Autonomously Computable Information Extraction

Besat Kassaie
Cheriton School of Computer Science
University of Waterloo
Waterloo, ON, Canada
bkassaie@uwaterloo.ca

Frank Wm. Tompa
Cheriton School of Computer Science
University of Waterloo
Waterloo, ON, Canada
fwtompa@uwaterloo.ca

ABSTRACT

Most optimization techniques deployed in information extraction systems assume that source documents are static. Instead, extracted relations can be considered to be materialized views defined by a language built on regular expressions. Using this perspective, we can provide an efficient verifier (using static analysis) that can be used to avoid the high cost of re-extracting information after an update. In particular, we propose an efficient mechanism to identify updates for which we can autonomously compute an extracted relation. We present experimental results that support the feasibility and practicality of this mechanism in real world extraction systems.

PVLDB Reference Format:
Besat Kassaie and Frank Wm. Tompa. Autonomously Computable Information Extraction. PVLDB, 16(10): 2431 - 2443, 2023.
doi:10.14778/3603581.3603585

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Besatkassaie/Differential-Maintenance-Engine.

1 INTRODUCTION

Information extraction is a crucial step in processing and understanding text. By identifying fields 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. *Knowledge Panels* [33] show how information extraction can improve the presentation of retrieval results in a commercial search engine. Extraction can be performed using ad-hoc text processing programs, machine learning, or reusable components offered by systems such as GATE [15] and SystemT [24]. In this paper, we address the third approach, where extractors are defined by regular expressions.

Extraction time can be a bottleneck for many applications [30, 32], 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. [5] propose an efficient algorithm for dictionary-based entity recognition, which can be used on many extraction platforms. Iperotis et al. [17] deal with many documents by keeping only "promising" documents for the extraction process. Shen et al. [32] propose a declarative approach to information extraction using Datalog with embedded procedural predicates to express extractors. This enables the creation of execution plans for the extraction program and thus the application of cost-based optimization techniques similar to query optimization in relational databases. Similarly, Reiss et al. [30] recommend using a SQL-like declarative language, AQL (used by SystemT), to be able to exploit query optimization strategies. Considering extractions within larger applications, Jain et al. [18] 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 re-computed 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 [6]. 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. [10] 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.

Freydenberger and Thompson [14] 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 permissible update is replacing one character from $\Sigma \cup \{\epsilon\}$ by another.

In a previous short paper [21], we have noted that the problem of re-evaluating extractions after updates is analogous to the problem of maintaining materialized views [7]. We identified updates to source documents that can be reflected in the extracted relation without re-computing the extractors (i.e., irrelevant updates), and we formulated the condition of an update being *pseudo-irrelevant*, where extracted regions are merely shifted in source documents. In unpublished extended work [20], we describe verification algorithms that attempt to determine whether an update is pseudo-irrelevant with respect to an extractor. Unfortunately, the algorithms rely on normalization that causes exponential blow-up in the input size and on finding complements and intersections of

Proceedings of the VLDB Endowment, Vol. 16, No. 10 ISSN 2150-8097.
doi:10.14778/3603581.3603585

automata, which in the worst case causes exponential blow-up in time and space.

No other prior work proposes to apply static analysis to programs that specify extractors and updates in order to determine whether re-extraction can be avoided or reduced.

In this paper, we continue the investigation of pseudo-irrelevant updates, one form of autonomously computable updates, for extractors expressed as document spanners (as defined below). To that end, in this paper:

(1) We extend the update model formalized in our earlier work to allow replacements to depend on the strings being replaced (Section 3).
(2) We propose a new algorithm to verify that an update is pseudo-irrelevant with respect to an extractor and prove that it runs in polynomial time via three theorems (Section 5). 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 in a proposition,[1] and it is built upon four auxiliary algorithms, each of which is proven to execute in polynomial time by an accompanying lemma.
(3) 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 (Section 6).

2 PRELIMINARIES

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 [30]. SystemT processes one document at a time and populates relational tables with *spans*, directly extracted from the input document. With SystemT, users encode extractors using a SQL-like language, namely the **A**nnotation **Q**uery **L**anguage (AQL), to manipulate tables.

The underlying principles adopted by SystemT have been formalized as *document spanners* by Fagin et al. [12]. Most of the material in this section has been introduced in that work and additional details can be found there.

2.1 Regular Expressions with Capture Variables

Given a finite alphabet Σ, a regular expression extended using variables chosen from a set V is called a *regex with capture variables* and conforms to γ in the grammar $G_S(\Sigma, V)$ as follows:

$$\begin{aligned} \gamma &:= \varnothing \mid \epsilon \mid \alpha \mid (\gamma \vee \gamma) \mid (\gamma \bullet \gamma) \mid (\gamma)^* \mid x\{\gamma\} \\ \alpha &:= \sigma \mid [\sigma, \sigma] \mid \delta \\ \delta &:= \Sigma \mid (\delta - \sigma) \end{aligned} \tag{1}$$

where σ represents any character in Σ, $[\sigma, \sigma]$ represents the disjunction of characters having their encoding between or equal to the encodings of the first and second character in the range, the terminal symbol Σ represents the disjunction of all characters in Σ, $\delta - \sigma$ represents the disjunction of all characters in δ with the exception of σ, and (what distinguishes these expressions from conventional regular expressions) x represents any variable in V. Given a regex with capture variables r, the corresponding *regex tree* $\mathcal{T}(r)$ represents the hierarchical structure of r, in which the tree's leaves have labels $\varnothing$, ϵ, characters in Σ, character ranges in Σ, or the character Σ itself, and internal nodes have labels $\bullet$, $\vee$, $*$, $-$, or a symbol in V. For convenience of notation, when writing a regex with capture variables, we follow common practice for regular expressions in allowing the following shorthand: omission of parentheses (relying instead on left associativity of all operations and precedence of $-$ over $*$ over $\bullet$ over $\vee$) and omission of the operator $\bullet$.

If E is a regex with capture variables, then we denote the set of capture variables in E as $SVars(E)$. The use of a subexpression of the form $n\{g\}$ in E signifies that whenever E matches a string, the span containing a substring matched by g is to be *marked by the capture variable n* (as explained in Section 2.3). It is apparent from the grammar that capture variables can be nested.

Example 2.1. Let Σ be the set of Latin alphanumeric, punctuation, and space characters (the last represented by $\_$). $\gamma_{fullDate}$ is a regex with capture variables:[2]

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

where $\gamma_d = [\texttt{0}, \texttt{9}]$ and $SVars(E) = \{F, Y, M, D\}$. For this example, we say that Y, M, D are *nested variables* and that F is *exposed*. We extend conventional set notation to write $Y \subset F$ and $\texttt{-} \in F$.

2.2 Document Spans

A document D is a finite string over some alphabet: $D \in \Sigma^*$ (Figure 1). A *span* of document D, denoted $[i, j\rangle$ $(1 \le i \le j \le |D| + 1)$, specifies the start and end offsets of a substring in D, which is in turn denoted $D_{[i,j\rangle}$, and extends from offset i through offset $j - 1$.

Example 2.2. In the document presented in Figure 1, $D_{[56,70\rangle}$ represents the substring `James␣F.␣Allen`.

$[i, i\rangle$ 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 $j_1 = j_2$.

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 [1]. These can be extended to capture the same basic relationships among spans (including empty spans) as summarized in Table 1. All possible relationships among spans can be described by disjunctions of these basic relationships; for example, "X overlaps Y" ($\Gamma_{(X \cap Y)}$) can be expressed as the disjunction of the last nine basic relationships.[3] "X overlaps but is not equal to Y" ($\Gamma_{(X \overset{\cap}{\neq} Y)}$) can be similarly expressed as the disjunction of the fifth through the twelfth basic relationships.

[1] An additional test for constraints on the alphabet is embedded in the algorithm.

[2] Throughout this paper, characters in Σ appearing in a formula are represented `like this` to distinguish them from the regular expression's meta-characters.

[3] The definition of overlapping spans given by Fagin et al. [12] 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 $[i, j\rangle$. We treat both as overlapping.

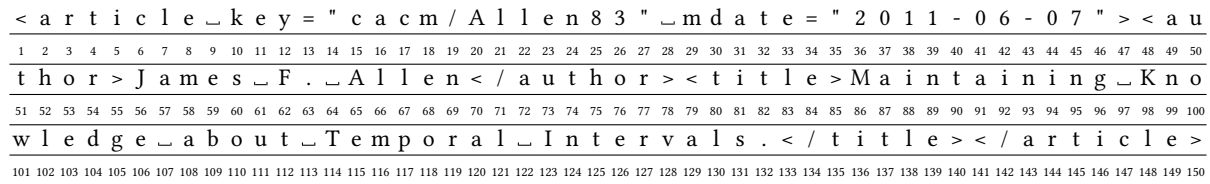

Figure 1: A sample input document D for our running example.

Table 1: Allen's interval relationships extended to spans.

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

[a] This abuse of notation represents a spanner (as described in Section 2.3) matching an automaton with operators that open and close the variables X and Y as indicated.

2.3 Extractors Expressed by Document Spanners

Determining membership in a language defined by a regex with capture variables E can be accomplished by executing a corresponding *vset-automaton* $\mathcal{A}(E)$. Given E and $V = SVars(E)$, $\mathcal{A}(E)$ is a non-deterministic finite state automaton augmented with a designated set (initially empty) and two operators for each variable $x \in V$, namely $x\vdash$ ("open x") and $\dashv x$ ("close x"). Besides including standard character transitions, $\mathcal{A}(E)$ also includes *operation transitions* that, instead of consuming a character from the input string, insert the variable x into the designated set if the transition label is $x\vdash$ and remove x from the designated set if the label is $\dashv x$. A document D is accepted by $\mathcal{A}(E)$ if, after scanning the whole input, we end up in a final state and the designated set is empty. A *matching* of E against document D is an accepting run in $\mathcal{A}(E)$, where for each variable $x \in SVars(E)$, the spans marked by x each begin with the offset in D when x is inserted into the designated set and end with the offset in D when x is removed from that set.

If E is a regex with capture variables, it specifies a *document spanner*, denoted as $[\![E]\!]$, with $SVars([\![E]\!]) = SVars(E)$. Applying a document spanner to a document D produces a *span relation*, i.e., a relation that contains spans of D. Thus, $[\![E]\!]$ is a function mapping strings over Σ^* to $\mathcal{S}^{|SVars(E)|}$ where $\mathcal{S}$ is the set of all spans of D. To ensure that the span 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 mark exactly one span for each variable for all accepting runs, regardless of the input document D. 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 $SVars(E)$ 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 $x \in SVars(E)$ is the span marked by x.

Example 2.3. Let γ_d be as defined in Example 2.1. Applying the spanner represented by $\gamma_{partialDate} = \Sigma^*F\{M\{\gamma_d\gamma_d\}\text{-}D\{\gamma_d\gamma_d\}\}\Sigma^*$ to the document in Figure 1 results in the span relation in Figure 2.

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

Figure 2: The extracted relation $[\![\gamma_{partialDate}]\!](D)$, where D is depicted in Figure 1.

Definition 2.4. Throughout this paper, a functional document spanner used for the purpose of information extraction is called an *extractor*, the regex with capture variables defining it is called an *extraction formula*, and the span relation produced for a document is called an *extracted relation*.

Let S, S_1, and S_2 be extractors where the last two are union-compatible (i.e., $SVars(S_1) = SVars(S_2)$); $X \subseteq SVars(S)$; and $x, y \in SVars(S)$. An algebra over spanners can be defined with operators:

(1) union: $S_1 \cup S_2$ having variables $SVars(S_1 \cup S_2) = SVars(S_1)$,
(2) projection: $\pi_X(S)$ having variables X,
(3) natural join: $S \bowtie S_1$ having variables $SVars(S) \cup SVars(S_1)$, and
(4) binary string selection: $\zeta^=_{x,y}S$ having variables $SVars(S)$.

The set of *core spanners* (corresponding to the core of SystemT's AQL) includes extractors specified by any extraction formula (so-called *primitive extractors*) together with all extractors in the closure of core spanners under this algebra. Applying a core spanner to any document D is equivalent to applying each included primitive spanner to D and then applying the corresponding relational operators to the extracted relations. The set of core spanners define the extractors and updates subject to analysis in this paper.

2.4 Efficient Construction of Extractors

Given one or more spanners as input, we use various algebraic operations defined over spanners to verify properties of those spanners statically. Specifically, we convert the inputs to *eVset-automata*, a variant of vset-automata with the same expressivity, as proposed by Morciano [26]. By preserving three properties of eVset-automata

(namely, *well-behaved*, *pruned*, and *operation-closed*), Morciano is able to construct eVset-automata in polynomial time for simulating the application of projection, union, and join over spanners. His thesis also shows that converting a regex with capture variables to an automaton and checking for emptiness can be done in polynomial time. It is trivial to show that renaming variables can also be accomplished in polynomial time with eVset-automata.

2.5 Model Architecture

In this work, we hypothesize systems that include a document database $\mathbb{D}$ and a set of core spanners $\{\mathbb{E}_1, \cdots, \mathbb{E}_e\}$ specifying extractors that run over $\mathbb{D}$. The union of span relations produced by $\mathbb{E}_k$ against the document database is stored in a relation $\mathbb{T}_k$ that includes an additional column to associate a document identifier with the spans for the corresponding extracted relation. These tables serve as materialized views of the document database.

3 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 paper, we concentrate on the latter form of update, where substring *replacement*, *deletion*, and *insertion* are basic update operations that we wish to support. A change to the text is typically preceded by some browsing activities or search operations to locate update positions in a target document. We use an extractor as our search mechanism and to specify which strings to replace.

Definition 3.1. An *update expression* is denoted as $Repl(g, U)$, where g is an update formula with one exposed capture variable that determines which spans contain strings to be replaced and U is a string that specifies the replacement value.

Given an update expression $Repl(g, U)$, the string U may include named back-references to g, similar to the mechanisms defined in several programming languages: $U \in ((\Sigma - \texttt{\$}) \cup (\texttt{\$(}\ SVars(g)\ \texttt{)}))^*$.

Example 3.2. Consider $Repl(\gamma_{DOI}, \texttt{"DOI:}\$(C)\texttt{"})$, where

$$\gamma_{DOI} = \Sigma^*\texttt{<ee>}(F\{\texttt{https://doi.org/}C\{(\Sigma - \texttt{<})^*\}\} \vee F\{\texttt{doi:}C\{(\Sigma - \texttt{<})^*\}\})\texttt{</ee>}\Sigma^*$$

F is the exposed variable, indicating that strings enclosed by <ee> ... </ee> and starting with a DOI specifier should be replaced. The second argument for the update expression indicates that each replacement should consist of the string "DOI: followed by whatever matches the capture variable C followed by another quotation mark.

3.1 Update Formulas

An *update formula* is a functional extraction formula with one exposed variable (v) and conforms to $\bar{\gamma}$ in the grammar $G_U(\Sigma, V, v)$:

$$\bar{\gamma} := (\bar{\gamma} \vee \bar{\gamma}) \mid (\gamma' \bullet \bar{\gamma}) \mid (\bar{\gamma} \bullet \gamma') \mid v\{\gamma''\} \quad (2)$$

where γ' is a variable-free regular expression, γ'' is a regex with capture variables V, and the exposed variable v is an additional capture variable (the *update variable*) within which all other capture variables are nested. Given an update formula g, the update variable is denoted as $UVar(g)$. The extraction formulas presented in Examples 2.1 and 2.3 conform to $\bar{\gamma}$ where F is the update variable.

Definition 3.3. An update formula is in *normalized form* if it is written as $\bigvee_{i=1}^{k} g_i$ where each g_i is a formula conforming to $\hat{\gamma}$:

$$\hat{\gamma} := \gamma' \bullet v\{\gamma''\} \bullet \gamma'$$

where each γ' is again a variable-free (possibly empty) regular expression and γ'' is again a regex with capture variables V.

To normalize an update formula, all disjunctions that have the update variable v in their disjuncts[4] can be "pulled up" over concatenations in the corresponding extended regex tree to create separate disjuncts at the outermost level of the formula.

Example 3.4. The expression γ_{DOI} in Example 3.2 conforms to grammar (2), and its corresponding normalized form is:

$$\gamma_{DOI} = \Sigma^*\texttt{<ee>}F\{\texttt{https://doi.org/}C\{(\Sigma - \texttt{<})^*\}\}\texttt{</ee>}\Sigma^* \vee \Sigma^*\texttt{<ee>}F\{\texttt{doi:}C\{(\Sigma - \texttt{<})^*\}\}\texttt{</ee>}\Sigma^*$$

Lemma 3.5. *Given any update formula g, Algorithm 1 normalizes it by producing the list of disjuncts $\Delta(g, UVar(g))$ in time that is polynomial in $|g|$, where $|x|$ denotes the length of x.*

Proof. The proof of correctness is by induction on the height of the expression tree, which can be constructed and traversed (using recursive descent) in linear time in the input length. For each occurrence of $UVar(g)$, 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 $UVar(g)$ in g.

□

3.2 Update Semantics

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 $[\![g]\!](D)$. When the spanner is used for updating a document D, substrings of D associated with the spans in the update relation are simultaneously replaced by new values specified by U.

More precisely, 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 substring $s_i \in D$ corresponding to a span marked by $UVar(g)$ is replaced by the string U_i where, for any $v \in SVars(g)$, U_i is the same as U except that every occurrence of a substring $\texttt{\$}(v)$ in U is replaced by the string in s_i corresponding to the span marked by v.

Note that if U is the empty string, then the update results in the deletion of the substrings 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 $i = |D| + 1$).

An update yields a specific functional mapping between the original and updated documents: $ReplSpan(g, U)([i, j)) \rightarrow [i', j'\rangle$ 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 the complete span marked by $UVar(g)$ (Figure 3). If $[i, j\rangle$ is disjoint from all spans marked by $UVar(g)$ when updating D, then $D'_{[i',j'\rangle} = D_{[i,j\rangle}$.

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

Algorithm 1: Normalize g with respect to v.

Input: extraction formula g with exposed variable v
Output: list of disjuncts $\Delta(g, v)$
Precondition: g is functional

$\mathcal{T} \leftarrow ExpressionTree(g)$;
return $normalize(\mathcal{T}, v)$

Function *normalize* ($\mathcal{T}$: expression tree, v: variable) : list
  Precondition: v is in $\mathcal{T}$
  $\Delta(g, v) \leftarrow list()$;
  if $\mathcal{T}.root == v$ **then**
    $\Delta(g, v).add(toRegExp(\mathcal{T}))$
  end
  if $\mathcal{T}.root == \vee$ **then**
    /* normalize subtrees */
    $\Delta(g, v).add(normalize(\mathcal{T}.left, v))$;
    $\Delta(g, v).add(normalize(\mathcal{T}.right, v))$
  end
  if $\mathcal{T}.root == \bullet$ **then**
    /* if v occurs in right (left) subtree, normalize right (left) subtree and then concatenate every expression in $\Delta(g, v)$ with left (right) expression */
    if v *in subtree*$(\mathcal{T}.right)$ **then**
      $\Delta(g, v).add(normalize(\mathcal{T}.right, v))$;
      $toRegExp(\mathcal{T}.left) \bullet \Delta(g, v)$
    else
      $\Delta(g, v).add(normalize(\mathcal{T}.left, v))$;
      $\Delta(g, v) \bullet toRegExp(\mathcal{T}.right)$
    end
  end
  return $\Delta(g, v)$
end

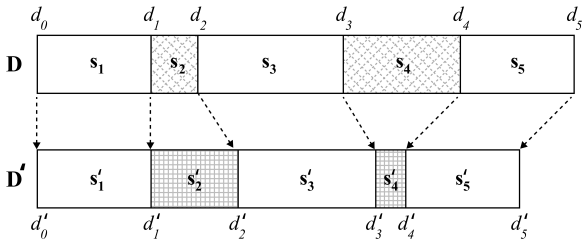

Figure 3: A sample document D and its updated peer D'. Filled areas in D are marked by the update variable. *ReplSpan* **maps $[d_i, d_{i+1}\rangle$ to $[d'_i, d'_{i+1}\rangle$.**

Definition 3.6. An update spanner specified by g is said to be *well-defined* if the following two properties hold for every 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.

Proposition 3.7. *Whether or not an update spanner specified by g is well-defined can be verified in polynomial time.*

Proof. The proof has two similar parts, one for each property. First we show that detecting whether there can be overlaps in spans that will be changed can be done by constructing a specific spanner and testing for emptiness. Then we show that detecting whether a single updated span might have ambiguously marked subspans can also be done by constructing a specific spanner and testing for emptiness. All operations take polynomial time.

Given an update formula g and symbols $X \notin SVars(g)$ and $Y \notin SVars(g)$, we create the spanner

$$conflicts(g) = \pi_X(\rho_{UVar(g)\to X}([\![g]\!])) \bowtie [\![\Gamma_{(X \between Y)}]\!] \bowtie \pi_Y(\rho_{UVar(g)\to Y}([\![g]\!]))$$

Recall that $\Gamma_{(X \between Y)}$ is the disjunction of the fifth through the twelfth basic relationships in Table 1; that is, we are testing whether spans in $\pi_{UVar(g)}([\![g]\!](D))$ could include two unequal spans that cover identical sub-spans. If $conflicts(g) = \emptyset$, then g satisfies the first condition. The operators utilized to construct $conflicts(g)$ use time that is at most quadratic in the size of inputs (Section 2.4).

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

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

where $\Gamma_{(X \neq Y)}$ is the disjunction of the first 12 basic relationships in Table 1; that is, spans in $[\![g]\!](D)$ could include two rows that match on $UVar(g)$ but do not match on Z. If $ambig(g, Z) = \emptyset$ for all $Z \in SVars(g) \setminus \{UVar(g)\}$, then g satisfies the second condition. Clearly the time complexity to check this property is also polynomial in the size of the input. □

Example 3.8. The spanner presented in Example 2.3 is not well-defined, since $conflicts(\gamma_{partialDate}) \neq \emptyset$ (M in one match for F can overlap D in another match).

Example 3.9. The spanner represented by the following extraction formula is easily verified to be not well-defined: the context ensures that it is free of conflicts, but it is ambiguous:

$$\gamma'_{date} = \Sigma^* \boxed{\texttt{<now>}} F\{Y\{\gamma^*_d\}M\{\gamma^*_d\}D\{\gamma^*_d\}\} \boxed{\texttt{</now>}} \Sigma^*$$

4 AUTONOMOUSLY COMPUTABLE UPDATES

Given an update expression $Repl(g, U)$ and an extractor $[\![E]\!]$, 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 [4, 16, 21].

Definition 4.1. 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 an extracted relation that is identical to applying $[\![E]\!]$ to D. That is, if $D' = Repl(g, A)(D)$, then $[\![E]\!](D') = [\![E]\!](D)$.

If an update expression is not irrelevant with respect to an extractor, it may still be that the modification to the extracted relation can be computed without re-running the extractor.

Definition 4.2. An update expression $Repl(g, U)$ is *autonomously computable* with respect to an extractor $[\![E]\!]$ if for every document in the database $\mathbb{D}$, applying $[\![E]\!]$ to $Repl(g, U)(D)$ can be computed from the update expression, the update relation, the extraction formula that defines the extractor, and the extracted relation.[5]

There is an important distinction between the problems of updating traditional relational views and updating materialized extractions. Extracted relations contain pairs of offsets from input documents, not document content. Thus, as happens for spans within regions S_3 and S_5 in Figure 3, replacing a string of one length by a string of another length somewhere in the document will cause a span further down in the document to shift, even if the content of that span is unaffected.

More generally, 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 $[\![g]\!]$, $ReplSpan(g, U)(S)$ 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 lengths of the strings specified by U, as captured by Algorithm 2.

Algorithm 2: Shift a span.

Input: span $S = [i, j\rangle$, update relation R, update variable X, replacement specification U
Output: span $S' = [i', j'\rangle = ReplSpan(g, U)(S)$
Precondition: R contains no span that overlaps S or another span in R

```
shift, l ← 0;
for t ∈ R do
    [m, n⟩ ← t[X]; /* span for the update variable */
    if m < i then
        l ← computeLength(U, t); /* using each |$(v_i)| from t */
        shift ← shift + (n − m) − l
    end
end
return [i − shift, j − shift⟩
```

Definition 4.3. Update expression $Repl(g, U)$ is *pseudo-irrelevant* with respect to an extractor $[\![E]\!]$ if for every document D and updated document $D' = Repl(g, U)(D)$, $[\![E]\!](D') = \{S' \mid \exists S \in [\![E]\!](D), S' = ReplSpan(g, U)(S)\}$ (i.e., extracted spans merely shift).

Thus, a pseudo-irrelevant update is a special case of an autonomously computable update. By definition, if an update expression is irrelevant with respect to an extractor, then it is also pseudo-irrelevant with respect to that spanner (all shifts are of length 0).

5 CATEGORIZING DOCUMENT UPDATES

We wish to identify whether an update is pseudo-irrelevant with respect to a given extractor, independently of input documents. The essence of our approach is to inspect various kinds of *overlap* between an update expression and an extractor. The proposed process verifies some *sufficient* conditions for pseudo-irrelevant updates.

By examining Definition 4.3, we can deduce three possibilities that could cause an update (changing document D to become D') to fail to be pseudo-irrelevant 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.

Example 5.1. Consider the wh-word extractor specified by

$$\Sigma^* \boxed{\_}\, Q\{\boxed{\texttt{Wh}}\, \gamma_{lc}^+\}\, \boxed{\_}(\Sigma - \boxed{.} - \boxed{!} - \boxed{?})^*\, \boxed{?}\, \Sigma^*$$

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.

We leave it to future work to determine under what conditions an update that overlaps extracted spans (e.g., changing "Who" to "Why" in the example) or their contexts (e.g., changing periods to exclamation points for this extractor) happens to be pseudo-irrelevant. Instead, we determine when there can be no overlap, and then under which further conditions an update is pseudo-irrelevant. Section 5.1 describes how to identify the contexts of extracted spans. Section 5.2 describes how to identify spans in an updated document D' that have had their contents changed by an update to D. Finally Section 5.3 describes a sound, but not complete, mechanism to determine whether an update expression, specified by the update formula and replacement specifier, is pseudo-irrelevant with respect to a document spanner specified by an extraction formula.

5.1 Contextualization of Extraction Formulas

For an extraction formula to match a document, certain strings must appear either inside or 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. The tests we have implemented compare updated regions to regions marked by a capture variable, so, given the specification of a document spanner E with exposed variable v, we define a corresponding regular formula $C_v(E)$ in which all contextual expressions are also marked with capture variables. For any document D, $[\![C_v(E)]\!]$ produces the same spans for the capture variable v as does $[\![E]\!]$, but it also captures spans of characters that must remain unchanged. Each uncovered subexpression in $C_v(E)$ is of the form X_i^* where $L(X_i)$ includes *unigrams* (i.e., strings of length 1) only, essentially restricting the uncovered subexpression to strings over a restricted alphabet $\hat{\Sigma}_i \subseteq \Sigma$. For example, in Example 5.1, $\boxed{\texttt{Wh}}\gamma_{lc}^*$ is already covered by the capture variable Q, but the blanks before and after that pattern are a part of the context than cannot change, as is the question mark at the end; they should be covered. On the other hand, the expressions

[5] We note in passing that updating the span relation when adding documents to or deleting documents from $\mathbb{D}$ can be performed autonomously. For new documents, we run the extractors on those documents only (the text of which forms part of the update) and then add the extracted tuples to the materialized view without any need to reference documents in $\mathbb{D}$. When deleting documents, we simply delete all tuples with corresponding document identifiers from the extracted relation.

Algorithm 3: Capture contexts for $v \in SVars(E)$.

Input: extraction formula E, $v \in SVars(E)$
Output: modified extraction formula $C_v(E)$
Precondition: v is exposed, $v_i \notin SVars(E)$ for all i

$\mathcal{D} \leftarrow \Delta(E, v)$; /* get the disjunctive form using Algorithm 1 */
$D_c \leftarrow emptyList()$; /* prepare for list of covered expressions */
forall $d \in \mathcal{D}$ **do**
  $\mathcal{T}_c \leftarrow cover(ExpressionTree(d), 1)$; /* start cover with v_1 */
  $\mathcal{T}_m \leftarrow mergeConsecVars(\mathcal{T}_c)$; /* $v_i\{e_i\}v_j\{e_j\} \rightarrow v_k\{e_ie_j\}$ */
  $D_c.add(\mathcal{T}_m)$ /* keep covered expr for disjunct */
end
/* ensure each disjunct includes all v_i so $C_v(E)$ is functional */
$result \leftarrow \epsilon$;
forall $d \in D_c$ **do**
  $\mathcal{T}_v \leftarrow findSubtree(d, v)$; /* find node with v */
  forall $y \in getAllVars(D_c) \setminus getAllVars(d)$ **do**
    /* change $v\{\ldots\}$ to $v\{y\{\ldots\}\}$ in this disjunct */
    $setParent(setParent(\mathcal{T}_v.child, newNode(y)), \mathcal{T}_v)$
  end
  $result \leftarrow result \vee toRegExp(d)$ /* add disjunct to $C_v(E)$ */
end
return *result*

Function *cover* ($\mathcal{T}$: *expression tree, i:int*) : *expression tree*
  Output: cover contexts in a subtree, starting with v_i
  $\mathcal{T}_c \leftarrow emptyTree()$;
  if $\mathcal{T}.root == *$ **then**
    if $matchesUnigrams(\mathcal{T}.root.left)$ **then**
      $\mathcal{T}_c = \mathcal{T}$ /* $\hat{\Sigma}^*$ does not get covered */
    else
      $\mathcal{T}_c \leftarrow setParent(\mathcal{T}.root, newNode(v_i))$
    end
  end
  if $\mathcal{T}.root == \vee$ **then**
    $\mathcal{T}_c \leftarrow setParent(\mathcal{T}.root, newNode(v_i))$
  end
  if $varNode(\mathcal{T}.root)$ **then**
    $\mathcal{T}_c = \mathcal{T}$ /* already covered */
  end
  if $\mathcal{T}.root == \bullet$ /* traverse both sides */
  then
    $\mathcal{T}.left \leftarrow cover(\mathcal{T}.left, i)$;
    $i' = maxIndex(\mathcal{T}.left)$;
    if $i \leq i'$ **then**
      $i = i' + 1$ /* set i to the largest index so far */
    end
    $\mathcal{T}.right \leftarrow cover(\mathcal{T}.right, i)$;
    $\mathcal{T}_c \leftarrow \mathcal{T}$
  else
    $\mathcal{T}_c \leftarrow setParent(\mathcal{T}.root, newNode(v_i))$ /* unigram */
  end
  return $\mathcal{T}_c$
end

$(\Sigma - [.] - [!] - [?])^*$ and Σ^* are examples of $\hat{\Sigma}^*$ that can be freely updated by any string in their languages.

Lemma 5.2. *Given an extraction formula E and exposed variable $v \in SVars(E)$, Algorithm 3 returns $C_v(E)$ in quadratic time.*

Proof. Using induction on the height of the expression tree for E, it is straightforward to show that Algorithm 3 covers all subexpressions except for the closure of those that satisfy the predicate *matchesUnigrams*. Testing this predicate requires merely checking that no node in the subtree is labelled $\bullet$ or $*$.
Complexity Analysis: Given an input expression E, the corresponding expression tree can be constructed in linear time in the input size, and projection via a top-down traversal of the expression tree can also be performed in linear time. Normalization has time complexity that is linear in the input length (Lemma 3.5), and $cover()$ and $mergeConsecVars()$ require a top-down traversal over the expression tree, again in linear time. For an input expression E with $m < |E|$ occurrences of v, the time complexity of the algorithm is thus $O(m * |E|)$. □

To be most effective in identifying contexts, we wish to identify as many instances of X_i^* as we can, since they indicate portions of the document that can be (almost) freely updated. To this end, standard rewrite rules for regular expressions can be applied, even in the presence of capture variables. For example, if R_i are any regular expressions with capture variables and $\hat{\Sigma}$ is any expression matching a unigram, distributive laws can be used to pull X_i^* outside disjunctions:

$$(\hat{\Sigma}^* R_1) \vee (\hat{\Sigma}^* R_2) \rightarrow \hat{\Sigma}^*(R_1 \vee R_2)$$
$$(R_1\hat{\Sigma}^*) \vee (R_2\hat{\Sigma}^*) \rightarrow (R_1 \vee R_2)\hat{\Sigma}^*$$

Example 5.3. Consider the following expression:

$$\gamma_{jb} = (\Sigma^*\langle\texttt{journal}\rangle \vee \Sigma^*\langle\texttt{booktitle}\rangle)T\{\gamma_t\}$$
$$(\langle\texttt{/journal}\rangle\Sigma^* \vee \langle\texttt{/booktitle}\rangle\Sigma^*)$$

Applying Algorithm 3 to γ_{jb} with variable T produces

$$C_T(\gamma_{jb}) = v_1\{(\Sigma^*\langle\texttt{journal}\rangle \vee \Sigma^*\langle\texttt{booktitle}\rangle)\}T\{\gamma_t\}$$
$$v_2\{(\langle\texttt{/journal}\rangle\Sigma^* \vee \langle\texttt{/booktitle}\rangle\Sigma^*)\}$$

After rewriting with the distributive laws, the algorithm produces

$$C_T(\gamma_{jb'}) = \Sigma^* v_1\{(\langle\texttt{journal}\rangle \vee \langle\texttt{booktitle}\rangle)\}T\{\gamma_t\}$$
$$v_2\{(\langle\texttt{/journal}\rangle \vee \langle\texttt{/booktitle}\rangle)\}\Sigma^*$$

marking the portions that are required to be in any matching document, while leaving matches to arbitrary text (i.e., Σ^*) unmarked.

Other simple rewrite rules, such as removing superfluous closures ($(R^*)^* \rightarrow R^*$, where R is any regular expression) and removing superfluous disjunctions ($L(R) \subseteq L(\hat{\Sigma}^*) \implies \hat{\Sigma}^* \vee R \rightarrow \hat{\Sigma}^*$), also provide means to expose more instances of $\hat{\Sigma}^*$.

5.2 Post-Update Spanner

We use information provided by the update expression to construct a *post-update spanner* that identifies regions in updated documents corresponding to updated spans. First we need to ensure that an update does not modify any of the context used to identify which spans to update.

Definition 5.4. An update spanner $[\![g]\!]$ is *durable* if it is well-defined and spans marked by the update variable are disjoint from spans marked by context variables after applying $[\![C_{UVar(g)}(g)]\!]$ to any document D.

Proposition 5.5. *Testing whether an update spanner is durable can be performed in polynomial time.*

Proof. Again the proof has two parts, one for each property. Testing for being well-defined is straightforward. Then we show that detecting whether an updated span might overlap with a span marked by a context variable can be done by constructing a specific spanner and testing for emptiness in polynomial time.

Let v represent $UVar(g)$. Testing whether $[\![g]\!]$ is well-defined requires polynomial time (Proposition 3.7), and $C_v(g)$ can be constructed in polynomial time (Lemma 5.2).

Let v_i represent the i^{th} variable marking the context in $C_v(g)$ and construct a spanner $p_i(g)$:

$$p_i(g) = \pi_X(\rho_{v\to X}([\![C_v(g)]\!])) \bowtie [\![\Gamma_{(X\cap Y)}]\!] \bowtie \pi_Y(\rho_{v_i\to Y}([\![C_v(g)]\!]))$$

where $\Gamma_{(X\cap Y)}$ is again the disjunction of the fifth through thirteenth basic relationships in Table 1. Therefore, the spanner $p_i(g)$ matches all documents that can be updated while spans marked by the update variable have at least one subspan in common with the set of spans marked as context. Therefore, if for all i, $p_i(g) = \emptyset$, then $[\![g]\!]$ is durable. Constructing $p_i(g)$ and testing for emptiness requires polynomial time, and the number of cover variables is less than the size of the input. □

Given a specification of an update spanner, we derive a new spanner to match documents that result from an update. Some substrings in an updated document come from outside the regions matched by the update variable, and others come from the replacement specifier U, either from substrings explicitly contained in U or from the use of back-references to substrings in the original document. Thus, U implicitly describes a language $L(U)$ comprising the set of possible replacement values, but it uses back-references to variables in g. We define a regular language $\diamond(U,g)$ that describes a slightly broader space of replacement values: $\diamond(U,g) \supseteq L(U)$ and $\diamond(U,g) \setminus L(U)$ is fairly small.

Lemma 5.6. *Given a replacement specifier U and a well-defined spanner specified by g, Algorithm 4 outputs $\diamond(U,g)$ with time complexity $O(|g| + |U|)$.*

Proof. The proof is by construction. Given g and

$$U = u_1 \cdots u_i \texttt{\$(}v_i\texttt{)}\ u_{i+1} \cdots u_j \texttt{\$(}v_j\texttt{)}\ u_{j+1} \cdots u_n$$

Algorithm 4 scans U and g and substitutes each back-reference `$(`v_i`)` with the disjunction of all expressions specified with that variable in g: $(exp_1^{v_i} \vee exp_2^{v_i} \vee \cdots)$, omitting other capture variables enclosed by v_i.[6] 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 encountered, and in $\diamond(U,g)$ the corresponding run will encounter its associated back-reference in U. The resulting regular expression is:

$$\diamond(U,g) = u_1 \bullet \cdots \bullet u_i \bullet (exp_1^{v_i} \vee exp_2^{v_i} \vee \cdots) \bullet u_{i+1} \bullet \cdots \bullet u_j \bullet (exp_1^{v_j} \vee exp_2^{v_j} \vee \cdots) \bullet u_{j+1} \bullet \cdots \bullet u_n$$

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 and then the expression tree of g is scanned, substituting back-references in U with associated regular expressions taken from g. The complexity is $O(|U|+|g|)$. □

Definition 5.7. 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 is specified by $\nabla(g,U)$.

Example 5.8. For the update expression in Example 3.2, the corresponding post-update spanner is

$$\nabla(\gamma_{DOI},\ \texttt{"DOI:\$(}C\texttt{)"}) = \Sigma^*\texttt{<ee>}F\{\texttt{"DOI:}(\Sigma - \texttt{<})^*\texttt{"}\}\texttt{</ee>}\Sigma^* \vee \Sigma^*\texttt{<ee>}F\{\texttt{"DOI:}(\Sigma - \texttt{<})^*\texttt{"}\}\texttt{</ee>}\Sigma^*$$

[6] Eliminating capture variables inside v_i does not affect the set of strings marked by v_i.

Algorithm 4: Define space of replacement values $\diamond(U,g)$.

Input: update expression $Repl(g,U)$
Output: $\diamond(U,g)$
$\mathcal{T} \leftarrow ExpressionTree(g)$;
$bkrefs[] \leftarrow get\_bkrefs(U)$; /* get all back-references from U */
$result\_strs \leftarrow \{\}$; /* empty dictionary */
forall *bkref in bkrefs* **do**
  /* retrieve all substrings associated with *bkref* */
  $subexps[] \leftarrow get\_subexp(\mathcal{T}, bkref)$;
  /* concatenate all substrings separated by disjunction symbols */
  $result\_str \leftarrow$ "∨".*join*(*toRegExp*(*subexp*[]));
  /* enclose with parentheses and add to the dictionary */
  $result\_strs[bkref] \leftarrow$ "(" + *result_str* + ")"
end
/* substitute each bkref with corresponding string in dictionary */
$\diamond(U,g) \leftarrow replace(U, result\_strs)$;
return $\diamond(U,g)$

Algorithm 5: Build post-update spanner $\nabla(g,U)$.

Input: update expression $Repl(g,U)$, update variable x
Output: $\nabla(g,U)$
Precondition: $[\![g]\!]$ is durable
1 $\mathcal{D} \leftarrow \Delta(g,x)$; /* get the disjunctive form using Algorithm 1 */
2 $\nabla(g,U) \leftarrow \emptyset$;
  /* process each disjunct g_i in $\Delta(g,x)$ */
3 **forall** $g_i \in \mathcal{D}$ **do**
4 $\mathcal{T} \leftarrow ExpressionTree(g_i)$;
  /* get subexpressions on left/right of update variable */
5 $\mathcal{T}_l \leftarrow get\_left(\mathcal{T}, x)$;
6 $\mathcal{T}_r \leftarrow get\_right(\mathcal{T}, x)$;
7 $\nabla(g,U) \leftarrow \nabla(g,U) \vee (toRegExp(\mathcal{T}_l) \bullet x\{\diamond(U,g_i)\}$
8 $\bullet toRegExp(\mathcal{T}_r))$
9 **end**
10 **return** $\nabla(g,U)$

Lemma 5.9. *For a durable update spanner $[\![g]\!]$, Algorithm 5 outputs $\nabla(g, U)$ in quadratic time in the input length.*

Proof. The proof is by contradiction, using Figure 3 for intuition.

Let D be a document that has m regions marked by $x = UVar(g)$ when processed by $[\![g]\!]$ and D' denote the result of updating D by $Repl(g, U)$. 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'. Using Figure 3 as a guide, let s_4 be the *leftmost* problematic region of D. Because the update spanner is durable, all subexpressions 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 (e.g., s'_1, s'_2, and s'_3 are correctly identified). Based on Lemma 5.6, the region s'_4 must match $\diamond(U, g)$. Thus the substring from the start of D to the end of s_4 ($D_{[d_0,d_4\rangle}$) corresponds correctly to the substring from the start of D' to the end of s'_4 ($D'_{[d'_0,d'_4\rangle}$). Therefore, s_4 cannot be problematic, and so there cannot be 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 contexts is $O(|g|)$, and the complexity of constructing $\diamond(U, g_i)$ is $O(|g_i| + |U|)$. Finally, if g has m disjuncts in $\Delta(g, x)$, m is $O(|g|)$ and thus the time complexity of the algorithm is $O(|g|^2 + |g| * |U|)$). □

5.3 Detecting Pseudo-Irrelevance for Spanners

An update is pseudo-irrelevant with respect to a core spanner S if it is pseudo-irrelevant with respect to each primitive extractor defining S:

Theorem 5.10. *Let $\mathbb{E}_{Repl(g,U)}$ represent the set of extractors for which $Repl(g, U)$ is pseudo-irrelevant. $\mathbb{E}_{Repl(g,U)}$ is closed under union, projection, natural join, and string selection.*

Proof. We need to show that if $[\![E_1]\!] \in \mathbb{E}_{Repl(g,U)}$, $[\![E_2]\!] \in \mathbb{E}_{Repl(g,U)}$, and $Y \subseteq SVars(E_1)$, then

(1) $[\![E_1 \cup E_2]\!] \in \mathbb{E}_{Repl(g,U)}$ when $[\![E_1]\!]$, $[\![E_2]\!]$ are union compatible,
(2) $[\![\Pi_Y E_1]\!] \in \mathbb{E}_{Repl(g,U)}$,
(3) $[\![E_1 \bowtie E_2]\!] \in \mathbb{E}_{Repl(g,U)}$, and
(4) $[\![\zeta^R_Y E_1]\!] \in \mathbb{E}_{Repl(g,U)}$.

Each of these can be proven by contradiction. □

Definition 5.11. Let $[\![\mathcal{S}]\!]$ be a spanner defined by update formula g or a post-update spanner specified by $\nabla(g, U)$. Given an extraction formula E with exposed variable v, $[\![E]\!]$ is *independent* of $[\![\mathcal{S}]\!]$ if for every document D, $[\![C_v(E)]\!](D)$ includes no span that overlaps a span in $[\![\mathcal{S}]\!](D)$. Otherwise, we say that $[\![E]\!]$ depends on $[\![\mathcal{S}]\!]$.

Proposition 5.12. *Independence can be verified in polynomial time.*

Proof. As before, we show that detecting whether an updated span might overlap a span marked by a context variable can be done by constructing a specific spanner and testing for emptiness in polynomial time.

First rename variables such that $UVar(g) \notin SVars(C_v(E))$ and $X, Y \notin (SVars(C_v(E)) \cup \{UVar(g)\})$, and let $v_i \in SVars(C_v(E))$. Then construct the following spanner:

$$depends(E, g, v_i, UVar(g)) = \pi_X(\rho_{v_i \to X}([\![C_v(E)]\!])) \bowtie [\![\Gamma_{(X \cap Y)}]\!] \bowtie \pi_Y(\rho_{UVar(g) \to Y}([\![g]\!]))$$

Again, $\Gamma_{(X \cap Y)}$ is the disjunction of the fifth through thirteenth basic relationships in Table 1, capturing overlapping regions. The spanner $depends\ (E, g, v_i, UVar(g)) \neq \emptyset$ if there exists at least one span in $\pi_{v_i}([\![C_v(E)]\!])$ that touches at least one span in $\pi_{UVar(g)}([\![g]\!])$. Therefore, if for all such v_i, $depends(E, g, v_i, UVar(g)) = \emptyset$, then $[\![E]\!]$ is independent of $[\![g]\!]$. There are at most $|E|$ such tests, each of which takes polynomial time, proving the claim. □

Definition 5.13. Given an extraction formula E with exposed variable v, an update expression $Repl(g, U)$ *respects the alphabets* in $C_v(E)$ if, for every expression of the form $\hat{\Sigma}^*$ not covered in $C_v(E)$, the update neither deletes nor inserts a symbol in $(\Sigma - \hat{\Sigma})$.

If $Repl(g, U)$ does not respect the alphabets in $C_v(E)$, tuples might be inserted into or deleted from an extracted relation $[\![E]\!](D)$, as illustrated by Example 5.1.

Theorem 5.14. *A durable update spanner $[\![g]\!]$ with the replacement specifier U is pseudo-irrelevant with respect to an extractor $[\![E]\!]$ if $[\![E]\!]$ is independent of both $[\![g]\!]$ and $[\![\nabla(g, U)]\!]$ and $Repl(g, U)$ respects the alphabets in $C_v(E)$.*

Proof. Consider any document D with m regions marked by $[\![g]\!]$ to be updated, like s_2 and s_4 in Figure 3. Let D' denote D's updated peer, which has regions marked by $[\![\nabla(g, U)]\!]$, including those like s'_2 and s'_4 in Figure 3. Assume that some spans of D are marked by $[\![E]\!]$ to be extracted. To prove that an update is pseudo-irrelevant with respect to an extractor, we must prove three properties (as explained earlier):

(1) For every extracted region in D, there should be a corresponding region in D' that will be extracted, i.e., no span disappears from the extracted relation after the update. Since $[\![E]\!]$ is independent of $[\![g]\!]$, the only regions that can be updated are those matching $\hat{\Sigma}^*$, where $\hat{\Sigma} \subseteq \Sigma$. However, since $Repl(g, U)$ respects the alphabets in $C_v(E)$, any update expression matching $\hat{\Sigma}^*$ includes a replacement specification U that also matches $\hat{\Sigma}^*$. Thus every span found before the update must have a corresponding span found after the update.

(2) For every region extracted from D', there should be a corresponding region extracted from D, i.e., no new span appears in the extracted relation because of the update. Since $[\![E]\!]$ is independent of $[\![\nabla(g, U)]\!]$, the only regions that could have been updated are those matching $\hat{\Sigma}^*$, where $\hat{\Sigma} \subseteq \Sigma$. As before, since $Repl(g, U)$ respects the alphabets in $C_v(E)$, 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.

(3) For extracted spans, *ReplSpan* should be realized as a *shift* function. Because of independence, the lengths of the extracted regions remain the same after the update, since no update can occur inside an extracted region. But other regions preceding an extracted region and matching $\hat{\Sigma}^*$ can be updated, which makes the starting offset of an extracted region move back or 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. □

Algorithm 6: Verify pseudo-irrelevancy.

Input: extraction program P, update expression $Repl(g, U)$, update variable v
Output: Boolean
Precondition: $[\![g]\!]$ durable and each primitive extractor E_j in P has an exposed variable $\bar{v}_j$

```
/* verify only primitive extractors (i.e., formulas) E_j in P          */
forall E_j in P do
    forall v_i ∈ SVars(C_{v̄_j}(E_j)) do
        if depends(E_j, g, v_i, v) ≠ ∅
        or depends(E_j, ∇(g, U), v_i, v) ≠ ∅ then
            return False
        end
    end
    /* test for conflicting symbols (i.e., respecting alphabets)       */
    forall uncovered Σ̂* in E_i do
        A ← {σ | σ ∈ v and σ outside all capture variables
          back-referenced in U} ∪ {σ | σ ∈ U} ;
        if A ∩ (Σ \ Σ̂) ≠ ∅ then
            return False /* σ might be removed or added (Ex. 5.1) */
        end
    end
end
return True
```

Theorem 5.15. *Pseudo-irrelevancy 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 spanner $[\![g]\!]$, a replacement specification U, and an extraction formula E, Algorithm 6 verifies pseudo-irrelevancy in polynomial time. □

6 EXPERIMENTS

We wish to demonstrate that practical updates can be verified to be pseudo-irrelevant. We also wish to show that our optimization strategy is, in fact, practical in realistic extraction scenarios using real-world large datasets.

To this end, we have augmented Morciano's code [26] to build a system in Scala that verifies whether an update spanner is pseudo-irrelevant with respect to 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; otherwise, the extractor needs to be re-executed (from scratch or incrementally).

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 code and all performance details regarding the experiments can be found at https://github.com/Besatkassaie/Differential-Maintenance-Engine.

6.1 Corpora

Our experiments are performed over two corpora. The Blog Authorship Corpus [31] is a collection of unstructured blogs used by Morciano to test his spanners. This corpus has 19,320 documents (one per blogger) with 681,288 posts consuming 8.1 MB in total. DBLP [23], a bibliography of computer science literature, is an evolving semi-structured dataset with frequent updates to add new records, edit existing records, and delete records [9]. For this corpus, 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 see when looking up an author, but exporting XML for each record). This corpus has 3,091,270 documents (one per author) with 20,021,301 bibliographic entries consuming 7.9 GB in total.

6.2 Extractors

We have designed several realistic extractors (Tables 2 and 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 $\bigcup_{i \le d}(X\Sigma^i Y)$. Because the use of such operators will cause our verifier to deem all updates to be relevant (the replacement text may cause the string to exceed the maximum length), we replace them with $X(\Sigma - [?] - [.])^* Y$ to require instead that matches to X and Y appear in the same sentence.

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

character sets:

$\gamma_u = [\texttt{A}, \texttt{Z}]$ $\quad \gamma_l = [\texttt{a}, \texttt{z}]$ $\quad \gamma_d = [\texttt{0}, \texttt{9}]$ $\quad \gamma_w = (\gamma_u \vee \gamma_l \vee \gamma_d)$

$\gamma_p = (\texttt{:} \vee \texttt{,} \vee \texttt{;} \vee \texttt{!} \vee \texttt{.} \vee \texttt{?})$ $\quad \gamma_b = \texttt{␣} \vee \texttt{,}$

$\gamma_t = (\gamma_w \vee \gamma_p \vee \texttt{-})$ $\quad \gamma_s = (\Sigma - \texttt{?} - \texttt{.})$

basic patterns:

$\gamma_{name} = \gamma_u \gamma_l^* \texttt{␣} \gamma_u \gamma_l^* (\texttt{␣} \vee \gamma_p)$

$\gamma_{film} = ((\texttt{"}\gamma_w) \vee (\texttt{'}(\gamma_u \vee \gamma_d)))\ (\gamma_t^* \vee \gamma_t^* \texttt{␣} \gamma_t^* \vee \gamma_t^* \texttt{␣} \gamma_t^* \texttt{␣} \gamma_t^* \vee \gamma_t^* \texttt{␣} \gamma_t^* \texttt{␣} \gamma_t^* \texttt{␣} \gamma_t^* \texttt{␣} \gamma_t^* \texttt{␣} \gamma_t^*)(\texttt{'} \vee \texttt{"})$

Morciano's extractors use several small word lists (for which we use names starting with upper case letters), including for example $\gamma_{Genre} = \texttt{action} \vee \texttt{adventure} \vee \texttt{children} \vee \texttt{war} \cdots$ and $\gamma_{Movie} = \texttt{feature} \vee \texttt{dvd} \vee \texttt{film} \vee \texttt{movie} \vee \texttt{flick}$. To avoid having spurious matches to these words (such as matching `fractions` when intending to match `action`, or `aware` when intending to match `war`), our extractors include additional context to require that any match to a word from a word list must be preceded by a blank (␣) 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 2 and 3) that complement those for the other corpus, and are built on basic patterns for authors' names and titles:

$\gamma_{author} = \gamma_u(\gamma_u \vee \gamma_l \vee \texttt{-} \vee \texttt{␣} \vee \texttt{.␣})^*$

$\gamma_{title} = (\gamma_t \vee \texttt{␣})^*$

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

6.3 Updates

We have chosen four representative update scenarios, each using $Repl(\gamma_i, U_i)$ with update variable F.

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

Table 2: Primitive extractors.

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

Table 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

follows, where $\gamma_h = \gamma_l \vee \gamma_d$ (any lower case letter or digit):

$\gamma_{hashtag} = \Sigma^* F\{\texttt{\#} \gamma_h^* (\texttt{photo} \vee \texttt{selfie}) \gamma_h^*\} \gamma_b \Sigma^*$

$U_{hashtag} = \$(F) \texttt{␣(photography)}$

The verifier confirms that this update is pseudo-irrelevant 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:

$\gamma_{URL} = \Sigma^* F\{\texttt{http://} T\{(\Sigma - \texttt{<} - \texttt{>} - \texttt{␣} - \texttt{"} - \texttt{'})^*\}\}$
$(\texttt{<} \vee \texttt{>} \vee \texttt{␣} \vee \texttt{"} \vee \texttt{'}) \Sigma^*$

$U_{URL} = \texttt{<a␣href="} \$(F) \texttt{">} \$(T) \texttt{<a/>}$

This time, the verifier reports that the update is pseudo-irrelevant with respect to Q_1, Q_2, Q_{10}, and Q_{12} (and therefore also Q_{13}) and also that it is pseudo-irrelevant with respect to all the DBLP extractors. However, the verifier reports that the update is not pseudo-irrelevant 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 `␣actor␣...␣http://Brad␣Pitt␣...` to `␣actor␣...␣<a␣href="http://Brad">Brad</a>␣Pitt␣...`. 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:

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

$U_{date} = \$(D) \texttt{-} \$(M)$

This update is reported to be pseudo-irrelevant with respect to all the DBLP extractors. When tested against the primitive Blog extractors, it is found to be pseudo-irrelevant with respect to all but Q_1 and Q_{10}, where the string `mdate=` could immediately precede a film title beginning `"2015-01-03␣..."`. 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:

$\gamma_{DOI} = \Sigma^* (\texttt{<ee>} \vee \texttt{<ee␣type="} \gamma_l\gamma_l \texttt{">}) F\{\texttt{https://doi.org/}\} \Sigma^*$

$U_{DOI} = \texttt{doi:}$

Again, the update is found to be pseudo-irrelevant with respect to all the DBLP extractors. However, because `.` $\in F$, it is not pseudo-irrelevant 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 were the update to be reversed: the verifier detects `.` $\in U$, which might result in fewer matches after the update.)

6.4 Run-time performance

Table 4: Extraction times (×1000 sec) for DBLP corpus.

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

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. Table 4 shows how many thousands of seconds each extractor requires (based on measurements from

random samples), ranging from 2.3 hours to 43.4 hours and with the final extractor requiring more than 125 hours. 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. [6] 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.[7] Elsewhere, Shen et al. [32] 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 5 shows how many seconds are required for verifying pseudo-irrelevancy 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 pseudo-irrelevancy 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 5: 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

Chen et al. [6] 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 pseudo-irrelevance. An update might be provably pseudo-irrelevant 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 pseudo-irrelevant, 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 pseudo-irrelevant.

[7] In practice, document-at-a-time extraction can easily be parallelized, so a system with 32 processors could reduce overall elapsed time to 12 minutes or so.

7 ADDITIONAL RELATED WORK

Document Spanners. Researchers have addressed many problems using the document spanner model, including how to deal with documents with missing information [25] and how to eliminate inconsistencies from extracted relations [11]. Others have studied the complexity of evaluating spanners and computing the results of various algebraic operations over span relations [2, 13, 28, 29].

Static Analysis of Programs Using Regular Languages. We use an extended form of finite-state automata to determine whether an update expression is pseudo-irrelevant with respect to a document spanner. Similar static analyses of regular expressions have been used in diverse areas, including access control mechanisms for XML database systems [27] and conflicting gestures in multi-touch environments [22]. We have also used finite automata to statically analyze extractors specified by JAPE [8] in the context of updating extracted views [19].

Materialized View Maintenance. 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 used to maintain the content of a view without recomputing it from scratch. Blakeley et al. address this problem where the update involves insertion or deletion of a set of tuples into/from one base relation at a time [3, 4]. They give sufficient and necessary conditions on the irrelevancy of an update, which is stronger than our work proposing only sufficient conditions. For a differential view update, Blakeley et al. additionally require the content of associated base relations before the updates.

8 CONCLUSIONS

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 pseudo-irrelevancy of updates with respect to an extraction program: *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 to determine that the revised extracted relation can be computed autonomously. 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.

ACKNOWLEDGMENTS

We gratefully acknowledge financial assistance received from the University of Waterloo and NSERC, the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] James F. Allen. 1983. Maintaining Knowledge about Temporal Intervals. *Commun. ACM* 26, 11 (1983), 832–843.

[2] Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. 2019. Constant-Delay Enumeration for Nondeterministic Document Spanners. In *Proc. 22nd International Conference on Database Theory, ICDT (LIPIcs)*, Pablo Barceló and Marco Calautti (Eds.), Vol. 127. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Lisbon, 22:1–22:19.

[3] José A. Blakeley, Neil Coburn, and Per-Åke Larson. 1989. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. *ACM Trans. Database Syst.* 14, 3 (1989), 369–400.

[4] José A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. 1986. Efficiently Updating Materialized Views. In *Proc. 1986 ACM SIGMOD International Conference on Management of Data*, Carlo Zaniolo (Ed.). ACM Press, Washington, DC, 61–71.

[5] Amit Chandel, P. C. Nagesh, and Sunita Sarawagi. 2006. Efficient Batch Top-k Search for Dictionary-based Entity Recognition. In *Proc. 22nd International Conference on Data Engineering, ICDE*, Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang (Eds.). IEEE Computer Society, Atlanta, 28:1–28:10.

[6] Fei Chen, AnHai Doan, Jun Yang, and Raghu Ramakrishnan. 2008. Efficient Information Extraction over Evolving Text Data. In *Proc. 24th International Conference on Data Engineering, ICDE*. IEEE Computer Society, Cancún, 943–952.

[7] Rada Chirkova and Jun Yang. 2012. Materialized Views. *Found. Trends Databases* 4, 4 (2012), 295–405.

[8] Hamish Cunningham, Diana Maynard, and Valentin Tablan. 2000. *JAPE: a Java annotation patterns engine*. Technical Report CS-00-10. Dept. Comp. Sci., Univ. Sheffield. 28 pages.

[9] The dblp team. 2022. Statistics. Retrieved November 7, 2022 from https://dblp.org/statistics/index.html

[10] Johannes Doleschal, Benny Kimelfeld, Wim Martens, Yoav Nahshon, and Frank Neven. 2019. Split-Correctness in Information Extraction. In *Proc. 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS*, Dan Suciu, Sebastian Skritek, and Christoph Koch (Eds.). ACM, Amsterdam, 149–163.

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

[12] Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. 2015. Document Spanners: A Formal Approach to Information Extraction. *J. ACM* 62, 2 (2015), 12:1–12:51.

[13] Fernando Florenzano, Cristian Riveros, Martín Ugarte, Stijn Vansummeren, and Domagoj Vrgoc. 2020. Efficient Enumeration Algorithms for Regular Document Spanners. *ACM Trans. Database Syst.* 45, 1 (2020), 3:1–3:42.

[14] Dominik D. Freydenberger and Sam M. Thompson. 2020. Dynamic Complexity of Document Spanners. In *Proc. 23rd International Conference on Database Theory, ICDT (LIPIcs)*, Carsten Lutz and Jean Christoph Jung (Eds.), Vol. 155. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Copenhagen, 11:1–11:21.

[15] Robert J. Gaizauskas, Hamish Cunningham, Yorick Wilks, Peter J. Rodgers, and Kevin Humphreys. 1996. GATE: An Environment to Support Research and Development in Natural Language Engineering. In *Proc. Eighth International Conference on Tools with Artificial Intelligence, ICTAI*. IEEE Computer Society, Tolouse, 58–66.

[16] Ashish Gupta, H. V. Jagadish, and Inderpal Singh Mumick. 1996. Data Integration using Self-Maintainable Views. In *Proc. Advances in Database Technology - EDBT'96, 5th International Conference on Extending Database Technology (Lecture Notes in Computer Science)*, Peter M. G. Apers, Mokrane Bouzeghoub, and Georges Gardarin (Eds.), Vol. 1057. Springer, Avignon, 140–144.

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

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

[19] Besat Kassaie and Frank Wm. Tompa. 2019. Predictable and Consistent Information Extraction. In *Proc. DocEng '19: ACM Symposium on Document Engineering*. ACM, Berlin, 14:1–14:10.

[20] Besat Kassaie and Frank Wm. Tompa. 2020. Detecting Opportunities for Differential Maintenance of Extracted Views. (2020), 19 pages. arXiv:2007.01973

[21] Besat Kassaie and Frank Wm. Tompa. 2020. A Framework for Extracted View Maintenance. In *Proc. DocEng '20: ACM Symposium on Document Engineering*. ACM, Virtual Event, 16:1–16:4.

[22] Kenrick Kin, Björn Hartmann, Tony DeRose, and Maneesh Agrawala. 2012. Proton: multitouch gestures as regular expressions. In *ACM Conf. on Human Factors in Computing Systems, CHI '12*. ACM, Austin, 2885–2894.

[23] Michael Ley. 2009. DBLP - Some Lessons Learned. *Proc. VLDB Endow.* 2, 2 (2009), 1493–1500.

[24] Yunyao Li, Frederick Reiss, and Laura Chiticariu. 2011. SystemT: A Declarative Information Extraction System. In *Proc. 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies - System Demonstrations*. ACL, Portland, OR, 109–114.

[25] Francisco Maturana, Cristian Riveros, and Domagoj Vrgoc. 2018. Document Spanners for Extracting Incomplete Information: Expressiveness and Complexity. In *Proc. 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM, Houston TX USA, 125–136.

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

[27] Makoto Murata, Akihiko Tozawa, Michiharu Kudo, and Satoshi Hada. 2006. XML access control using static analysis. *ACM Trans. Inf. Syst. Secur.* 9, 3 (2006), 292–324.

[28] Liat Peterfreund, Dominik D. Freydenberger, Benny Kimelfeld, and Markus Kröll. 2019. Complexity Bounds for Relational Algebra over Document Spanners. In *Proc. 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS*, Dan Suciu, Sebastian Skritek, and Christoph Koch (Eds.). ACM, Amsterdam, 320–334.

[29] Liat Peterfreund, Balder ten Cate, Ronald Fagin, and Benny Kimelfeld. 2019. Recursive Programs for Document Spanners. In *Proc. 22nd International Conference on Database Theory, ICDT (LIPIcs)*, Pablo Barceló and Marco Calautti (Eds.), Vol. 127. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Lisbon, 13:1–13:18.

[30] Frederick Reiss, Sriram Raghavan, Rajasekar Krishnamurthy, Huaiyu Zhu, and Shivakumar Vaithyanathan. 2008. An Algebraic Approach to Rule-Based Information Extraction. In *Proc. 24th International Conference on Data Engineering, ICDE*. IEEE Computer Society, Cancún, 933–942.

[31] Jonathan Schler, Moshe Koppel, Shlomo Argamon, and James W. Pennebaker. 2006. Effects of Age and Gender on Blogging. In *Proc. 2006 AAAI Spring Symp. on Computational Approaches to Analyzing Weblogs*. AAAI, Stanford, 199–205.

[32] Warren Shen, AnHai Doan, Jeffrey F. Naughton, and Raghu Ramakrishnan. 2007. Declarative Information Extraction Using Datalog with Embedded Extraction Predicates. In *Proc. 33rd International Conference on Very Large Data Bases, VLDB*. ACM, Vienna, 1033–1044.

[33] Danny Sullivan. 2020. *A reintroduction to our Knowledge Graph and knowledge panels*. Google. Retrieved December 7, 2022 from https://blog.google/products/search/about-knowledge-graph-and-knowledge-panels/