.4). For each test the verifier relies on relational operators provided by the operation engine as well as spanner representations provided by the convertor.

4.4.1 Platform

The verifier is a single-threaded program written in Scala 2.11.2 together with Java SE 11. All experiments are performed on an AMD EPYC 7502P 32-Core Processor under Ubuntu 20.04.1 LTS (Focal Fossa). The source code, all performance details, and data can be found in the project's Git repository.[7]

[7]https://git.uwaterloo.ca/bkassaie/updatableviews

Chapter 5

Conclusions and Future Work

5.1 Summary

We have introduced two enhancements to extraction systems: 1) extracted relations should be considered as materialized views over documents; 2) updating source documents and extracted views should be viewed as an integral part of any extraction system (see Figure 1.3). Within this context, we have tackled two research challenges.

1. Efficiently maintaining materialized extracted views (Chapter 3):

   Given a program defined as a document spanner and an update specification, we determine sufficient conditions for autonomously re-computing which spans of an updated document are extracted. In particular, we propose three sufficient conditions for shiftability of updates with respect to an extraction program, namely, *durability*, *independence*, and *respect for alphabets*. We prove that we require time and space that are polynomial in the size of the extraction program and the update specification to perform the five required tests (Figure 3.6) to determine that the revised extracted relation can be computed autonomously.

   The sufficient conditions introduced for this problem cover intuitive and natural cases. For instance, *durability* is likely a practical condition, but there could be an update that modifies its own context and can be proven to be shiftable.

   We also designed some practical extractors and conducted experiments on two real-world datasets to conclude that the runtime overhead imposed by our verification is small in practice when compared to re-evaluating extractors, even if the re-evaluation is performed incrementally. Furthermore, because it uses static analysis, verification is independent of database size.

2. Translating updates to extracted views into updates to documents (Chapter 4):

   We propose a conversion mechanism that constructs a spanner representation for a given program, written in a significant subset of the JAPE language. We characterize extraction algorithms that are resilient to changes in their source documents intended to reflect predetermined changes to the extracted table, i.e., *stable* extractors. We further propose a straightforward algorithm that modifies the input document considering the stable extractor and a set of domain preserving functions. The modified document can be fed into the extractor to produce the expected updated table. Moreover, we propose a verification process to ensure the stability of programs written in the JAPE language. The verifier first converts the input JAPE program into its spanner representation and then tests four sufficient conditions (Figure 4.8). The sufficient conditions are most likely to hold in practice. However, a key limitation of our method is that we have not incorporated phase policies into the spanner representation. This could lead to the misclassification of numerous stable extractors as unstable.

5.2 Implications and Impacts

The problems addressed in this thesis can be formulated for alternative extraction settings such as those based on other declarative or imperative languages such as Datalog or JAVA, or based on machine learning models. The approach we have taken involves understanding and analyzing the extractors. Having tried ideas in two separate extraction frameworks, we believe that similar mechanisms can be adopted to other extraction systems based on regular languages and their extensions. However, devising a similar approach to other settings requires a level of understanding of the extraction mechanisms. For example, if an extractor is expressed as a deep learning model, research questions such as "what information can be gained from the model to perform similar static analyses?" or "can we use some information from the training step to build another model capable of deciding whether we need to re-extract after an update to the source documents?" might arise.

One crucial aspect of Core AQL queries that enabled our approach is the composability of Core AQL programs. Specifically, shiftability with respect to a given update can be inherited from basic queries expressed as regular expressions to complex queries. We believe that this property could prove valuable in devising solutions for other settings as well. By leveraging composability, complex extraction mechanisms can be broken down into smaller pieces and their outcomes can be propagated to the whole system.

Additionally, our work makes a significant contribution by explicitly establishing the connection between updates in the information extraction setting and view update and view maintenance in the relational setting. We firmly believe that this connection will lead to the formulation and resolution of numerous intriguing problems, ultimately having a meaningful impact on text processing and understanding. For example, text summarization, text translation, image captioning and other applications of NLP may also be able to benefit from treating the output as materialized views. By bridging these two domains, we open up new avenues for research and development, fostering advancements that can revolutionize how we handle and interpret textual data.

5.3 Future Work

For the extracted view maintenance problem, we have established some sufficient conditions for updates to be shiftable with regard to extractors expressed in Core AQL, but we have not yet investigated whether there are necessary conditions nor whether there are additional sufficient conditions that would be especially useful in practice. For example, we have already seen that if the string in an extracted span is replaced by a string of the same length and it is also extractable, the update is not only shiftable, but also irrelevant. Furthermore, we have not yet investigated other autonomously computable conditions, such as those that might result in span modifications or insertions of extracted tuples. AQL is a broad language with many more operators and features beyond what is covered in this work.

Our model for document updates is also quite limited. First of all, only one variable is used to identify spans that can be updated, even though correlated updates might require multiple related variables to update. Secondly, the substitute value is limited to being constructed based on the substring matched by the update variable, whereas real world applications might need to use various values based on some factors, such as the relative position of the update, some associated string values, or the contexts of matched spans. Thirdly, for each document, all intended spans are updated once and simultaneously, a fundamental assumption that can be violated in practical situations. Loosening any of these restrictions creates new research challenges for verifying shiftability or other update properties.

In this work, we assume that specific constraints on documents are unspecified. However, a document may be required to conform to a schema involving elements such as title and abstract. Given an extractor expressed as a regex, E, and a document update, g expressed in $G_S(\Sigma, V, x)$, if the document constraints are provided as a regular expression R_D (i.e., $D \in L(R_D) \subset \Sigma^*$), our view maintenance algorithms and results remain valid even if we substitute E and g with $E \bowtie R_D$ and $g \bowtie R_D$, respectively, throughout.

For updatable extracted views, we have made some simplifying assumptions in our work, each of which can be modified or eliminated to expand the class of programs deemed to be stable. For example, we have assumed independence between extracted attributes, thus requiring that at most one extracted value can be affected by each change in the source document. What if instead we are given constraints among the attributes, such as A_2 and A_5 must be identical or must have (computably) dependent values? We have also assumed that each table attribute can be given a value from a single span in the input document. What if several words or phrases from multiple places can be combined to create an extracted value? Loosening our simplifying assumptions might result in being able to verify more useful extractors.

We have presented a property verification process applicable to programs expressed by a subset of JAPE. We have designed the complete verification process and proved its correctness. However, we have not addressed other JAPE capabilities, such as allowing annotations to be removed from the graph or using Java code to describe a rule's actions (which, in general, will make stability undecidable). Furthermore, when creating the corresponding spanner representation, we did not take into account the phase policy, potentially leading to the misidentification of a stable program as unstable. We plan to incorporate the policy into the spanner representation to enhance the accuracy of identifying stable programs.

We have developed a set of sufficient properties for stable JAPE programs, but again we have not investigated which properties might be necessary. We might also wish to explore whether some programs that cannot be verified as stable can be transformed into ones that possess the required properties of stability.

A natural extension for both problems involves formulating each problem in the other language; specifically, maintaining extracted views for JAPE programs and ensuring the updatability of extracted views for AQL. We wish to enrich both verifiers with the capabilities of providing developers with insights for designing autonomously computable and stable extractors. Finally, we are interested in developing verification tools (for both problems) for other forms of extractors, including those based on machine learning technology.

5.3.1 Potential of Derivatives of Regular Expressions

Our approach to extracted view maintenance works on a restricted set of document spanners that satisfy certain sufficient conditions. To verify other sufficient or necessary conditions one probably will need to deal with a variety of Boolean combinations of regular expression constraints such as complement and intersection. These combinations in the worst case end up constructing automata that have exponential space blowup. To avoid such a blowup, Brzozowski [15] avoids building up the whole state space at once. Therefore, a constraint such as:

$$S \in R$$

where R is a regular expression and $S = s_0 s_1 \cdots s_n$ is a concrete string $S \in \Sigma^*$ is reduced to:

$$s_1 s_2 \cdots s_n \in D_{s_0}(R)$$

where $D_a(R)$ is the derivative of the regular expression with respect to a character a. $D_a(R)$ returns a regular expression representing the suffix of R. With this approach, only the states that are needed are explored, as opposed to the conventional technique that builds the complete state upfront.

In practice, regular expressions used as extractors have a large alphabet, the natural language alphabet, which requires some considerations. In recent work, Stanford et al. [87] deal with the regular expression's derivative symbolically to solve regular expression constraints such as:

$$s \in R$$

where s is an uninterpreted string and R is a boolean combination of regular expression constraints. For example, R might be a constraint on the data value:

$$\texttt{date} \in \texttt{\textbackslash d}\{4\}\texttt{-}[a - zA - Z]\{3\}\texttt{-\textbackslash d}\{2\} \cap (\texttt{date} \in 2019\Sigma^* \cup \texttt{date} \in 2020\Sigma^*)$$

and we wish to know whether there is at least one string that conforms to the constraint [87]: $R \neq \emptyset$. To date, this construction of derivatives is the most efficient approach to solving complex regular expression constraints that might be created for extended shiftability verification or view updatability.

References

[1] dblp: Statistics: https://dblp.org/statistics/index.html.

[2] The dblp team: dblp computer science bibliography. Monthly snapshot release of October 2022 https://dblp.org/xml/release/dblp-2022-10-02.xml.gz.

[3] The dblp team: the sources for the org.dblp.mmdb package https://dblp.org/src/mmdb-2019-04-29-sources.jar.

[4] Insider Inc. https://www.businessinsider.com. Accessed: 2023-11-02.

[5] Introducing ChatGPT. https://openai.com/blog/chatgpt. Accessed: 2023-11-02.

[6] Serge Abiteboul, Jason McHugh, Michael Rys, Vasilis Vassalos, and Janet L. Wiener. Incremental maintenance for materialized views over semistructured data. In Ashish Gupta, Oded Shmueli, and Jennifer Widom, editors, *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 38–49. Morgan Kaufmann, 1998.

[7] Monica Agrawal, Stefan Hegselmann, Hunter Lang, Yoon Kim, and David A. Sontag. Large language models are few-shot clinical information extractors. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pages 1998–2022. Association for Computational Linguistics, 2022.

[8] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[9] Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. Constant-delay enumeration for nondeterministic document spanners. In Pablo Barceló and

Marco Calautti, editors, *22nd International Conference on Database Theory, ICDT 2019, March 26-28, 2019, Lisbon, Portugal*, volume 127 of *LIPIcs*, pages 22:1–22:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[10] Douglas E. Appelt and Boyan A. Onyshkevych. The common pattern specification language. In *TIPSTER TEXT PROGRAM PHASE III: Proceedings of a Workshop held at Baltimore, MD, USA, October 13-15, 1998*, Baltimore MD USA, 1998.

[11] Philip Bille and Inge Li Gørtz. From regular expression matching to parsing. *Acta Informatica*, pages 1–16, 2022.

[12] José A. Blakeley, Neil Coburn, and Per-Åke Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Transactions on Database Systems, TODS*, 14(3):369–400, 1989.

[13] José A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. Efficiently updating materialized views. In Ashish Gupta and Iderpal Singh Mumick, editors, *Materialized Views: Techniques, Implementations, and Applications*, pages 163–175. MIT Press, Cambridge MA USA, 1999. (reprinted from *ACM Sigmod '86*, pp. 61-71).

[14] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.

[15] Janusz A Brzozowski. Derivatives of regular expressions. *Journal of the ACM (JACM)*, 11(4):481–494, 1964.

[16] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott M. Lundberg, Harsha Nori, Hamid Palangi, Marco Túlio Ribeiro, and Yi Zhang. Sparks of artificial general intelligence: Early experiments with GPT-4. *CoRR*, abs/2303.12712, 2023.

[17] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Why and where: A characterization of data provenance. In *Proceedings of 4th Int. Conference on Database Theory (ICDT 2001)*, pages 316–330, 2001.

[18] Pere-Lluís Huguet Cabot and Roberto Navigli. REBEL: relation extraction by end-to-end language generation. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Findings of the Association for Computational Linguistics: EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 16-20 November, 2021*, pages 2370–2381. Association for Computational Linguistics, 2021.

[19] Mary Elaine Califf and Raymond J. Mooney. Relational learning of pattern-match rules for information extraction. In Jim Hendler and Devika Subramanian, editors, *Proceedings of the Sixteenth National Conference on Artificial Intelligence and Eleventh Conference on Innovative Applications of Artificial Intelligence, July 18-22, 1999, Orlando, Florida, USA*, pages 328–334. AAAI Press / The MIT Press, 1999.

[20] Xiaoyong Chai, Ba-Quy Vuong, AnHai Doan, and Jeffrey F. Naughton. Efficiently incorporating user feedback into information extraction and integration programs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 87–100, Rhode Island USA, 2009. ACM.

[21] Amit Chandel, P. C. Nagesh, and Sunita Sarawagi. Efficient batch top-k search for dictionary-based entity recognition. In Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang, editors, *Proceedings of 22nd International Conference on Data Engineering, ICDE*, pages 28:1–28:10, Atlanta, April 2006. IEEE Computer Society.

[22] Fei Chen, AnHai Doan, Jun Yang, and Raghu Ramakrishnan. Efficient information extraction over evolving text data. In *Proceedings of the 24th International Conference on Data Engineering, ICDE*, pages 943–952, Cancún, Mexico, 2008. IEEE Computer Society.

[23] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage,

Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

[24] James Cheney, Laura Chiticariu, and Wang Chiew Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[25] Rada Chirkova, Jun Yang, et al. Materialized views. *Foundations and Trends® in Databases*, 4(4):295–405, 2012.

[26] Laura Chiticariu, Yunyao Li, and Frederick R. Reiss. Rule-based information extraction is dead! long live rule-based information extraction systems! In *Proceedings of 2013 Conference on Empirical Methods in Natural Language Processing. (EMNLP 2013)*, pages 827–832, 2013.

[27] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[28] Hamish Cunningham, Diana Maynard, Kalina Bontcheva, Valentin Tablan, Niraj Aswani, Ian Roberts, Genevieve Gorrell, Adam Funk, Angus Roberts, Danica Damljanovic, et al. Developing language processing components with gate version 8. *University of Sheffield Department of Computer Science*, 2014.

[29] Hamish Cunningham, Valentin Tablan, and Diana Maynard. JAPE: a Java annotation patterns engine. *Research Memorandum CS–00–10, Department of Computer Science, University of Sheffield*, 43, 2000.

[30] Umeshwar Dayal and Philip A. Bernstein. On the updatability of relational views. In *Fourth International Conference on Very Large Data Bases*, pages 368–377, West Berlin Germany, 1978. IEEE Computer Society.

[31] Umeshwar Dayal and Philip A. Bernstein. On the correct translation of update operations on relational views. *ACM Transactions Database Systems*, 7(3):381–416, 1982.

[32] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA,*

June 2-7, 2019, Volume 1 (Long and Short Papers), pages 4171–4186. Association for Computational Linguistics, 2019.

[33] Thomas G. Dietterich. What's wrong with llms and what we should be building instead. ValgrAI Scientific Council Forum, 2023.

[34] Johannes Doleschal, Benny Kimelfeld, Wim Martens, Yoav Nahshon, and Frank Neven. Split-correctness in information extraction. In Dan Suciu, Sebastian Skritek, and Christoph Koch, editors, *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 149–163. ACM, 2019.

[35] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014.

[36] Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. Cleaning inconsistencies in information extraction via prioritized repairs. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 164–175, Snowbird UT USA, 2014. ACM.

[37] Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. Document spanners: A formal approach to information extraction. *Journal of the ACM*, 62(2):12:1–12:51, 2015.

[38] Fernando Florenzano, Cristian Riveros, Martín Ugarte, Stijn Vansummeren, and Domagoj Vrgoc. Efficient enumeration algorithms for regular document spanners. *ACM Transactions Database Systems*, 45(1):3:1–3:42, 2020.

[39] Dominik D. Freydenberger and Sam M. Thompson. Dynamic complexity of document spanners. In Carsten Lutz and Jean Christoph Jung, editors, *23rd International Conference on Database Theory, ICDT 2020, March 30-April 2, 2020, Copenhagen, Denmark*, volume 155 of *LIPIcs*, pages 11:1–11:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[40] Antonio L. Furtado and Marco A. Casanova. Updating relational views. In *Query Processing in Database Systems*, pages 127–142. Springer, 1985.

[41] Antonio L. Furtado, Kenneth C. Sevcik, and Clesio Saraiva dos Santos. Permitting updates through views of data bases. *Information Systems*, 4(4):269–283, 1979.

[42] Robert Gaizauskas and Yorick Wilks. Information extraction: Beyond document retrieval. *Journal of Documentation*, 54(1):70–105, 1998.

[43] Robert J. Gaizauskas, Hamish Cunningham, Yorick Wilks, Peter J. Rodgers, and Kevin Humphreys. GATE: an environment to support research and development in natural language engineering. In *Eigth International Conference on Tools with Artificial Intelligence, ICTAI '96, Toulouse, France, November 16-19, 1996*, pages 58–66. IEEE Computer Society, 1996.

[44] Ralph Grishman and Beth Sundheim. Message understanding conference- 6: A brief history. In *Proceedings of 16th International Conference on Computational Linguistics (COLING 1996)*, pages 466–471, 1996.

[45] Ashish Gupta, H. V. Jagadish, and Inderpal Singh Mumick. Data integration using self-maintainable views. In Peter M. G. Apers, Mokrane Bouzeghoub, and Georges Gardarin, editors, *Proceedings of Advances in Database Technology - EDBT'96, 5th International Conference on Extending Database Technology*, volume 1057 of *Lecture Notes in Computer Science*, pages 140–144, Avignon, March 1996. Springer.

[46] Ashish Gupta and Iderpal Singh Mumick, editors. *Materialized Views: Techniques, Implementations, and Applications.* MIT Press, Cambridge, MA, USA, 1999.

[47] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[48] Ihab F. Ilyas and Xu Chu. *Data Cleaning.* Morgan and Claypool, 2019.

[49] Panagiotis G. Ipeirotis, Eugene Agichtein, Pranay Jain, and Luis Gravano. Towards a query optimizer for text-centric tasks. *ACM Transactions Database Systems*, 32(4):21:1–21:46, 2007.

[50] Hideki Isozaki and Hideto Kazawa. Efficient support vector classifiers for named entity recognition. In *19th International Conference on Computational Linguistics, COLING 2002, Howard International House and Academia Sinica, Taipei, Taiwan, August 24 - September 1, 2002*, 2002.

[51] Alpa Jain, Panagiotis G. Ipeirotis, and Luis Gravano. Building query optimizers for information extraction: the sqout project. *SIGMOD Record.*, 37(4):28–34, 2008.

[52] Martin Josifoski, Marija Sakota, Maxime Peyrard, and Robert West. Exploiting asymmetry for synthetic training data generation: Synthie and the case of information extraction. *CoRR*, abs/2303.04132, 2023.

[53] Shiva Prasad Kasiviswanathan, Homin K. Lee, Kobbi Nissim, Sofya Raskhodnikova, and Adam D. Smith. What can we learn privately? *SIAM Journal on Computing*, 40(3):793–826, 2011.

[54] Besat Kassaie and Frank Wm. Tompa. Predictable and consistent information extraction. In *Proceedings of the ACM Symposium on Document Engineering*, pages 14:1–14:10, Berlin Germany, 2019. ACM.

[55] Besat Kassaie and Frank Wm. Tompa. A framework for extracted view maintenance. In *DocEng '20: ACM Symposium on Document Engineering 2020, Virtual Event, CA, USA, September 29 - October 1, 2020*, pages 16:1–16:4. ACM, 2020.

[56] Besat Kassaie and Frank Wm. Tompa. Autonomously computable information extraction. *Proceedings of the VLDB Endowment*, 16(10):2431–2443, Aug 2023.

[57] Arthur M. Keller. The role of semantics in translating view updates. *Computer*, 19(1):63–73, 1986.

[58] Kenrick Kin, Björn Hartmann, Tony DeRose, and Maneesh Agrawala. Proton: multitouch gestures as regular expressions. In *ACM Conference on Human Factors in Computing Systems*, pages 2885–2894, 2012.

[59] Hanna Kozankiewicz, Jacek Leszczylowski, and Kazimierz Subieta. Updatable XML views. In Leonid A. Kalinichenko, Rainer Manthey, Bernhard Thalheim, and Uwe Wloka, editors, *Advances in Databases and Information Systems, 7th East European Conference, ADBIS 2003, Dresden, Germany, September 3-6, 2003, Proceedings*, volume 2798 of *Lecture Notes in Computer Science*, pages 381–399. Springer, 2003.

[60] John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In Carla E. Brodley and Andrea Pohoreckyj Danyluk, editors, *Proceedings of the Eighteenth International Conference on Machine Learning (ICML 2001), Williams College, Williamstown, MA, USA, June 28 - July 1, 2001*, pages 282–289. Morgan Kaufmann, 2001.

[61] Kristina Lerman, Steven Minton, and Craig A. Knoblock. Wrapper maintenance: A machine learning approach. *Journal of Artificial Intelligence Research (JAIR)*, 18:149–181, 2003.

[62] Michael Ley. DBLP - some lessons learned. *Proceedings of the VLDB Endowment*, 2(2):1493–1500, 2009.

[63] Peng Li, Tianxiang Sun, Qiong Tang, Hang Yan, Yuanbin Wu, Xuanjing Huang, and Xipeng Qiu. Codeie: Large code generation models are better few-shot information extractors. In Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 15339–15353. Association for Computational Linguistics, 2023.

[64] Y Edmund Lien. Multivalued dependencies with null values in relational data bases. In *Fifth International Conference on Very Large Data Bases, 1979.*, pages 61–66. IEEE, 1979.

[65] Peter Linz. *An introduction to formal languages and automata, 5th Edition.* Jones and Bartlett Publishers, 2012.

[66] Bin Liu, Laura Chiticariu, Vivian Chu, HV Jagadish, and Frederick R. Reiss. Automatic rule refinement for information extraction. *Proceedings of the VLDB Endowment*, 3(1-2):588–597, 2010.

[67] Yubo Ma, Yixin Cao, YongChing Hong, and Aixin Sun. Large language model is not a good few-shot information extractor, but a good reranker for hard samples! *CoRR*, abs/2303.08559, 2023.

[68] Francisco Maturana, Cristian Riveros, and Domagoj Vrgoc. Document spanners for extracting incomplete information: Expressiveness and complexity. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 125–136, Houston TX USA, 2018. ACM.

[69] Claudia Bauzer Medeiros and Frank Wm. Tompa. Understanding the implications of view update policies. *Algorithmica*, 1(3):337–360, 1986.

[70] Alexandra Meliou, Wolfgang Gatterbauer, and Dan Suciu. Reverse data management. *Proceedings of the VLDB Endowment*, 4(12):1490–1493, 2011.

[71] George A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41, 1995.

[72] Andrea Morciano. Engineering a runtime system for AQL. Master's thesis, École Polytechnique de Bruxelles, Université Libre de Bruxelles, 2016.

[73] Makoto Murata, Akihiko Tozawa, Michiharu Kudo, and Satoshi Hada. XML access control using static analysis. *ACM Transactions on Information and System Security (TISSEC)*, 9(3):292–324, 2006.

[74] Alan Nash, Luc Segoufin, and Victor Vianu. Views and queries: Determinacy and rewriting. *ACM Transactions on Database Systems*, 35(3):21:1–21:41, 2010.

[75] Thien Huu Nguyen. *Deep Learning for Information Extraction.* PhD thesis, New York University, USA, 2018.

[76] Christina Niklaus, Matthias Cetto, Andre Freitas, and Siegfried Handschuh. A survey on open information extraction. In *Proceedings of the 27th International Conference on Computational Linguistics*, 2018. 27th International Conference on Computational Linguistics, COLING 2018 ; Conference date: 20-08-2018 Through 26-08-2018.

[77] Aleksander Obuchowski, Barbara Klaudel, and Patryk Jasik. Information extraction from polish radiology reports using language models. In *Proceedings of the 9th Workshop on Slavic Natural Language Processing 2023 (SlavicNLP 2023)*, pages 113–122, 2023.

[78] Liat Peterfreund, Dominik D. Freydenberger, Benny Kimelfeld, and Markus Kröll. Complexity bounds for relational algebra over document spanners. In Dan Suciu, Sebastian Skritek, and Christoph Koch, editors, *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 320–334. ACM, 2019.

[79] Liat Peterfreund, Balder ten Cate, Ronald Fagin, and Benny Kimelfeld. Recursive programs for document spanners. In Pablo Barceló and Marco Calautti, editors, *22nd International Conference on Database Theory, ICDT 2019, March 26-28, 2019, Lisbon, Portugal*, volume 127 of *LIPIcs*, pages 13:1–13:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[80] Jiexing Qi, Jingyao Tang, Ziwei He, Xiangpeng Wan, Yu Cheng, Chenghu Zhou, Xinbing Wang, Quanshi Zhang, and Zhouhan Lin. RASAT: integrating relational structures into pretrained seq2seq model for text-to-sql. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pages 3215–3229. Association for Computational Linguistics, 2022.

[81] Frederick Reiss, Sriram Raghavan, Rajasekar Krishnamurthy, Huaiyu Zhu, and Shivakumar Vaithyanathan. An algebraic approach to rule-based information extraction.

In *Proceedings of the 24th International Conference on Data Engineering, ICDE*, pages 933–942, Cancún, Mexico, 2008. IEEE Computer Society.

[82] Alan Ritter, Sam Clark, Mausam, and Oren Etzioni. Named entity recognition in tweets: An experimental study. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing, EMNLP*, pages 1524–1534, Edinburgh UK, 2011.

[83] Sudeepa Roy, Laura Chiticariu, Vitaly Feldman, Frederick R Reiss, and Huaiyu Zhu. Provenance-based dictionary refinement in information extraction. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pages 457–468. ACM, 2013.

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

[85] Jonathan Schler, Moshe Koppel, Shlomo Argamon, and James W. Pennebaker. Effects of age and gender on blogging. In *Proceedings of 2006 AAAI Spring Symp. on Computational Approaches to Analyzing Weblogs*, pages 199–205, 2006.

[86] Warren Shen, AnHai Doan, Jeffrey F. Naughton, and Raghu Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 1033–1044. ACM, 2007.

[87] Caleb Stanford, Margus Veanes, and Nikolaj Bjørner. Symbolic boolean derivatives for efficiently solving extended regular expression constraints. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 620–635, 2021.

[88] Danny Sullivan. A reintroduction to our knowledge graph and knowledge panels. https://blog.google/products/search/about-knowledge-graph-and-knowledge-panels, May 2020.

[89] Latanya Sweeney. Weaving technology and policy together to maintain confidentiality. *The Journal of Law, Medicine & Ethics*, 25(2-3):98–110, 1997.

[90] Wang-Chiew Tan. Unstructured and structured data: Can we have the best of both worlds with large language models? *CoRR*, abs/2304.13010, 2023.

[91] Yiming Tan, Dehai Min, Yu Li, Wenbo Li, Nan Hu, Yongrui Chen, and Guilin Qi. Evaluation of chatgpt as a question answering system for answering complex questions. *CoRR*, abs/2303.07992, 2023.

[92] Ken Thompson. Regular expression search algorithm. *Communications of the ACM (CACM)*, 11(6):419–422, 1968.

[93] James Thorne, Majid Yazdani, Marzieh Saeidi, Fabrizio Silvestri, Sebastian Riedel, and Alon Y. Levy. From natural language processing to neural databases. *Proc. VLDB Endow.*, 14(6):1033–1039, 2021.

[94] Logesh Kumar Umapathi, Ankit Pal, and Malaikannan Sankarasubbu. Med-halt: Medical domain hallucination test for large language models. *CoRR*, abs/2307.15343, 2023.

[95] Gary Wassermann, Carl Gould, Zhendong Su, and Premkumar T. Devanbu. Static checking of dynamically generated queries in database applications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(4):14, 2007.

[96] Benjamin Weggenmann and Florian Kerschbaum. Syntf: Synthetic and differentially private term frequency vectors for privacy-preserving text mining. *CoRR*, abs/1805.00904, 2018.

[97] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. Emergent abilities of large language models. *Trans. Mach. Learn. Res.*, 2022, 2022.

[98] Hailemariam Mehari Yohannes and Toshiyuki Amagasa. Named-entity recognition for a low-resource language using pre-trained language model. In Jiman Hong, Miroslav Bures, Juw Won Park, and Tomás Cerný, editors, *SAC '22: The 37th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, April 25 - 29, 2022*, pages 837–844. ACM, 2022.

[99] Fang Yu, Muath Alkhalaf, Tevfik Bultan, and Oscar H. Ibarra. Automata-based symbolic string analysis for vulnerability detection. *Formal Methods in System Design*, 44(1):44–70, 2014.

[100] Xiang Yue, Boshi Wang, Kai Zhang, Ziru Chen, Yu Su, and Huan Sun. Automatic evaluation of attribution by large language models. *CoRR*, abs/2305.06311, 2023.

[101] Haiyan Zhao, Hanjie Chen, Fan Yang, Ninghao Liu, Huiqi Deng, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, and Mengnan Du. Explainability for large language models: A survey. *CoRR*, abs/2309.01029, 2023.

[102] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A survey of large language models. *CoRR*, abs/2303.18223, 2023.

[103] Li Zhong and Zilong Wang. A study on robustness and reliability of large language model code generation. *CoRR*, abs/2308.10335, 2023.