


# Dynamic Complexity of Document Spanners

Dominik D. Freydenberger 

Loughborough University, Loughborough, United Kingdom

Sam M. Thompson 

Loughborough University, Loughborough, United Kingdom

---

## Abstract

---

The present paper investigates the dynamic complexity of document spanners, a formal framework for information extraction introduced by Fagin, Kimelfeld, Reiss, and Vansummeren (JACM 2015). We first look at the class of regular spanners and prove that any regular spanner can be maintained in the dynamic complexity class DynPROP. This result follows from work done previously on the dynamic complexity of formal languages by Gelade, Marquardt, and Schwentick (TOCL 2012).

To investigate core spanners we use SpLog, a concatenation logic that exactly captures core spanners. We show that the dynamic complexity class DynCQ is more expressive than SpLog and therefore can maintain any core spanner. This result is then extended to show that DynFO can maintain any generalized core spanner and that DynFO is more powerful than SpLog with negation.

**2012 ACM Subject Classification** Theory of computation → Complexity theory and logic; Information systems → Information extraction

**Keywords and phrases** Document spanners, information extraction, dynamic complexity, descriptive complexity, word equations

**Digital Object Identifier** 10.4230/LIPIcs.ICDT.2020.11

**Related Version** A full version of this paper is available at [9], <https://arxiv.org/abs/1909.10869>.

**Acknowledgements** The authors would like to thank the anonymous reviewers as well as Thomas Schwentick for their helpful comments and suggestions. The authors would also like to thank Thomas Zeume for clarifying a result from his thesis.

## 1 Introduction

Document spanners were introduced by Fagin, Kimelfeld, Reiss, and Vansummeren [4] as a formalization of IBM's information retrieval language AQL. Essentially, they can be explained as a formalism of querying text like one would query a relational database.

The universe of document spanners are *spans*, intervals of positions in a text. For example, if one searches for a word inside a larger text, every match can be understood as being one span inside the text. Spanners generalize this by mapping an input text to a table of spans.

More specifically, the process can be described as follows. First, *primitive spanners*, so-called *extractors*, are used to convert the input text into tables of spans. These extractors can be assumed to be *regex formulas*, which are regular expressions with variables. The tables can then be combined using relational algebra. As one might expect, different types of spanners allow different choices of operators. In this paper, we deal with three types of spanners that were introduced by Fagin et al. [4]. *Regular spanners*, currently the most widely studied in literature, allow the operators  $\cup$  (union),  $\pi$  (projection), and  $\bowtie$  (join). *Core spanners* extend regular spanners by allowing the string equality selection operator  $\xi^=$ , which allows one to check whether spans describe the same string (but potentially at different places). *Generalized core spanners* then extend these with the set difference  $\setminus$ .



© Dominik D. Freydenberger and Sam M. Thompson;  
licensed under Creative Commons License CC-BY

23rd International Conference on Database Theory (ICDT 2020).

Editors: Carsten Lutz and Jean Christoph Jung; Article No. 11; pp. 11:1–11:21



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In the last few years, various aspects of spanners have received considerable attention (see our related work section). The main focus was on evaluation and enumeration of results. But very few papers have considered aspects of maintaining the results of spanners under updates on the input text, and these have only focused on regular spanners.

In this paper, we examine the complexity of this problem from a *dynamic complexity* point of view. The classic dynamic complexity setting was independently introduced by Dong, Su, and Topor [3] and Patnaik and Immerman [16]. The “default setting” of dynamic complexity assumes a big relational database that is constantly changing (where the updates consist of adding or removing tuples from relations). The goal is then to maintain a set of auxiliary relations that can be updated with “little effort”. As this is a descriptive complexity point of view, little effort is defined as using only first-order formulas. The class of all problems that can be maintained in this way is called DynFO.

A more restricted setting is DynPROP, where only quantifier-free formulas can be used. As one might expect, restricting the update formulas leads to various classes between DynPROP and DynFO. Of particular interest to this paper are DynCQ and DynUCQ, where the update formulas are conjunctive queries or unions of conjunctive queries. As shown by Zeume and Schwentick [21], DynCQ = DynUCQ holds; but it is open whether these are proper subclasses of DynFO (see Zeume [20] for detailed background information).

As document spanners are defined on words, we adapt the dynamic complexity setting for formal languages by Gelade, Marquardt, and Schwentick [10]. This interprets a word structure as a linear order (of positions in the word) with unary predicates for every terminal symbol. To account for the dynamic complexity setting, positions can be undefined, and the update operations are setting a position to a symbol (an insertion or a symbol change) and resetting a position to undefined (deleting a symbol).

We show that in this setting, regular spanners can be maintained in DynPROP, core spanners in DynUCQ (and, hence, by [21] in DynCQ), and generalized core spanners in DynFO. Here, the second of these results is the main result of the present paper (the third follows directly from it, and the first almost immediately from [10]). To achieve it, we do not convert core spanners directly, but use the concatenation logic SpLog as an intermediate model.

SpLog (short for *spanner logic*) was introduced by Freydenberger [6] and has the same expressive power as core spanners (under some caveats that we discuss in Section 2.2). An additional benefit of the main result is that SpLog can be used to simplify proofs that languages or word relations can be maintained in DynCQ.

**Related work.** Recently, algorithmic and complexity theoretic aspects of evaluation and enumeration of spanners have received a considerable amount of attention, see [1, 5, 7, 8, 6, 12, 13, 14, 17, 18]. But these almost exclusively consider spanners in a static setting. To the authors’ knowledge, the only articles to also examine updates are Losemann [12] and Amarilli, Bourhis, Mengel, and Niewerth [1]. Both do not take a DynFO point of view; moreover, both only deal with regular spanners and there is no obvious way to also include the string equalities that are required for core spanners and generalized core spanners.

Doleschal, Kimelfeld, Martens, Nahshon, and Neven [2] introduce the notion of split-correctness. Without going into details, this examines spanners for which it is possible to split the input word into subwords on which the spanner is then evaluated. This can be viewed as a special case of update, but again was restricted to regular spanners.

Gelade, Marquardt, and Schwentick [10] examined the dynamic complexity of formal languages. Their result that DynPROP captures the regular languages is the basis for Proposition 3.1 in the current paper. While they also established that every context free language is in DynFO and that every Dyck-language is in DynQF (DynPROP with auxiliary functions), they did not examine DynUCQ and DynCQ, which the present paper does.

Muñoz, Vortmeier, and Zeume [15] studied the dynamic complexity in a graph database setting, namely for *conjunctive regular path queries* (CRQPs) and *extended conjunctive regular path queries* (ECRPQs). In particular, Theorem 14 in [15] states that on acyclic graphs, even a generalization of ECRPQs can be maintained in DynFO. Fagin et al. [4] established that on marked paths (a certain type of graph) core spanners have the same expressive powers as a CRPQs with string equalities (a fragment of ECRPQs). While marked paths are not acyclic in a strict sense, Section 7 of [6] proposes a variant of this model that could be directly combined with the construction from [15]. Thus, one could combine these results and observe that core spanners can be maintained in DynFO. In contrast to this, the present paper allows us to lower the upper bound to DynCQ. Moreover, if one is satisfied with DynUCQ, the constructions in the present paper also guarantee that all auxiliary relations only contain active nodes (nodes which carry a letter) of the word-structure, the only exception being the special case where the word-structure represents the empty string.

**Structure of the paper.** Section 2 contains the central definitions. Section 3 establishes dynamic upper bounds for the three central classes of document spanners (regular, core, and generalized core spanners), in particular the main result (Theorem 3.13). Section 4 further examines the relative expressive powers of core spanners and DynCQ. Section 5 concludes the paper. Some of the longer proofs have been omitted, see the full version for those proofs [9].

## 2 Preliminaries

Let  $\mathbb{N} := \{0, 1, 2, \dots\}$  and let  $\mathbb{N}_+ := \mathbb{N} \setminus \{0\}$ , where  $\setminus$  denotes set difference. We write  $|S|$  to represent the *cardinality* of a set  $S$ . We use  $\subseteq$  for subset and  $\subset$  for proper subset. We denote the powerset of  $S$  by  $\mathcal{P}(S)$ . Let  $\emptyset$  be the empty set. If  $R$  is a relation of arity 0, then  $R$  is the empty set, or  $R$  is the set containing the empty tuple. We define  $[n] := \{1, 2, \dots, n\}$ .

Let  $A$  be an alphabet<sup>1</sup>. We write  $|w|$  to denote the length of a word  $w \in A^*$ . The number of occurrences of some  $a \in A$  in a word  $w \in A^*$  is represented by  $|w|_a$ . We use  $\varepsilon$  to denote the empty word. Given two words  $u \in A^*$  and  $v \in A^*$ , we write  $u \cdot v$ , or simply  $uv$  for concatenation. If  $w = v_1uv_2$  where  $v_1 \in A^*$  and  $v_2 \in A^*$ , then  $u$  is a *subword* of  $w$ . We use  $\sqsubseteq$  for subword and  $\sqsubset$  for the proper subword relation. If  $u$  is not a subword of  $w$ , we write  $u \not\sqsubseteq w$ . Let  $\Sigma$  be a finite alphabet of so-called *terminal symbols*. Let  $\Xi$  be an infinite set of so-called *variables*, which is disjoint from  $\Sigma$ . Let  $\mathcal{L}(A)$  (or  $\mathcal{L}(\alpha)$ ) denote the language of a nondeterministic finite automaton (NFA)  $A$  (or of a regular expression  $\alpha$ ).

The rest of this section is structured as follows: First, we define various types of document spanners in Section 2.1 and equivalent logics (Section 2.2). After that, we define dynamic complexity, with a particular focus on its application to document spanners (Section 2.3).

<sup>1</sup> We use  $A$  here as a generic alphabet since we look at both the alphabet of terminal symbols and the alphabet of variables, and the concepts defined here apply to both.

## 2.1 Document Spanners and Spanner Algebra

In this section, we introduce document spanners and their representations. We begin with *primitive spanners* (Section 2.1.1) and then combine these to *spanner algebras* (Section 2.1.2).

### 2.1.1 Primitive Spanner Representations

Let  $w := a_1 \cdot a_2 \cdots a_n$  be a word, where  $n \geq 0$  and  $a_1, \dots, a_n \in \Sigma$ . A *span* of  $w$  is an interval  $[i, j)$  with  $1 \leq i \leq j \leq n + 1$  and  $i, j \geq 0$ . Given a span  $[i, j)$  of a word  $w$ , we define the subword  $w_{[i, j)}$  as  $a_i \cdot a_{i+1} \cdots a_{j-1}$ .

► **Example 2.1.** Consider the word  $w := \text{banana}$ . As  $|w| = 6$ , the spans of  $w$  are the  $[i, j)$  with  $1 \leq i \leq j \leq 7$ . For example, we have  $w_{[1, 2)} = \text{b}$  and  $w_{[2, 4)} = w_{[4, 6)} = \text{an}$ . Although both spans describe the same subword  $\text{an}$ , the two occurrences are at different locations (and, thus, at different spans). Analogously, we have  $w_{[1, 1)} = w_{[2, 2)} = \cdots = w_{[7, 7)} = \varepsilon$ , but  $[i, i) \neq [i', i')$  for all distinct  $1 \leq i, i' \leq 7$ .

Let  $V \subseteq \Xi$  and  $w \in \Sigma^*$ . A  $(V, w)$ -*tuple* is a function  $\mu$  that maps each  $x \in V$  to a span  $\mu(x)$  of  $w$ . A set of  $(V, w)$ -tuples is called a  $(V, w)$ -*relation*. A *spanner*  $P$  is a function that maps every  $w \in \Sigma^*$  to a  $(V, w)$ -relation  $P(w)$ . We write  $\text{SVars}(P)$  to denote the set of variables  $V$  of a spanner  $P$ . Two spanners  $P_1$  and  $P_2$  are *equivalent* if  $\text{SVars}(P_1) = \text{SVars}(P_2)$  and  $P_1(w) = P_2(w)$  holds for all  $w \in \Sigma^*$ .

In the usual applications of spans and spanners, the word  $w$  is some type of text. Hence, we can view a spanner  $P$  as mapping an input text  $w$  to a  $(V, w)$ -relation  $P(w)$ , which can be understood as a table of spans of  $w$ .

To define spanners, we use two types of *primitive spanner representations*, the so-called *regex formulas* and *variable-set automata*. Both extend classical mechanisms for regular languages (regular expressions and NFAs, respectively) with variables.

**Regex formulas:** The syntax of regex formulas is defined by the following  $\alpha := \emptyset \mid \varepsilon \mid a \mid (\alpha \vee \alpha) \mid (\alpha \cdot \alpha) \mid (\alpha)^* \mid x\{\alpha\}$ , where  $a \in \Sigma$  and  $x \in \Xi$ . We use  $\alpha^+$  to denote  $\alpha \cdot \alpha^*$ .

Like [6], we define the semantics of regex formulas using two step-semantics with *ref-words* (originally introduced by Schmid [19] in a different context). A ref-word is a word over the extended alphabet  $(\Sigma \cup \Gamma)$  where  $\Gamma := \{\vdash_x, \dashv_x \mid x \in \Xi\}$ . The symbols  $\vdash_x$  and  $\dashv_x$  represent the beginning and end of the span for the variable  $x$ . The first step in the definition of semantics is treating each regex formula  $\alpha$  as generators of languages of ref-words  $\mathcal{R}(\alpha) \subseteq (\Sigma \cup \Gamma)^*$ , which is defined by  $\mathcal{R}(\emptyset) := \emptyset$ ,  $\mathcal{R}(a) := \{a\}$  where  $a \in \Sigma \cup \{\varepsilon\}$ ,  $\mathcal{R}(\alpha_1 \vee \alpha_2) := \mathcal{R}(\alpha_1) \cup \mathcal{R}(\alpha_2)$ ,  $\mathcal{R}(\alpha_1 \cdot \alpha_2) := \mathcal{R}(\alpha_1) \cdot \mathcal{R}(\alpha_2)$ ,  $\mathcal{R}(\alpha^*) := \mathcal{R}(\alpha)^*$ , and  $\mathcal{R}(x\{\alpha\}) := \vdash_x \mathcal{R}(\alpha) \dashv_x$ .

Let  $\text{SVars}(\alpha)$  be the set of all  $x \in \Xi$  such that  $x\{\}$  occurs somewhere in  $\alpha$ . A ref-word  $r \in \mathcal{R}(\alpha)$  is *valid* if for all  $x \in \text{SVars}(\alpha)$ , we have that  $|r|_{\vdash_x} = 1$ . We denote the set of valid ref-words in  $\mathcal{R}(\alpha)$  as  $\text{Ref}(\alpha)$  and say that a regex formula is *functional* if  $\mathcal{R}(\alpha) = \text{Ref}(\alpha)$ . We write  $\text{RGX}$  for the set of all functional regex formulas. By definition, for every  $\alpha \in \text{RGX}$ , every  $r \in \text{Ref}(\alpha)$ , and every  $x \in \text{SVars}(\alpha)$ , there is a unique factorization  $r = r_1 \vdash_x r_2 \dashv_x r_3$ .

This allows us to define the second step of the semantics, which turns such a factorization for some variable  $x$  into a span  $\mu(x)$ . To this end, we define a morphism  $\text{clr}: (\Sigma \cup \Gamma)^* \rightarrow \Sigma^*$  by  $\text{clr}(a) := a$  for  $a \in \Sigma$  and  $\text{clr}(g) = \varepsilon$  for all  $g \in \Gamma$ . For a factorization  $r = r_1 \vdash_x r_2 \dashv_x r_3$ ,  $\text{clr}(r_1)$  is the substring of  $w$  that appears before  $\mu(x)$  and  $\text{clr}(r_2)$  is the substring  $w_{\mu(x)}$ .

We use this for the definition of the semantics as follows: For  $\alpha \in \text{RGX}$  and  $w \in \Sigma^*$ , let  $V := \text{SVars}(\alpha)$  and (more importantly)  $\text{Ref}(\alpha, w) := \{r \in \text{Ref}(\alpha) \mid \text{clr}(r) = w\}$ .

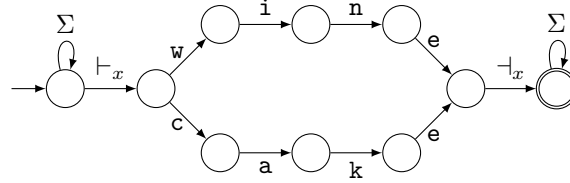
Every  $r \in \text{Ref}(\alpha, w)$  defines a  $(V, w)$ -tuple  $\mu^r$  in the following way: For every  $x \in \text{SVars}(\alpha)$ , we use the unique factorization  $r = r_1 \vdash_x r_2 \dashv_x r_3$  to define  $\mu^r(x) := [|\text{clr}(r_1)| + 1, |\text{clr}(r_1 r_2)| + 1]$ . The spanner  $\llbracket \alpha \rrbracket$  is then defined by  $\llbracket \alpha \rrbracket(w) := \{\mu^r \mid r \in \text{Ref}(\alpha, w)\}$  for all  $w \in \Sigma^*$ .

**Variable-set automata:** Variable-set automata (short: *vset-automata*) are NFAs that may use variable operations  $\vdash_x$  and  $\dashv_x$  as transitions. More formally, let  $V \subset \Xi$  be a finite set of variables. A variable-set automaton over  $\Sigma$  with variables  $V$  is a tuple  $A = (Q, q_0, q_f, \delta)$ , where  $Q$  is the set of states,  $q_0 \in Q$  is the initial state,  $q_f \in Q$  is the accepting state, and  $\delta: Q \times (\Sigma \cup \{\varepsilon\} \cup \Gamma_V) \rightarrow \mathcal{P}(Q)$  is the transition function with  $\Gamma_V := \{\vdash_x, \dashv_x \mid x \in V\}$ .

We define the semantics using a two-step approach analogous to the semantic definition of regex formulas. Firstly, we treat  $A$  as an NFA that defines the ref-language defined by  $\mathcal{R}(A) := \{r \in (\Sigma \cup \Gamma_V)^* \mid q_f \in \delta^*(q_0, r)\}$ , where the function  $\delta^*: Q \times (\Sigma \cup \Gamma_V) \rightarrow \mathcal{P}(Q)$  is defined such that for all  $p, q \in Q$  and  $r \in (\Sigma \cup \Gamma_V)^*$ ,  $q \in \delta^*(p, r)$  if and only if there exists a path in  $A$  from  $p$  to  $q$  with the label  $r$ .

Secondly, let  $\text{SVars}(A)$  be the set of  $x \in V$  such that  $\vdash_x$  or  $\dashv_x$  appears in  $A$ . A ref-word  $r \in \mathcal{R}(A)$  is *valid* if for every  $x \in \text{SVars}(A)$ ,  $|r|_{\vdash_x} = |r|_{\dashv_x} = 1$ , and  $\vdash_x$  always occurs to the left of  $\dashv_x$ . Then  $\text{Ref}(A)$ ,  $\text{Ref}(A, w)$  and  $\llbracket A \rrbracket$  are defined analogously to regex formulas. We denote the set of all vset-automata using  $\text{VA}_{\text{set}}$ . As for regex formulas, a vset-automaton  $A \in \text{VA}_{\text{set}}$  is called *functional* if  $\mathcal{R}(A) = \text{Ref}(A)$ .

► **Example 2.2.** We define the functional regex formula  $\alpha := \Sigma^* \cdot x\{(\text{wine}) \vee (\text{cake})\} \cdot \Sigma^*$ . We also define the functional vset-automaton  $A$  as follows:



For all  $w \in \Sigma^*$ , we have that  $\llbracket \alpha \rrbracket(w) = \llbracket A \rrbracket(w)$  contains exactly those  $(\{x\}, w)$ -tuples  $\mu$  that have  $w_{\mu(x)} = \text{wine}$  or  $w_{\mu(x)} = \text{cake}$ .

## 2.1.2 Spanner Algebra

We now introduce an algebra on spanners in order to construct more complex spanners.

► **Definition 2.3.** Two spanners  $P_1$  and  $P_2$  are compatible if  $\text{SVars}(P_1) = \text{SVars}(P_2)$ . We define the following algebraic operators for all spanners  $P, P_1, P_2$ :

- If  $P_1$  and  $P_2$  are compatible, their union  $(P_1 \cup P_2)$  and their difference  $(P_1 \setminus P_2)$  are defined by  $(P_1 \cup P_2)(w) := P_1(w) \cup P_2(w)$  and  $(P_1 \setminus P_2)(w) := P_1(w) \setminus P_2(w)$ .
- The projection  $\pi_Y P$  for  $Y \subseteq \text{SVars}(P)$  is defined by  $\pi_Y P(w) := P|_Y(w)$ , where  $P|_Y(w)$  is the restriction of all  $\mu \in P(w)$  to  $Y$ .
- The natural join  $P_1 \bowtie P_2$  is obtained by defining each  $(P_1 \bowtie P_2)(w)$  as the set of all  $(V_1 \cup V_2, w)$ -tuples  $\mu$  for which there exists  $\mu_1 \in P_1(w)$  and  $\mu_2 \in P_2(w)$  with  $\mu|_{V_1}(w) = \mu_1(w)$  and  $\mu|_{V_2}(w) = \mu_2(w)$ , where  $V_i := \text{SVars}(P_i)$  for  $i \in \{1, 2\}$ .
- For every  $k$ -ary relation  $R \subseteq (\Sigma^*)^k$  and variables  $x_1, \dots, x_k \in \text{SVars}(P)$ , the selection  $\xi_{x_1 \dots x_k}^R P$  is defined by  $\xi_{x_1 \dots x_k}^R P(w) := \{\mu \in P(w) \mid (w_{\mu(x_1)}, \dots, w_{\mu(x_k)}) \in R\}$  for  $w \in \Sigma^*$ . Let  $\text{SVars}(P_1 \cup P_2) := \text{SVars}(P_1 \setminus P_2) := \text{SVars}(P_1) = \text{SVars}(P_2)$ ,  $\text{SVars}(\pi_Y P) := Y$ ,  $\text{SVars}(P_1 \bowtie P_2) := \text{SVars}(P_1) \cup \text{SVars}(P_2)$ , and  $\text{SVars}(\xi_{x_1 \dots x_k}^R P) := \text{SVars}(P)$ .

Note that the relations  $R$  in the selection are usually infinite; and they are never considered part of the input.

Let  $O$  be a spanner algebra and let  $C$  be a class of primitive spanner representations, then we use  $C^O$  to denote the set of all spanner representations that can be constructed by repeated combinations of the symbols for the operators from  $O$  with the spanner representation from  $C$ . We denote the closure of  $\llbracket C \rrbracket$  under the spanner operators  $O$  as  $\llbracket C^O \rrbracket$ .

► **Example 2.4.** Let  $\alpha_1 := \Sigma^* x \{\Sigma^*\} \Sigma^* y \{\Sigma^*\} \Sigma^*$  and  $\alpha_2 := \Sigma^* \cdot x \{(\mathbf{wine}) \vee (\mathbf{cake})\} \cdot \Sigma^*$  (recall Example 2.2). We combine the two regex formulas into a core spanner  $P := \pi_x \xi_{x,y}^{\equiv} (\alpha_1 \bowtie \alpha_2)$ . Then  $\llbracket P \rrbracket(w)$  contains all  $(\{x\}, w)$ -tuples  $\mu$  such that  $w_{\mu(x)}$  is an occurrence of **wine** or **cake** in  $w$  that is followed by another occurrence of the same word.

Like Fagin et al. [4], we are mostly concerned with string equality selections  $\xi^{\equiv}$ . Following [4, 18], we focus on the class of *regular spanners*  $\llbracket \text{RGX}^{\text{reg}} \rrbracket$ , the class of *core spanners*<sup>2</sup>  $\llbracket \text{RGX}^{\text{core}} \rrbracket$  and the class of *generalized core spanners*  $\llbracket \text{RGX}^{\text{core} \cup \{\setminus\}} \rrbracket$ , where  $\text{reg} := \{\pi, \cup, \bowtie\}$  and  $\text{core} := \{\pi, \xi^{\equiv}, \cup, \bowtie\}$ . As shown in [4], we have

$$\llbracket \text{RGX}^{\text{reg}} \rrbracket = \llbracket \text{VA}_{\text{set}}^{\text{reg}} \rrbracket = \llbracket \text{VA}_{\text{set}} \rrbracket \subset \llbracket \text{RGX}^{\text{core}} \rrbracket = \llbracket \text{VA}_{\text{set}}^{\text{core}} \rrbracket \subset \llbracket \text{RGX}^{\text{core} \cup \{\setminus\}} \rrbracket = \llbracket \text{VA}_{\text{set}}^{\text{core} \cup \{\setminus\}} \rrbracket.$$

In other words, there is a proper hierarchy of regular, core, and generalized core spanners; and for each of the classes, we can choose regex formulas or vset-automata as primitive spanner representations. As shown in [6], functional vset-automata have the same expressive power as vset-automata in general. The size difference can be exponential, but this does not matter for the purpose of the present paper.

## 2.2 Spanner Logic

In this section, we define **SpLog** (spanner logic) and relate it to spanners. **SpLog** is a fragment of  $\text{EC}^{\text{reg}}$ , the existential theory of concatenation with regular constraints (a logic that is built around the concatenation operator). It was introduced by Freydenberger [6] and has the same expressive power as core spanners; and conversions between both models are possible in polynomial time. To define **SpLog**, we first introduce *word equations*.

A *pattern*  $\alpha$  is a word from  $(\Sigma \cup \Xi)^*$ . In other words, patterns may contain variables and terminal symbol. A *word equation* is a pair of patterns  $(\eta_L, \eta_R)$ , which are called the *left* and *right* side of the equation, respectively. We usually write a word equation as  $\eta_L \doteq \eta_R$ . The set of all variables in a pattern  $\alpha$  is denoted by  $\text{var}(\alpha)$ . This is extended to word equations  $\eta = (\eta_L, \eta_R)$  by  $\text{var}(\eta) := \text{var}(\eta_L) \cup \text{var}(\eta_R)$ .

A *pattern substitution* is a morphism  $\sigma : (\Sigma \cup \Xi)^* \rightarrow \Sigma^*$  such that  $\sigma(a) = a$  holds for all  $a \in \Sigma$ . As every substitution  $\sigma$  is a morphism, we have  $\sigma(\alpha_1 \cdot \alpha_2) = \sigma(\alpha_1) \cdot \sigma(\alpha_2)$  for all patterns  $\alpha_1$  and  $\alpha_2$ . Hence, to define  $\sigma$ , it suffices to define  $\sigma(x) \in \Sigma^*$  for all  $x \in \Xi$ .

The main idea of **SpLog** is choosing a special main variable  $W$  that shall correspond to the input string of a spanner. **SpLog** is then an existential-positive logic over words, where the atoms are regular predicates or word equations of the form  $W \doteq \eta_R$ . Formally, we define syntax and semantics as follows:

► **Definition 2.5.** Let  $W \in \Xi$ . Then  $\text{SpLog}(W)$ , the set of all *SpLog*-formulas with main variable  $W$ , is defined recursively as containing the following formulas:

**B1.**  $(W \doteq \eta_R)$  for every  $\eta_R \in (\Xi \cup \Sigma)^*$ .

<sup>2</sup> As this class captures the core functionality of SystemT.

- R1.**  $(\varphi_1 \wedge \varphi_2)$  for all  $\varphi_1, \varphi_2 \in \text{SpLog}(W)$ .  
**R2.**  $(\varphi_1 \vee \varphi_2)$  for all  $\varphi_1, \varphi_2 \in \text{SpLog}(W)$  with  $\text{free}(\varphi_1) = \text{free}(\varphi_2)$ .  
**R3.**  $\exists x: \varphi$  for all  $\varphi \in \text{SpLog}(W)$  and  $x \in \text{free}(\varphi) \setminus \{W\}$ .  
**R4.**  $(\varphi \wedge C_A(x))$  for every  $\varphi \in \text{SpLog}(W)$ , every  $x \in \text{free}(\varphi)$ , and every NFA  $A$ .

Let  $\text{free}(\varphi)$  be  $\text{free}(\eta) := \text{var}(\eta)$ ,  $\text{free}(\varphi_1 \wedge \varphi_2) := \text{free}(\varphi_1 \vee \varphi_2) := \text{free}(\varphi_1) \cup \text{free}(\varphi_2)$ ,  $\text{free}(\exists x: \varphi) := \text{free}(\varphi) \setminus \{x\}$ , and  $\text{free}(\varphi \wedge C_A(x)) := \text{free}(\varphi)$ .

For every pattern substitution  $\sigma$  and every  $\varphi \in \text{SpLog}(W)$ , we define  $\sigma \models \varphi$  as follows:

- $\sigma \models (W \doteq \eta_R)$  if  $\sigma(W) = \sigma(\eta_R)$ ,
- $\sigma \models (\varphi_1 \wedge \varphi_2)$  if  $\sigma \models \varphi_1$  and  $\sigma \models \varphi_2$ ; and  $\sigma \models (\varphi_1 \vee \varphi_2)$  is defined analogously,
- $\sigma \models \exists x: \varphi$  if  $\sigma_{\frac{x}{w}} \models \varphi$  for some  $w \in \Sigma^*$ , where  $\sigma_{\frac{x}{w}}(x) := w$  and  $\sigma_{\frac{x}{w}}(y) = \sigma(y)$  if  $y \neq x$ ,
- $\sigma \models (\varphi \wedge C_A(x))$  if  $\sigma \models \varphi$  and  $\sigma(x) \in \mathcal{L}(A)$ .

Let  $\text{SpLog}$  be the union of all  $\text{SpLog}(W)$  with  $W \in \Xi$ . We add and omit parentheses, as long as the meaning remains unambiguous. We also allow constraints of the form  $C_\alpha(x)$ , where  $\alpha$  is a regular expression. For readability, we use  $\varphi(W; x_1, x_2 \dots x_k)$  to express that the  $\text{SpLog}$ -formula  $\varphi$  has the main variable  $W$  and free variables  $\{x_1, x_2 \dots x_k\}$ . As a convention, assume that no word equation  $(W \doteq \eta_R)$  has the main variable  $W$  occur in the right side; that is, that  $|\eta_R|_W = 0$  holds.

► **Example 2.6.** For the  $\text{SpLog}$ -formula  $\varphi(W) := \exists x: ((W \doteq xxx) \wedge C_{\text{ab}^*}(x))$ , we have  $\sigma \models \varphi$  if and only if  $\sigma(W) = w w w$  for some  $w \in \text{ab}^*$ .

We also extend the definition of  $\text{SpLog}$  to  $\text{SpLog}^\neg$ , which we call  $\text{SpLog}$  with negation.

► **Definition 2.7.** Let  $W \in \Xi$ . Then  $\text{SpLog}^\neg(W)$ , the set of  $\text{SpLog}^\neg$ -formulas with the main variable  $W$ , is defined by extending Definition 2.5 with the additional rule that if  $\varphi \in \text{SpLog}^\neg(W)$ , then  $(\neg\varphi) \in \text{SpLog}^\neg(W)$ , with  $\text{free}(\varphi) = \text{free}(\neg\varphi)$ . We define  $\sigma \models \neg\varphi$  as:

- $\sigma(x) \sqsubseteq \sigma(W)$  for all  $x \in \text{free}(\varphi)$ , and
- $\sigma \models \varphi$  does not hold.

To compare the expressive power of  $\text{SpLog}$  and document spanners, we need to overcome the difficulty that the former reasons about words, while the latter reason over positions in an input word. To this end, we use the following notion that was introduced by Freydenberger and Holldack [7] in the context of  $\text{EC}^{\text{reg}}$ .

► **Definition 2.8.** Let  $\varphi \in \text{SpLog}$  with  $\text{free}(\varphi) := \{W\} \cup \{x_p, x_c \mid x \in \text{SVars}(P)\}$ . Let  $P$  be a spanner. Let  $\llbracket \varphi \rrbracket(w)$  denote the set of all  $\sigma$  such that  $\sigma \models \varphi$  and  $\sigma(W) = w$ . We then say that  $\varphi$  realizes  $P$  if for all  $w \in \Sigma^*$ , we have  $\sigma \in \llbracket \varphi \rrbracket(w)$  if and only if  $\mu \in P(w)$  where for each  $x \in \text{SVars}(P)$  and  $[i, j] := \mu(x)$ , both  $\sigma(x_p) = w_{[1, i]}$  and  $\sigma(x_c) = w_{[i, j]}$ .

Intuitively, this definition uses two main ideas: Firstly, the spanner's input word  $w$  is represented by the main variable  $W$ . Secondly, every spanner variable  $x$  is represented by two  $\text{SpLog}$ -variables  $x_p$  and  $x_c$ , such that in each  $(V, w)$ -tuple  $\mu$ , we have that  $x_c$  contains the actual content  $w_{\mu(x)}$  and  $x_p$  contains the prefix of  $w$  before the start of  $\mu(x)$ .

As shown in Section 4.1 of [6], under this lens,  $\text{SpLog}$  has exactly the same expressive power as  $\llbracket \text{RGX}^{\text{core}} \rrbracket$  (the core spanners), and  $\text{SpLog}^\neg$  exactly the same as  $\llbracket \text{RGX}^{\text{core} \cup \{\setminus\}} \rrbracket$  (the generalized core spanners).

One of the central questions in [4, 6] is which relations  $R$  can be added to spanners or  $\text{SpLog}$  without increasing the expressive power (using  $\xi^R$  or a new constraint symbol for  $R$ , respectively). This is reflected in the notion of *selectable relations*. A relation  $R \subseteq (\Sigma^*)^k$  is called *SpLog-selectable* if for every  $\varphi \in \text{SpLog}(W)$  and every sequence  $\vec{x} = (x_1, \dots, x_k)$  of

variables with  $x_1, \dots, x_k \in \text{free}(\varphi) \setminus \{W\}$ , there is a SpLog-formula  $\varphi_{\vec{x}}^R$  with  $\text{free}(\varphi) = \text{free}(\varphi_{\vec{x}}^R)$ , and  $\sigma \models \varphi_{\vec{x}}^R$  if and only if  $\sigma \models \varphi$  and  $(\sigma(x_1), \dots, \sigma(x_k)) \in R$ . This is equivalent to the analogously defined notion of core spanner selectable relations, see Section 5.1 of [6] for details. We shall use selectability both in the way to our main result (namely, in Lemma 3.12) and for further observations in Section 4.

### 2.3 Dynamic Complexity

Our definitions of dynamic complexity are based on the setting of dynamic formal languages as described by Gelade, Marquardt, and Schwentick [10]. In this setting, strings are modeled by a relational structure. Insertions and deletions of symbols can be performed on this structure and (auxiliary) relations are *maintained* by logic formulas, called *update formulas*. We extend this with a predetermined relation which is maintained to hold the result of some spanner performed on the current word. The idea of dynamic complexity, which was introduced by Patnaik and Immerman [16], is to have dynamic descriptive complexity classes based upon the logic needed to maintain a relation, or in our case a spanner. We now formally define these concepts.

Let  $\Sigma$  be a fixed and finite alphabet of terminal symbols. We represent words using a *word-structure*. A word-structure has a fixed and finite set known as the domain  $D := [n + 1]$  as well as a 2-ary order relation  $<$  on  $D$ . We use the shorthands  $x \leq y$  for  $(x < y) \vee (x \doteq y)$ . We have in our word-structure the constant  $\$$  which is interpreted by the element  $n + 1$ , the  $<$ -maximal element of  $D$ . This  $<$ -maximal element marks the end of the word structure and is required for dynamic spanners, which are defined later. For each symbol  $\zeta \in \Sigma$  the word-structure has a unary relation  $R_\zeta(i)$  and there is at *most* one  $\zeta \in \Sigma$  such that  $R_\zeta(i)$  for  $i \in [n]$ . If we have  $R_\zeta(i)$  then we write  $w(i) = \zeta$ , otherwise we write  $w(i) = \varepsilon$ . If  $w(i) \neq \varepsilon$  for some  $i \in D$ , then we call  $i$  a *symbol-element*.

Given a word-structure  $\mathcal{W}$ , the word that  $\mathcal{W}$  represents is denoted by  $\text{word}(\mathcal{W})$  and this is defined as  $\text{word}(\mathcal{W}) := w(1) \cdot w(2) \cdot \dots \cdot w(n)$ . Since for some  $j \in D$  it could be that  $w(j) = \varepsilon$ , it follows that the length of the word  $\text{word}(\mathcal{W})$  is likely to be less than  $n$ . Let  $w := \text{word}(\mathcal{W})$ , we write  $w[i, j]$  to represent the subword  $w[i, j] := w(i) \cdot w(i+1) \cdot \dots \cdot w(j)$  where  $i, j \in D$  such that  $i < j$ .

We now define the set of *abstract updates*  $\Delta := \{\text{ins}_\zeta \mid \zeta \in \Sigma\} \cup \{\text{reset}\}$ . A *concrete update* is  $\text{ins}_\zeta(i)$  or  $\text{reset}(i)$ , for some  $i \in D \setminus \{\$\}$  and  $\zeta \in \Sigma$ . The difference between abstract updates and concrete updates is that concrete updates can be *performed* on a word-structure. Given a word-structure with a domain of size  $n$ , we use  $\Delta_n$  to represent the set of possible concrete updates. For some  $\partial \in \Delta_n$ , we denote the word-structure  $\mathcal{W}$  after an update is performed by  $\partial(\mathcal{W})$  and this is defined as:

- If  $\partial = \text{ins}_\zeta(i)$ , then  $R_\zeta(i)$  is true and  $R_{\zeta'}(i)$  is false for all  $\zeta' \in \Sigma$  where  $\zeta \neq \zeta'$ .
- If  $\partial = \text{reset}(i)$  then  $R_\zeta(i)$  is false for all  $\zeta \in \Sigma$ .

All other elements keep the symbol they had before the update. For  $k \geq 1$ , let  $\partial^* := \partial_1, \partial_2, \dots, \partial_k$  be a sequence of updates. We use  $\partial^*(\mathcal{W})$  as a short hand to represent  $\partial_k(\dots(\partial_2(\partial_1(\mathcal{W})))\dots)$ . We place the restriction that updates must *change* the string. We do not allow  $\text{reset}(i)$  if  $w(i) = \varepsilon$  and we do not allow  $\text{ins}_\zeta(i)$  if  $w(i) = \zeta$ .

► **Example 2.9.** Given a word-structure  $\mathcal{W}$  over the alphabet  $\Sigma := \{a, b\}$  with domain  $D = [6]$ , where  $6 = \$$ . If we have that  $R_a = \{2, 4\}$  and  $R_b := \{5\}$ , it follows that  $\text{word}(\mathcal{W}) = aab$ . Performing the operation  $\text{ins}_b(1)$  would give us an updated word of  $baab$ . Say if we then perform  $\text{reset}(4)$  on our new word structure, we would have the word  $bab$ .



We define the *auxiliary structure*  $\mathcal{W}_{aux}$  as a set of relations over the domain of  $\mathcal{W}$ . A *program state*  $\mathcal{S} := (\mathcal{W}, \mathcal{W}_{aux})$  is a word-structure and an auxiliary structure. An *update program*  $\vec{P}$  is a finite set of update formulas, which are of the form  $\phi_{op}^R(y; x_1, \dots, x_k)$ . We have an update formula for each  $R \in \mathcal{W}_{aux}$  and  $op \in \Delta$ . An update,  $op(i)$ , performed on  $\mathcal{S}$  yields  $\mathcal{S}' = (\partial(\mathcal{W}), \mathcal{W}'_{aux})$  where all relations  $R' \in \mathcal{W}'_{aux}$  are defined by  $R' := \{\vec{j} \mid \vec{S} \models \phi_{op}^R(i; \vec{j})\}$ , where  $\vec{j}$  is a  $k$ -tuple (where  $k$  is the arity of  $R$ ) and where  $\vec{S} := (\partial(\mathcal{W}), \mathcal{W}_{aux})$ .

We use  $w$  to denote  $\text{word}(\mathcal{W})$  for some word structure  $\mathcal{W}$  and we use  $w'$  for  $\text{word}(\partial(\mathcal{W}))$  where  $\partial \in \Delta_n$  is some update performed on  $\mathcal{W}$ .

Given some  $x \in D$  where  $w(x) \neq \varepsilon$ , we write that  $\text{pos}_w(x) = 1$  if for all  $x' \in D$  where  $x' < x$  we have that  $w(x') = \varepsilon$ . Let  $z, y$  be elements from the domain such that  $z < y$  and  $w(z) \neq \varepsilon$  and  $w(y) \neq \varepsilon$ . If for all  $x \in D$  where  $z < x < y$  we have that  $w(x) = \varepsilon$  then  $\text{pos}_w(y) = \text{pos}_w(z) + 1$ . We write  $x \rightsquigarrow_w y$  if and only if  $\text{pos}_w(y) = \text{pos}_w(x) + 1$ . If it is not the case that  $x \rightsquigarrow_w y$  then we write  $x \not\rightsquigarrow_w y$ .

For every spanner  $P$  with  $\text{SVars}(P) := \{x_1, x_2 \dots x_k\}$  and every word-structure  $\mathcal{W}$ , the spanner relation  $R^P$  is a  $2k$ -ary relation over  $D$  where each spanner variable  $x_i$  is represented by two components  $x_i^o$  and  $x_i^c$ . We obtain  $R^P$  on  $\mathcal{W}$  by converting each  $\mu \in P(w)$  into a  $2k$ -tuple  $(x_1^o, x_1^c, x_2^o, x_2^c \dots x_k^o, x_k^c)$ , where for each  $i \in [k]$ , we have  $\mu(x_i) = [\text{pos}_w(x_i^o), \text{pos}_w(x_i^c)]$ . The only exception is if  $\mu(x_i) = [j, k]$  and  $k > |w|$  then  $x_i^c = \$$  for such a tuple  $(x_1^o, x_1^c, x_2^o, x_2^c \dots x_k^o, x_k^c)$ . In Example 2.11 we give a spanner represented by a regex formula and show the corresponding spanner-relation on a word-structure.

► **Definition 2.10.** A *dynamic program* is a triple, containing:

- $\vec{P}$  - an update program over  $(\mathcal{W}, \mathcal{W}_{aux})$ .
- INIT - a first-order initialization program.
- $R^P \in \mathcal{W}_{aux}$  - a designated spanner-relation.

For each  $R \in \mathcal{W}_{aux}$ , we have some  $\psi_R(\vec{j}) \in \text{INIT}$  which defines the initial tuples of  $R$  (before any updates to the input structure occur). Note that  $\vec{j}$  is a  $k$ -tuple where the arity of  $R$  is  $k$ . For our work  $\psi_R$  is a first-order logic formula.

A dynamic program *maintains* a spanner  $P$  if we have that  $R^P \in \mathcal{W}_{aux}$  always corresponds to  $P(\partial^*(\mathcal{W}))$ . We can then extend this to saying that we maintain a *relation* if there is a designated  $R \in \mathcal{W}_{aux}$  which is always equivalent to some relation where the relation is defined in terms of the input word.

► **Example 2.11.** Consider the regex formula  $\alpha := \Sigma^* \cdot x\{a \cdot b\} \cdot \Sigma^*$  where  $a, b \in \Sigma$  and  $x \in \Xi$ . Now consider the following word-structure:

1	2	3	4	5	6	\$
$a$	$\varepsilon$	$b$	$\varepsilon$	$a$	$\varepsilon$	$\varepsilon$

Note that the top row is the elements of the domain in order, and the bottom row is the corresponding symbols. If we maintain the *spanner relation* of  $\alpha$ , given the word-structure above, we have the relation  $R^P \in \mathcal{W}_{aux}$  such that  $R^P := \{(1, 5)\}$ . Now assume we perform the update  $\text{ins}_i(6)$ . The word-structure is now in the following state:

1	2	3	4	5	6	\$
$a$	$\varepsilon$	$b$	$\varepsilon$	$a$	$b$	$\varepsilon$

It must be that  $\phi_{\text{ins}_i}^{R^P}(6; x, y)$  updates the relation  $R^P$  to  $\{(1, 5), (5, \$)\}$  for us to correctly maintain the spanner.

► **Definition 2.12.** *DynFO is the class of all relations which can be maintained by update formulas which are defined using first-order logic. DynPROP is a subclass of DynFO where all the update formulas are quantifier-free.*

A first-order formula is a *conjunctive query*, or CQ for short, if it is built up from atomic formulae, conjunction and existential quantification. We also have unions of conjunctive queries, or UCQ for short, which allows for the finite disjunction of conjunctive queries. We therefore have the classes DynCQ and DynUCQ which use conjunctive queries and unions of conjunctive queries as update formulas respectively.

For this work, we assume that the input structure is initially empty and that every auxiliary relation is initialized by some first-order initialization. This is to allow us to use the result from Zeume and Schwentick [21] that  $\text{DynUCQ} = \text{DynCQ}$ . However, in our work we only require a very weak form of initialization and hence if DynUCQ is sufficient, one could define the precise class needed for the precomputation. We do not do this as the dynamic complexity class needed to *maintain* a spanner is the main focus of this work<sup>3</sup>.

For the proofs in the present paper, one could change the setting by allowing the insertion of unmarked nodes at any point of the word-structure (with an update to the  $<$ -relation), given that the word is non-empty. The auxiliary relations in our proofs do not operate on unmarked nodes and do not need to be updated after this. In the same way, we can remove unmarked nodes. However, the present paper does not look at this setting.

### 3 Core Spanners are in DynCQ

In this section, we first look at the dynamic complexity of regular spanners. We show that any regular spanner can be maintained by a DynPROP program. We then turn our attention to the main result of this paper, that any core spanner can be maintained by a DynCQ program. In doing so, we also show that DynCQ is at least as expressive as SpLog. We then extend this result to show that DynFO is at least as powerful as SpLog with negation, and therefore any generalized core spanner can be maintained in DynFO.

► **Proposition 3.1.** *Regular spanners can be maintained in DynPROP.*

**Proof.** Due to the work done by Fagin et al. [4] we can assume that our vset-automaton is a so called *vset-path union*. We define a vset-path as an ordered sequence of regular deterministic finite automata  $A_1, A_2, \dots, A_n$  for some  $n \in \mathbb{N}$ . Each automaton  $A_i$  is of the form  $(Q, q_0, F, \delta)$  where  $Q$  is the set of states,  $q_0 \in Q$  is the initial state,  $F$  is the set of accepting states, and  $\delta$  is the transition function of the form  $\delta: Q \times \Sigma \rightarrow Q$ . We have the extra assumption that each  $f \in F$  only has incoming transitions. All automata,  $A_1, A_2, \dots, A_n$  share the same set of input symbols  $\Sigma$ .

Let  $A$  be a vset-path. In  $A$ , each automata  $A_i$  where  $1 < i \leq n$ , the initial state for  $A_i$  has incoming transitions from each accepting state from the automaton  $A_{i-1}$ . These extra transitions between the sequence of automata are labeled,  $\vdash_x$  or  $\dashv_x$  where  $x \in \text{SVars}(A)$ . We treat the vset-path as a regular vset-automaton and all semantics follow from the definitions in Section 2.1.1. We can assume that  $A$  is functional [6].

Any vset-automaton can be represented as a union of vset-paths [4]. Therefore to prove that any regular spanner can be maintained in DynPROP, it is sufficient to prove that we can maintain a spanner represented by a vset-path, since union can be simulated via disjunction.

<sup>3</sup> As helpfully pointed out by one of the anonymous reviewers of this paper.

Let  $A$  be a vset-path. From Gelade et al. [10], we know that the following relations can be maintained in DynPROP:

- For any pair of states  $p, q \in Q$ ,  $R_{p,q} := \{(i, j) \mid i < j \text{ and } \delta^*(p, w[i+1, j-1]) = q\}$ .
- For each state  $q$ ,  $R_q^I := \{i \mid \delta^*(q_0, w[1, j-1]) = q\}$ .
- For each state  $p$ ,  $R_p^F := \{j \mid \delta^*(p, [i+1, n]) \in F\}$ .

We maintain these relations for the vset-path. Some work is needed to deal with the transitions labeled  $\vdash_x$  and  $\dashv_x$ . Let  $A_i$  and  $A_{i+1}$  be two sub-automata such that  $1 \leq i < n$ , where  $n$  is the number of sub-automata. Let  $s_i$  and  $s_{i+1}$  be the starting states for automata  $A_i$  and  $A_{i+1}$  respectively. Likewise, let  $F_i$  and  $F_{i+1}$  be the sets of accepting states of  $A_i$  and  $A_{i+1}$  respectively. The intuition is that if  $R_{p,f_i}(x, y)$  where  $f_i \in F_i$  holds, then so should  $R_{p,s_{i+1}}(x, y)$  since the transition from an accepting state of  $A_i$  to the starting state of  $A_{i+1}$  is  $\vdash_x$  or  $\dashv_x$ . To achieve this, we have the following update formula for  $R_{p,s_{i+1}}$

$$\phi_{\partial}^{R_{p,s_{i+1}}}(u; x, y) := \bigvee_{f \in F_i} \phi_{\partial}^{R_{p,f}}(u; x, y).$$

We do the analogous for  $R_q^I$  and  $R_p^F$ . If  $R_{f_i}^I(x)$  holds for any  $f_i \in F_i$ , then so should  $R_{s_{i+1}}^I(x)$ . Similarly, if  $R_{s_{i+1}}^F(x)$  holds, then so should  $R_{f_i}^F(x)$  for all  $f_i \in F_i$ . To achieve this, we proceed analogously to what was done for  $\phi_{\partial}^{R_{p,s_{i+1}}}(u; x, y)$ . We also maintain the 0-ary relation ACC to say whether the word-structure is a member of the language of the vset-path.

We will now give two useful subformulas

$$\psi^{k'} := \bigwedge_{1 \leq i \leq k'} \left( \bigvee_{\zeta \in \Sigma} (R_{\zeta}(x_i^o) \wedge R'_{s_i}(x_i^o) \wedge \bigvee_{\substack{p \in Q, \\ \delta(s_i, \zeta) = p}} (R'_{p,s_{i+1}}(x_i^o, x_i^c) \wedge \bigvee_{\zeta_2 \in \Sigma} R_{\zeta_2}(x_i^c))) \right)$$

and

$$\psi^{\$} := \bigvee_{\zeta \in \Sigma} (R_{\zeta}(x_k^o) \wedge R'_{s_k}(x_k^o) \wedge R'_{F_k}(x_k^c) \wedge (x_k^c \doteq \$)).$$

We now give the update formula to maintain a vset-path spanner  $A$  with variables  $\text{SVars}(A) := \{x_1, x_2, \dots, x_k\}$

$$\phi_{\partial}^{R^A}(u; x_1^o, x_1^c, \dots, x_k^o, x_k^c) := \phi_{\partial}^{\text{ACC}}(u) \wedge (\psi^k \vee (\psi^{k-1} \wedge \psi^{\$})).$$

Note that, without loss of generality,  $R'_{p,q}(x, y)$  is used as a shorthand for  $\phi_{\partial}^{R_{p,q}}(u; x, y)$ . ◀

Since Gelade et al. [10] proved that DynPROP maintains exactly the regular languages, it is somewhat unsurprising that we can extend that result to regular spanners. Some work is needed in order to maintain the relation of the spanner, which is why a formal proof of Proposition 3.1 is given.

► **Definition 3.2.** *The next symbol relation is defined as  $R_{\text{Next}} := \{(x, y) \in D^2 \mid x \rightsquigarrow_w y\}$ .*

As stated in Section 2.3, it is known that DynCQ = DynUCQ and therefore to show that a relation can be maintained in DynCQ, it is sufficient to show that the relation can be maintained with UCQ update formulas. We use this to prove many of our results.

► **Lemma 3.3.** *The next symbol relation can be maintained in DynCQ.*

## 11:12 Dynamic Complexity of Document Spanners

To prove Lemma 3.3, we maintain the relations  $R_{\text{first}} := \{x \in D \mid \text{pos}_w(x) = 1\}$  and  $R_{\text{last}} := \{x \in D \mid \text{pos}_w(y) = |w|\}$ . Note that these relations would be undefined for an empty input structure (because  $\text{pos}_w(x)$  is undefined). Hence we have that if  $|w| = 0$  then  $x \in R_{\text{first}}$  if and only if  $x = \$$ , and  $y \in R_{\text{last}}$  if and only if  $y$  is the  $<$ -minimal element. This requires the initialization of  $R_{\text{first}} := \{\$\}$  and  $R_{\text{last}} := \{1\}$ . This is the only initialization required in our work, however the stated *first-order initialization* of auxiliary relations is needed to ensure  $\text{DynUCQ} = \text{DynCQ}$ .

► **Example 3.4.** Consider the following word-structure:

1	2	3	4	5	6	\$
$\varepsilon$	$a$	$b$	$\varepsilon$	$b$	$\varepsilon$	$\varepsilon$

We have that  $R_{\text{first}} = \{2\}$  and  $R_{\text{last}} = \{5\}$  and  $R_{\text{Next}} = \{(2, 3), (3, 5)\}$ .

We will now give an idea for the proof of Lemma 3.3. Let  $u$  be the node which is being updated. For insertion, if  $x \rightsquigarrow_w y$  and  $x < u < y$  then  $x \rightsquigarrow_{w'} u \rightsquigarrow_{w'} y$ . If  $R_{\text{first}}(x)$  and  $u < x$ , then  $R'_{\text{first}}(u)$  and  $u \rightsquigarrow_{w'} x$ . The analogous is done if  $R_{\text{last}}(x)$  and  $u > x$ . For deletion, if  $x \rightsquigarrow_w u \rightsquigarrow_w y$  then  $x \rightsquigarrow_{w'} y$ . The full proof also looks at when  $x \rightsquigarrow_w y$  and  $x \rightsquigarrow_{w'} y$  (for example when  $u < x$  or when  $u > y$ ). See the full version of the paper for the proof [9].

► **Definition 3.5.** *The equal substring relation,  $R_{\text{eq}}$ , is the set of 4-tuples  $(x_o, x_c, y_o, y_c)$  such that  $w[x_o, x_c] = w[y_o, y_c]$ ,  $x_c < y_o$ , and  $w[z] \neq \varepsilon$  for all  $z \in \{x_o, x_c, y_o, y_c\}$ .*

Less formally, we have that if  $(x_o, x_c, y_o, y_c) \in R_{\text{eq}}$  then the word  $w[x_o, x_c]$  is equal to the word  $w[y_o, y_c]$ . For our uses, we do not want these subwords to overlap, hence the constraint  $x_c < y_o$ . The reason for this will become clear later on when we look at maintaining pattern languages. We also wish that each tuple represents a unique pair of subwords, therefore we have that  $x_o, x_c, y_o$ , and  $y_c$  each have symbols associated to them.

► **Example 3.6.** Consider the following word-structure:

1	2	3	4	5	6	7	8	9	10	\$
$a$	$\varepsilon$	$\varepsilon$	$b$	$a$	$\varepsilon$	$b$	$\varepsilon$	$a$	$b$	$\varepsilon$

The equal substring relation for this structure is  $R_{\text{eq}} = \{(1, 1, 5, 5), (1, 1, 9, 9), (4, 4, 7, 7), (4, 4, 10, 10), (5, 5, 9, 9), (7, 7, 10, 10), (1, 4, 5, 7), (1, 4, 9, 10), (4, 5, 7, 9), (5, 7, 9, 10)\}$ .

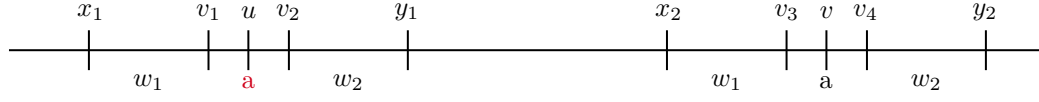
Although  $w[3, 5] = w[7, 9]$ , this does not imply  $(3, 5, 7, 9) \in R_{\text{eq}}$  because  $w[3] = \varepsilon$ . We also do not have  $(9, 10, 5, 7) \in R_{\text{eq}}$  because  $10 > 5$ .

► **Lemma 3.7.** *The equal substring relation can be maintained in DynCQ.*

We now give a proof idea for Lemma 3.7. There are four main cases for the tuple  $(x_1, y_1, x_2, y_2)$  we examine in the full proof.

- Case 1:  $w[x_1, y_1] = w[x_2, y_2]$  and  $w'[x_1, y_1] \neq w'[x_2, y_2]$ .
- Case 2:  $w[x_1, y_1] = w[x_2, y_2]$  and  $w'[x_1, y_1] = w'[x_2, y_2]$ .
- Case 3:  $w[x_1, y_1] \neq w[x_2, y_2]$  and  $w'[x_1, y_1] = w'[x_2, y_2]$ .
- Case 4:  $w[x_1, y_1] \neq w[x_2, y_2]$  and  $w'[x_1, y_1] \neq w'[x_2, y_2]$ .

Where we assume that  $y_1 < x_2$ . One can see that the main case out of these four is Case 3. One of the interesting sub-cases of Case 3 is illustrated in Figure 1. Here, one can think of the new symbol at node  $u$  as a “bridge” between the two equal substrings  $w[x_1, v_1]$  and  $w[x_2, v_3]$  (which are the word  $w_1$ ) and the equal substrings  $w[v_2, y_1]$  and  $w[v_4, y_2]$  (which are the word  $w_2$ ). Hence, after the update we have that  $w'[x_1, y_1] = w'[x_2, y_2]$  even though



■ **Figure 1** Word after the insertion of the symbol  $a$  at node  $u$ .

$w[x_1, y_1] \neq w[x_2, y_2]$  (under the assumptions that  $w(v) = a$ ,  $v_1 \rightsquigarrow_{w'} u \rightsquigarrow_{w'} v_2$  and that  $v_3 \rightsquigarrow_{w'} v \rightsquigarrow_{w'} v_4$ ). After examining a case like this, one would need to write an update formula to realize it.

The proof of Lemma 3.7 looks through all the cases and produces a UCQ update formula for each. These subformulae are joined together by disjunction to give us an update formula  $\phi_{\partial}^{R_{\text{eq}}}(u)$  which is in DynUCQ, and hence we have proven that we can maintain the equal substrings relation in DynCQ. See the full version for the proof [9].

Lemma 3.7 is a central part of the proof of our main result, and some may consider maintaining this relation also to be the most technical aspect of the present paper. This relation will be the main feature of a construction to maintain so-called *pattern languages*, which we then extend with regular constraints to maintain any relations selectable by SpLog.

Given a pattern  $\alpha \in (\Sigma \cup \Xi)^+$ , we define the *non-erasing* language it generates as  $\mathcal{L}_{\text{NE}, \Sigma}(\alpha) := \{\sigma(\alpha) \mid \sigma: (\Sigma \cup \Xi)^+ \rightarrow \Sigma^+ \text{ where } \sigma \text{ is a substitution}\}$ . Given the same pattern  $\alpha$ , we have  $\mathcal{L}_{\text{E}, \Sigma}(\alpha) := \{\sigma(\alpha) \mid \sigma: (\Sigma \cup \Xi)^+ \rightarrow \Sigma^* \text{ where } \sigma \text{ is a substitution}\}$  which is the *erasing* language  $\alpha$  generates. Pattern languages are not only used as a part of word equations but also as language generators (see [7] for more details, in particular regarding their relation to document spanners).

► **Example 3.8.** Consider  $\alpha := axxb$  where  $a, b \in \Sigma$  and  $x \in \Xi$ . Then  $ab \in \mathcal{L}_{\text{E}, \Sigma}(\alpha)$  with  $\sigma(x) = \varepsilon$ , but  $ab \notin \mathcal{L}_{\text{NE}, \Sigma}(\alpha)$ . We can also see that  $ababab \in \mathcal{L}_{\text{NE}, \Sigma}(\alpha)$  and  $ababab \in \mathcal{L}_{\text{E}, \Sigma}(\alpha)$  using  $\sigma(x) = ba$ .

We take the definition of maintaining a language from [10]. We can *maintain a language*  $L$  if a dynamic program maintains a 0-ary relation which is true if and only if  $\text{word}(\mathcal{W}) \in L$ .

► **Lemma 3.9.** *Every non-erasing pattern language can be maintained in DynCQ.*

**Proof.** To prove this lemma, we give a way to symbolically construct an update formula to maintain a 0-ary relation  $\mathcal{P}$  which updates to true if and only if  $w' \in \mathcal{L}_{\text{NE}, \Sigma}(\alpha)$  for any specified  $\alpha \in (\Sigma \cup \Xi)^+$ . Let  $|\alpha|$  be the length of the pattern  $\alpha$ . Let  $\alpha_i$  denote the  $i^{\text{th}}$  symbol (from  $\Xi$  or  $\Sigma$ ) of the pattern  $\alpha$  where  $1 \leq i \leq |\alpha|$ . We give the construction in Algorithm 1.

Note that occurrences of  $R'_{\text{Next}}$  and  $R'_{\text{eq}}$  in Algorithm 1 are the relations correct *after* the update. To achieve this, we can replace occurrences of  $R'_{\text{Next}}(\dots)$  with  $\phi_{\partial}^{R_{\text{Next}}}(\dots)$ , where  $\partial$  is the update for which the update formula of  $\mathcal{P}$  is being constructed. The equivalent is done for  $R_{\text{eq}}$ . ◀

► **Example 3.10.** Let  $\alpha := axbx$  be a pattern such that  $a, b \in \Sigma$  and  $x \in \Xi$ . As stated, we wish to maintain a 0-ary relation  $\mathcal{P}$  such that  $\mathcal{P}$  is true if and only if  $w' \in \mathcal{L}_{\text{NE}, \Sigma}(\alpha)$  where  $w'$  is our word after some update.

- $\alpha_1 = a$ : therefore  $\alpha_1 \in \Sigma$  and hence we have  $\omega_1 := R_a(t_1) \wedge R'_{\text{first}}(t_1)$ .
- $\alpha_2 = x$ : therefore  $\alpha_2 \in \Xi$  therefore we have  $\omega_2 := R'_{\text{Next}}(t_1, x_2) \wedge (x_2 \leq t_2) \wedge \omega_1$ .
- $\alpha_3 = b$ : therefore  $\alpha_3 \in \Sigma$  and hence we have  $\omega_3 := R_b(t_3) \wedge R'_{\text{Next}}(t_2, t_3) \wedge \omega_2$ .
- $\alpha_4 = x$  and  $\alpha_4 = \alpha_2$ : therefore  $\omega_4 := R'_{\text{Next}}(t_3, x_4) \wedge (x_4 \leq t_4) \wedge R'_{\text{eq}}(x_2, t_2, x_4, t_4) \wedge \omega_3$ .

---

**Algorithm 1** Pattern Language Update Formula Construction.

---

**Input:** A pattern  $\alpha \in (\Sigma \cup X)^+$ .

**Output:** Update formulas  $\phi_{\text{ins}_c}^{\mathcal{P}}(u)$  and  $\phi_{\text{reset}}^{\mathcal{P}}(u)$ .

If  $\alpha_1 \in \Sigma$  then  $\omega_1 := R_{\alpha_1}(t_1) \wedge R'_{\text{first}}(t_1)$ ;

If  $\alpha_1 \in \Xi$  then  $\omega_1 := (x_1 \leq t_1) \wedge R'_{\text{first}}(x_1)$ ;

**for**  $i := 2$  **to**  $|\alpha|$  **do**

**if**  $\alpha_i \in \Sigma$  **then**

$\omega_i := R_{\alpha_i}(t_i) \wedge R'_{\text{Next}}(t_{i-1}, t_i) \wedge \omega_{i-1}$ ;

**if**  $\alpha_i \in \Xi$  **then**

**if** *there exists*  $j \in \mathbb{N}$  *where*  $j < i$  *such that*  $\alpha_i = \alpha_j$  **then**

$j_{\text{max}} :=$  Largest  $j$  value such that  $j < i$  and  $\alpha_i = \alpha_j$ ;

$\omega_i := R'_{\text{Next}}(t_{i-1}, x_i) \wedge (x_i \leq t_i) \wedge R'_{\text{eq}}(x_{j_{\text{max}}}, t_{j_{\text{max}}}, x_i, t_i) \wedge \omega_{i-1}$ ;

**else**

$\omega_i := R'_{\text{Next}}(t_{i-1}, x_i) \wedge (x_i \leq t_i) \wedge \omega_{i-1}$ ;

$\omega := (\omega_{|\alpha|} \wedge R'_{\text{last}}(t_{|\alpha|}))$ ;

For every occurrence of some  $t_i$  in  $\omega$ , where  $i \leq |\alpha|$ , add  $\exists t_i$  to the front of  $\omega$ ;

For every occurrence of some  $x_i$  in  $\omega$  add  $\exists x_i$  to the front of  $\omega$ ;

$\phi_{\text{ins}_c}^{\mathcal{P}}(u) := \omega$ ;     $\phi_{\text{reset}}^{\mathcal{P}}(u) := \omega$ ;

---

We rearrange the atoms in  $\omega$  to help with readability, giving us:

$$\begin{aligned} \omega := & R'_{\text{first}}(t_1) \wedge R_a(t_1) \wedge R'_{\text{Next}}(t_1, x_2) \wedge (x_2 \leq t_2) \wedge R'_{\text{Next}}(t_2, t_3) \wedge R_b(t_3) \\ & \wedge R'_{\text{Next}}(t_3, x_4) \wedge (x_4 \leq t_4) \wedge R'_{\text{eq}}(x_2, t_2, x_4, t_4) \wedge R'_{\text{last}}(t_4). \end{aligned}$$

Hence  $\phi_{\mathcal{D}}^{\mathcal{P}}(u) := \exists t_1, t_2, t_3, t_4, x_2, x_4 : (\omega)$  which holds for a word-structure of the form:

$$\begin{array}{cccccccc} \dots & t_1 & \mathbf{x_2} & \dots & \mathbf{t_2} & t_3 & \mathbf{x_4} & \dots & \mathbf{t_4} & \dots \\ \varepsilon & a & & \dots & & b & & \dots & & \varepsilon \end{array}$$

We have that  $x_2, t_2, x_4, t_4$  are in bold to demonstrate the fact that it must be that  $w'[x_2, t_2] = w'[x_4, t_4]$  for  $\phi_{\mathcal{D}}^{\mathcal{P}}(u)$  to hold. Note that  $t_1$  may not be  $< -\text{minimal}$  and  $t_4$  may not be  $< -\text{maximal}$ , but because  $R'_{\text{first}}(t_1)$  and  $R'_{\text{last}}(t_4)$  must hold,  $t_1$  and  $t_4$  are the first and last symbol-elements respectively.

One side effect of Lemma 3.9 is that we get the dynamic complexity upper bounds of a class of languages, the pattern languages. Pattern languages were not looked at in [10] and hence this result extends what is known about the dynamic complexity of formal languages.

► **Corollary 3.11.** *Every erasing pattern language can be maintained in DynCQ.*

**Proof.** From Jiang et al. [11] it is known that every erasing pattern language is the finite union of non-erasing pattern languages. Therefore, we can create 0-ary relations for each non-erasing pattern language and join them with a disjunction. There is the case where  $\varepsilon \in \mathcal{L}_{\Xi, \Sigma}(\alpha)$  which we can deal with using the following:  $\exists x : (R_{\text{first}}(x) \wedge (x \doteq \$))$ . We can do this because  $R_{\text{first}} = \{\$\}$  whenever  $w = \varepsilon$ . ◀

Since we are able to maintain any erasing pattern language in DynCQ, we can extend this result to word-equations in SpLog-formulas. Using this along with the fact that regular languages can be maintained in DynPROP, we can conclude the following:

► **Lemma 3.12.** *Any relation selectable in SpLog can be maintained in DynCQ.*

**Proof.** We prove this lemma using structural induction with the recursive definition of a SpLog formula, given in Definition 2.5.

**B1.** ( $W \doteq \eta_R$ ) for every  $\eta_R \in (\Xi \cup \Sigma)^*$ : Since we are assuming that  $\sigma(W) \in \Sigma^*$  and that  $\eta_R$  does not contain  $W$ , we have that  $W \doteq \eta_R$  is equivalent to  $\sigma(W) \in \mathcal{L}_{\Xi, \Sigma}(\eta_R)$ . We have proven in Corollary 3.11, that we can maintain a 0-ary relation which is true if and only if, given some pattern  $\alpha \in (\Xi \cup \Sigma)^*$ , the word structure is currently a member of  $\mathcal{L}_{\Xi, \Sigma}(\alpha)$ . According to the construction which we gave in Lemma 3.9, given a variable  $x \in \Xi$ , where  $x = \alpha_i$ , we have two variables  $x_i, t_i \in D$  such that the word  $w[x_i, t_i]$  represents  $\sigma(x)$  for some substitution  $\sigma$ . Removing the existential quantifiers for  $x_i$  and  $t_i$  allows us to maintain the relation defined by  $\alpha$ .

**R1.** ( $\psi_1 \wedge \psi_2$ ) for all  $\psi_1, \psi_2 \in \text{SpLog}(W)$ : Under the assumption that we have update formulas  $\phi_{\delta}^{\psi_1}(u; \vec{v}_1)$  and  $\phi_{\delta}^{\psi_2}(u; \vec{v}_2)$  for SpLog formulas  $\psi_1$  and  $\psi_2$  respectively, the update formula for  $\phi_{\delta}^{\psi_1 \wedge \psi_2}(u; \vec{v}_1 \cup \vec{v}_2)$  is  $\phi_{\delta}^{\psi_1}(u; \vec{v}_1) \wedge \phi_{\delta}^{\psi_2}(u; \vec{v}_2)$ .

**R2.** ( $\psi_1 \vee \psi_2$ ) for all  $\psi_1, \psi_2 \in \text{SpLog}(W)$  with  $\text{free}(\psi_1) = \text{free}(\psi_2)$ : Assuming we have update formulas  $\phi_{\delta}^{\psi_1}(u; \vec{v})$  and  $\phi_{\delta}^{\psi_2}(u; \vec{v})$  for SpLog formulas  $\psi_1$  and  $\psi_2$  respectively, the update formula for  $\phi_{\delta}^{(\psi_1 \vee \psi_2)}(u; \vec{v})$  is  $\phi_{\delta}^{\psi_1}(u; \vec{v}) \vee \phi_{\delta}^{\psi_2}(u; \vec{v})$ .

**R3.**  $\exists x: \psi$  for all  $\psi \in \text{SpLog}(W)$  and  $x \in \text{free}(\psi) \setminus \{W\}$ : If a variable  $x \in \Xi$  is existentially quantified within the SpLog formula, then we existentially quantify the variables  $x_i, t_i \in D$  where  $w[x_i, t_i]$  represents  $\sigma(x)$  for some substitution  $\sigma$ .

**R4.** ( $\psi \wedge C_A(x)$ ) for every  $\psi \in \text{SpLog}(W)$ , every  $x \in \text{free}(\psi)$ , and every NFA  $A$ : let  $A := (Q, \delta, s, F)$  be an NFA. We have that  $Q$  is a finite set of states,  $\delta: Q \times \Sigma \rightarrow Q$  is the transition function,  $s$  is the initial state and  $F \subseteq Q$  is the set of accepting states. We denote the reflexive and transitive closure of  $\delta$  as  $\delta^*: Q \times \Sigma^* \rightarrow Q$ . For regular constraints, we maintain the relation  $R_A := \{(i, j) \in D^2 \mid w[i, j] \in \mathcal{L}(A)\}$

From Proposition 3.3 in Gelade, Marquardt, and Schwentick [10], we know that the following relations can be maintained in DynPROP, and from [20] (Theorem 3.1.5, part b) we know that DynPROP is a strict subclass of DynCQ. Hence we can maintain the following in DynCQ:

$$\begin{aligned} R_{p,q} &:= \{(i, j) \in D^2 \mid i < j \text{ and } \delta^*(p, w[i+1, j-1]) = q\}, \\ I_q &:= \{j \in D \mid \delta^*(s, w[1, j-1]) = q\}, \\ F_p &:= \{i \in D \mid \delta^*(p, w[i+1, n]) \in F\}. \end{aligned}$$

Where  $p, q \in Q$ . We also know, from [10], that we can maintain the 0-ary relation ACC, which is true if and only if  $w' \in \mathcal{L}(A)$ .

We maintain  $R_A$  with  $\phi_{\delta}^{R_A}(u; x, y) := \psi_1^{R_A} \vee \psi_2^{R_A} \vee \psi_3^{R_A} \vee \psi_4^{R_A}$  where each  $\psi_i^{R_A}$  is a subformula which we now define for separate cases. Note that  $R'(\vec{x})$  is shorthand for  $\phi_{\delta}^R(u; \vec{x})$ . We define  $\psi_1^{R_A}$  as

$$\psi_1^{R_A} := \exists x_2, y_2: (R'_{\text{Next}}(x_2, x) \wedge R'_{\text{Next}}(y, y_1) \wedge \bigvee_{f \in F} (R'_{s,f}(x_2, y_2)).$$

Since  $R_{p,q}(x, y)$  refers to the substring from position  $x+1$  to  $y-1$ , and we wish to examine the string from position  $x$  to  $y$ , we look at  $R'_{s,f}(x_2, y_2)$  where  $x_2 \rightsquigarrow_{w'} x$  and  $y \rightsquigarrow_{w'} y_2$ . If it is indeed the case that  $x_2 \rightsquigarrow_{w'} x$  and  $y \rightsquigarrow_{w'} y_2$  then  $w'[x_2+1, y_2-1] = w[x, y]$ . Therefore  $R'_{s,f}(x_2, y_2)$ , for  $f \in F$ , is true for such  $x_2$  and  $y_2$  if and only if  $\delta^*(s, w[x, y]) \in F$  which is the desired behavior for this case. Note that  $\psi_1^{R_A}$  fails if there doesn't exist  $x_2$  such that

## 11:16 Dynamic Complexity of Document Spanners

$x_2 \rightsquigarrow_{w'} x$  or there doesn't exist  $y_2$  such that  $y \rightsquigarrow_{w'} y_2$ . This is dealt with using  $\psi_2^{RA}$ ,  $\psi_3^{RA}$  and  $\psi_4^{RA}$ , which we explore next.

If  $R'_{\text{last}}(y)$  then  $w'[x, y] = w'[x, n]$  where  $n = |D|$ . Therefore, we can use  $F'_s(x_2)$  for some  $x_2 \in D$  where  $x_2 \rightsquigarrow_{w'} x$  and  $s$  is the initial state of the NFA, to see whether  $\delta^*(s, w'[x, n]) \in F$  and hence whether  $\delta^*(s, w'[x, y]) \in F$ . To realize this behavior, we define  $\psi_2^{RA}$  as

$$\psi_2^{RA} := \exists x_2 : (R'_{\text{Next}}(x_2, x) \wedge R'_{\text{last}}(y) \wedge F'_s(x_2)).$$

If  $R'_{\text{first}}(x)$  then  $w'[1, y] = w'[x, y]$ . Therefore, we can use  $I'_f(y_2)$  for some  $y_2 \in D$  where  $y \rightsquigarrow_{w'} y_2$  and  $f \in F$ , to see whether  $\delta^*(s, w'[1, y]) \in F$  and hence whether  $\delta^*(s, w'[x, y]) \in F$ . To realize this behavior, we define  $\psi_3^{RA}$  as

$$\psi_3^{RA} := \exists y_2 : (R'_{\text{Next}}(y, y_2) \wedge R'_{\text{first}}(x) \wedge \bigvee_{f \in F} (I'_f(y_2))).$$

If  $R'_{\text{first}}(x)$  and  $R'_{\text{last}}(y)$  then  $w'[x, y] = w'$  and therefore it follows that  $w'[x, y] \in \mathcal{L}(A)$  if and only if  $w' \in \mathcal{L}(A)$ . We only need to see if  $\text{ACC}'$  is true for this case. We realize this behavior by defining  $\psi_4^{RA}$  as

$$\psi_4^{RA} := R'_{\text{first}}(x) \wedge R'_{\text{last}}(y) \wedge \text{ACC}'.$$

To simulate  $(\psi \wedge C_A(x))$  for every  $\psi \in \text{SpLog}(\mathbf{W})$ , every  $x \in \text{free}(\psi)$ , and every NFA  $A$  within DynCQ, we do the following; let  $\phi_{\partial}^{\psi}(u; \vec{v})$  be an update formula for  $\psi \in \text{SpLog}$  and since for some  $\sigma(x)$ , where  $x \in \text{free}(\psi)$ , has  $x_i, t_i \in D$  associated with it, we can use  $\phi_{\partial}^{\psi}(u; \vec{v}) \wedge \phi_{\partial}^{RA}(u; x_i, t_i)$  which is true if and only if  $w'[x_i, t_i] \in \mathcal{L}(A)$ . ◀

Most of the work for this proof follows from Lemma 3.9 and Corollary 3.11. Extra work is done in order to simulate regular constraints, although this follows on from the fact that DynPROP maintains the regular languages [10].

► **Theorem 3.13.** *Core spanners can be maintained in DynCQ.*

**Proof.** Although maintaining the SpLog relation that realizes a spanner is not the same as maintaining the spanner relation as defined in Section 2.3, the changes we need to make are trivial. Let  $P$  be a spanner and let  $\psi_P$  be a SpLog formula that realizes  $P$ . We know that  $\text{free}(\psi_P) = \{x_p, x_c \mid x \in \text{SVars}(P)\}$ , and for every  $x \in \text{SVars}(P)$  where  $[i, j] := \mu(x)$ , we have both  $\sigma(x_p) = w_{[1, i]}$  and  $\sigma(x_c) = w_{[i, j]}$ . Let  $R^P$  be a relation that maintains the spanner  $P$ . The only difference between update formulas that maintain  $P$  and update formulas that maintain the relation SpLog selects which realizes  $P$  is that the two elements  $x_p^o, x_p^c \in D$  that are used to represent the SpLog variable  $x_p \in \Xi$  are existentially quantified whereas the two variables  $x_c^o, x_c^c \in D$  which represent  $x_c \in \Xi$  are not. ◀

Theorem 3.13 shows us that DynCQ is at least as expressive as SpLog. We will use this along with Proposition 4.1 to show that DynCQ is more expressive than core spanners. Given that we can maintain any relation selectable in SpLog using DynCQ, it is no big surprise that adding negation allows us to maintain  $\text{SpLog}^{\neg}$  in DynFO.

► **Lemma 3.14.** *Any relation selectable in  $\text{SpLog}^{\neg}$  can be maintained in DynFO.*

**Proof.** Let  $\psi \in \text{SpLog}(\mathbf{W})$  and let  $R^{\psi}$  be the relation maintaining  $\psi$  where the update formulas for  $R^{\psi}$  are in CQ. The extra recursive rule allowing for  $(\neg\psi) \in \text{SpLog}^{\neg}(\mathbf{W})$  can be maintained by  $\phi_{\partial}^{R^{\neg\psi}}(u; \vec{x}) = \neg\phi_{\partial}^{R^{\psi}}(u; \vec{x})$ . ◀



As with Theorem 3.13, we can use the result from Lemma 3.14 along with Corollary 4.2 to show that DynFO is more expressive than  $\text{SpLog}^\neg$ .

► **Theorem 3.15.** *Generalized core spanners can be maintained in DynFO.*

Since  $\text{SpLog}^\neg$  captures the generalized core spanners, it follows from Lemma 3.14 that any generalized core spanner can be maintained in DynFO. In Section 4 we show that DynFO is more expressive than  $\text{SpLog}^\neg$ , it therefore follows that DynFO is more expressive than generalized core spanners.

## 4 Relations in SpLog and DynCQ

In this section, we examine the comparative expressive power of SpLog and DynCQ. Recall that we defined the notion of SpLog-selectable relations at the end of Section 2.2. We now define an analogous concept for DynCQ. For a relation  $R \subseteq (\Sigma^*)^k$ , we define the corresponding relation in the dynamic setting  $\bar{R}$  as the  $2k$ -ary relation of all  $(x_1, y_1, \dots, x_k, y_k) \in D^{2k}$  such that  $(w[x_1, y_1], \dots, w[x_k, y_k]) \in R$ . We say that  $R$  is selectable in DynCQ if  $\bar{R}$  can be maintained in DynCQ.

For example, the equal length relation is defined as  $R_{\text{len}} := \{(w_1, w_2) \mid |w_1| = |w_2|\}$ . From Fagin et al. [4] it is known that this relation is not selectable with core spanners. This relation in the dynamic setting is  $\bar{R}_{\text{len}} = \{(u_1, u_2, v_1, v_2) \in D^4 \mid |w[u_1, u_2]| = |w[v_1, v_2]|\}$ .

► **Proposition 4.1.** *The equal length relation is selectable in DynCQ.*

**Proof.** To maintain the equal length relation, we take the update formulas from Lemma 3.7 and omit any atoms relating to the symbol of an element of the domain  $D$ . We also remove the constraint that the first subword must appear before the second. We also use  $\bar{R}_{\text{len}}$  in any update formula, rather than  $R_{\text{eq}}$ . The only exception to omitting all atoms relating to the symbol of an element, is to ensure that  $w[u_1] \neq \varepsilon$ ,  $w[u_2] \neq \varepsilon$ ,  $w[v_1] \neq \varepsilon$ , and  $w[v_2] \neq \varepsilon$ . ◀

While this allows us to separate the languages that are definable in SpLog from the ones that can be maintained in DynCQ, we consider the following more wide-ranging example:

► **Lemma 4.2.** *The language  $\{w \in \Sigma^* \mid |w| = 2^n, n \geq 0\}$  is maintainable in DynCQ.*

**Proof.** Let  $P$  be a 2-ary relation such that  $P(x, y)$  holds if and only if  $|w[x, y]| = 2^n$  for some  $n \in \mathbb{N}$ . This can be maintained by having that  $P(x, y)$  holds if  $|w[x, y]| = 1$  or if there exists  $z_1, z_2 \in D$  such that  $P(x, z_1)$ ,  $P(z_2, y)$ ,  $R'_{\text{Next}}(z_1, z_2)$  and that  $\bar{R}_{\text{len}}(x, z_1, z_2, y)$ . If we assume that  $|w[x, z_1]| = 2^n$  for some  $n \in \mathbb{N}$ , which we do because we have the base case of  $w[x, y] = a$ , and that  $|w[x, z_1]| = |w[z_2, y]|$ , then it follows that if  $R'_{\text{Next}}(z_1, z_2)$  then  $w[x, y] = w[x, z_1] \cdot w[z_2, y]$  and therefore  $|w[x, y]| = 2|w[x, z_1]|$  and hence  $|w[x, y]| = 2^{n+1}$ . We then have that  $|w| = 2^n$  if  $\exists x, y: (R'_{\text{first}}(x) \wedge R'_{\text{last}}(y) \wedge P'(x, y))$ . ◀

For every choice of  $\Sigma$ , this language is not expressible in  $\text{SpLog}^\neg$  (and, hence, not in SpLog). This is easily seen by considering the case that  $\Sigma$  is unary<sup>4</sup>. As shown in [7] for core spanners and then in [18] for generalized core spanners, both classes collapse to exactly the class of regular languages if  $|\Sigma| = 1$ . As the language of all words  $a^{2^n}$  is not regular, this shows that even DynCQ can define languages that are not expressible in  $\text{SpLog}^\neg$ .

<sup>4</sup> Larger alphabets then follow by observing that the class of  $\text{SpLog}^\neg$ -languages is trivially closed under intersection with regular languages.

## 11:18 Dynamic Complexity of Document Spanners

Combining this with Theorem 3.13 and Theorem 3.15, we respectively conclude that DynCQ is strictly more expressive than core spanners and that DynFO is strictly more expressive than generalized core spanners.

As explained in Section 6 of [6], there are few inexpressibility results for SpLog that generalize to non-unary alphabets (and basically none for SpLog<sup>⊖</sup>), apart from straightforward complexity observations that are not particularly illuminating. Nonetheless, Proposition 6.7 in [6] establishes that none of the following relations is SpLog-selectable:

► **Proposition 4.3.** *The following relations are DynCQ-selectable but not SpLog-selectable:*

$$\begin{aligned} R_{\text{num}(a)} &:= \{(w_1, w_2) \mid |w_1|_a = |w_2|_a\} \text{ for } a \in \Sigma, \\ R_{\text{perm}} &:= \{(w_1, w_2) \mid |w_1|_a = |w_2|_a \text{ for all } a \in \Sigma\}, \\ R_{\text{rev}} &:= \{(w_1, w_2) \mid w_2 = w_1^R\}, \text{ where } w_1^R \text{ is the reversal of } w_1, \\ R_{<} &:= \{(w_1, w_2) \mid |w_1| < |w_2|\}, \\ R_{\text{scatt}} &:= \{(w_1, w_2) \mid w_1 \text{ is a scattered subword of } w_2\}, \end{aligned}$$

where  $w_1$  is a scattered subword of  $w_2$  if, for some  $n \geq 1$ , there exist  $s_1, \dots, s_n, \bar{s}_0, \dots, \bar{s}_n \in \Sigma^*$  such that  $w_1 = s_1 \cdots s_n$  and  $w_2 = \bar{s}_0 s_1 \bar{s}_1 \cdots s_n \bar{s}_n$ .

**Proof.** The relations  $R_{\text{scatt}}$ ,  $R_{\text{num}(a)}$ , and  $R_{\text{rev}}$  have case distinctions equivalent to the proof of Lemma 3.7, therefore we give the overarching idea of the proof but without exploring every case. See [9] for a full proof of Lemma 3.7.

**Maintaining  $R_{\text{scatt}}$ :** For insertion, we give three steps for this proof; inheritance, base case, and an inductive step.

We have that if  $w[u_1, u_2]$  is a scattered subword of  $w[v_1, v_2]$  and  $u$  is outside of the interval  $[u_1, u_2]$ , then  $w'[u_1, u_2]$  remains a scattered subword of  $w'[v_1, v_2]$  and therefore  $R'_{\text{scatt}}(u_1, u_2, v_1, v_2)$  should hold. We call this step inheritance.

The base case is that given the update  $\text{ins}_\zeta(u)$  for some  $u \in D$ , if there exists  $v \in D$  such that  $v_1 \leq v \leq v_2$  and  $w(v) = w(u) = \zeta$ , then it follows that  $w(u)$  is a scattered subword of  $w[v_1, v_2]$  and therefore  $R'_{\text{scatt}}(u, u, v_1, v_2)$  should hold.

For the inductive step, given that we have some update  $\text{ins}_\zeta(u)$ , if  $w[u_1, x_1]$  is a scattered subword of  $w[v_1, x_2]$  and  $w[x_3, u_2]$  is a scattered subword of  $w[x_4, v_2]$ , it follows that  $w[u_1, u_2]$  is a scattered subword of  $w[v_1, v_2]$  if  $x_1 \rightsquigarrow_{w'} u \rightsquigarrow_{w'} x_3$  and  $w(u)$  is a scattered subword of  $w[x_2, x_4]$ . Deletion is dealt with analogously, although without the base case.

**Maintaining  $R_{\text{num}(a)}$ :** We again give three steps; inheritance, the base case(s), and an inductive step.

We have that if  $|w[u_1, u_2]|_a = |w[v_1, v_2]|_a$  and  $u$  is outside of the interval  $[u_1, u_2]$ , then  $|w'[u_1, u_2]|_a = |w'[v_1, v_2]|_a$  and therefore  $R'_{\text{num}(a)}(u_1, u_2, v_1, v_2)$  should hold. We call this step inheritance. We have that  $(u_1, u_2, v_1, v_2)$  is not inherited if  $u \in [u_1, u_2]$  or  $u \in [v_1, v_2]$ , but this should be dealt with by the inductive step.

To maintain  $R_{\text{num}(a)}$ , we have two base cases. Given the update  $\text{ins}_a(u)$ , we have that  $|w'(u)|_a = |w'(v)|_a$  if  $w'(v) = a$ .

For the inductive step, we have that if  $|w[u_1, x_1]|_a = |w[v_1, x_2]|_a$  and  $|w(u)|_a = |w(v)|_a$  and  $|w[x_3, u_2]|_a = |w[x_4, v_2]|_a$  where  $x_1 \rightsquigarrow_{w'} u \rightsquigarrow_{w'} x_3$  and  $x_2 \rightsquigarrow_{w'} v \rightsquigarrow_{w'} x_4$ , then  $|w'[u_1, u_2]|_a = |w'[v_1, v_2]|_a$ . Dealing with deletion is analogous to insertion but without the base case.

**Maintaining  $R_{\text{rev}}$ :** We can maintain this with a simple variation of the update formula which maintains  $R_{\text{eq}}$ . Firstly, we remove the constraint that the first subword must appear before the second. Then, whenever  $R_{\text{eq}}(\cdot)$  is used as a subformula, one would need to

use  $R_{\text{rev}}(\cdot)$  instead. The more involved aspect of altering the update formulas would be to reverse the ordering of certain indices. Informally, check  $y \rightsquigarrow_w x$  instead of  $x \rightsquigarrow_w y$  where necessary.

**Maintaining  $R_{\text{perm}}$ :**  $\phi_{\partial}^{R_{\text{perm}}}(u; u_1, u_2, v_1, v_2) := \bigwedge_{\zeta \in \Sigma} (\phi_{\partial}^{R_{\text{num}(\zeta)}}(u; u_1, u_2, v_1, v_2))$ .

**Maintaining  $R_{<}$ :**

$$\phi_{\partial}^{R_{<}}(u; u_1, u_2, v_1, v_2) := \exists x_1 \exists x_2 : (R_{\text{len}}(u_1, u_2, x_1, x_2) \wedge (x_1 < v_1) \wedge (v_1 \leq v_2) \wedge (v_2 < x_2)).$$

◀

By Lemma 5.1 in [6], a  $k$ -ary relation  $R$  is **SpLog**-selectable if and only there is some **SpLog**-formula  $\varphi(W; x_1, \dots, x_k)$  such that for all  $\sigma$  that satisfy  $\sigma(x_i) \sqsubseteq \sigma(W)$  for all  $i \in [k]$ , we have  $\sigma \models \varphi$  if and only if  $(\sigma(x_1), \dots, \sigma(x_k)) \in R$ . One can show with little effort that relations like string inequality, the substring relation, or equality modulo a bounded Levenshtein-distance are all **SpLog**-selectable (see Section 5.1 of [6]). By Lemma 3.12, we can directly use these relations in constructions for **DynCQ**-definable languages and **DynCQ**-selectable relations.

► **Example 4.4.** For  $k \geq 1$  and  $w_1, w_2 \in \Sigma^*$ , we say that  $w_1$  is a  $k$ -scattered subword of  $w_2$  if there exist  $s_1, \dots, s_k, \bar{s}_0, \dots, \bar{s}_k \in \Sigma^*$  such that  $w_1 = s_1 \cdots s_k$  and  $w_2 = \bar{s}_0 s_1 \bar{s}_1 \cdots s_k \bar{s}_k$ . This relation is **SpLog**-selectable<sup>5</sup>, as demonstrated by the following **SpLog**-formula which uses syntactic sugar from Section 5.1 of [6]:

$$\varphi(W; w_1, w_2) := \exists s_1, \dots, s_k, \bar{s}_0, \dots, \bar{s}_k : ((w_1 \dot{=} s_1 \cdots s_k) \wedge (w_2 \dot{=} \bar{s}_0 s_1 \bar{s}_1 \cdots s_k \bar{s}_k)).$$

Although one could show directly that the  $k$ -scattered subword relation is **DynCQ**-selectable, using **SpLog** and Lemma 3.12 can avoid hand-waving.

We can even generalize this approach beyond **SpLog**. In the proof of Lemma 3.12, we use the fact the every regular language is in **DynCQ** to maintain regular constraints for **SpLog**. Analogously, we can extend **SpLog** with relation symbols for any **DynCQ**-selectable relation and use the resulting logic for **DynCQ**. Of course, all this applies to **SpLog**<sup>∇</sup> and **DynFO**.

## 5 Conclusions

From a document spanner point of view, the present paper establishes upper bounds for maintaining the three most commonly examined classes of document spanners, namely **DynPROP** for regular spanners, **DynCQ** for core spanners, and **DynFO** for generalized core spanners. While the bounds for regular spanners and generalized core spanners are what one might expect from related work, the **DynCQ**-bound for core spanners might be considered surprising low (keeping in mind, of course, that it is still open whether **DynCQ** is less expressive than **DynFO**).

By analyzing the proof of Lemma 3.12, the central construction of this main result, it seems that the most important part of maintaining core spanners is updating the string equality relation and the regular constraints. One big question for future work is whether this might have any practical use for the evaluation of core spanners. Although some may consider this unlikely, there is at least some possibility that some techniques might be useful.

<sup>5</sup> Unlike a relation for unbounded scattered subword.

In the present paper, we only examine updates that affect single letters. At least as far as the main result is concerned, it should be possible to generalize this to cut and paste operations, as they are commonly found in text editors. These other operations beyond single letters are promising directions for further work.

From a dynamic complexity point of view, Section 4 describes how SpLog can be used as a convenient tool that allows shorter proofs that languages can be maintained in DynCQ. One consequence of this is that a large class of regular expressions with backreference operators (see Section 5.3 of [6]) are in fact DynCQ-languages.

---

## References

- 1 Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. Constant-Delay Enumeration for Nondeterministic Document Spanners. In *Proceedings of ICDT 2019*, pages 22:1–22:19, 2019.
- 2 Johannes Doleschal, Benny Kimelfeld, Wim Martens, Yoav Nahshon, and Frank Neven. Split-Correctness in Information Extraction. In *Proceedings of PODS 2019*, pages 149–163, 2019.
- 3 Guozhu Dong, Jianwen Su, and Rodney Topor. Nonrecursive incremental evaluation of datalog queries. *Annals of Mathematics and Artificial Intelligence*, 14(2-4):187–223, 1995.
- 4 Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. Document Spanners: A Formal Approach to Information Extraction. *Journal of the ACM*, 62(2):12, 2015.
- 5 Fernando Florenzano, Cristian Riveros, Martín Ugarte, Stijn Vansummeren, and Domagoj Vrgoc. Constant Delay Algorithms for Regular Document Spanners. In *Proceedings of PODS 2018*, pages 165–177, 2018.
- 6 Dominik D. Freydenberger. A Logic for Document Spanners. *Theory of Computing Systems*, 63(7):1679–1754, 2019.
- 7 Dominik D. Freydenberger and Mario Holldack. Document Spanners: From Expressive Power to Decision Problems. *Theory of Computing Systems*, 62(4):854–898, 2018.
- 8 Dominik D. Freydenberger, Benny Kimelfeld, and Liat Peterfreund. Joining Extractions of Regular Expressions. In *Proceedings of PODS 2018*, pages 137–149, 2018.
- 9 Dominik D. Freydenberger and Sam M. Thompson. Dynamic Complexity of Document Spanners, 2019. [arXiv:1909.10869](https://arxiv.org/abs/1909.10869).
- 10 Wouter Gelade, Marcel Marquardt, and Thomas Schwentick. The dynamic complexity of formal languages. *ACM Transactions on Computational Logic*, 13(3):19:1–19:36, 2012.
- 11 Tao Jiang, Efim Kinber, Arto Salomaa, Kai Salomaa, and Sheng Yu. Pattern languages with and without erasing. *International Journal of Computer Mathematics*, 50(3-4):147–163, 1994.
- 12 Katja Losemann. *Foundations of Regular Languages for Processing RDF and XML*. PhD thesis, University of Bayreuth, 2015. URL: <https://epub.uni-bayreuth.de/2536/>.
- 13 Francisco Maturana, Cristian Riveros, and Domagoj Vrgoc. Document Spanners for Extracting Incomplete Information: Expressiveness and Complexity. In *Proceedings of PODS 2018*, pages 125–136, 2018.
- 14 Andrea Morciano, Martín Ugarte, and Stijn Vansummeren. Automata-Based Evaluation of AQL queries. Technical report, Université Libre de Bruxelles, 2016.
- 15 Pablo Muñoz, Nils Vortmeier, and Thomas Zeume. Dynamic Graph Queries. In *Proceedings of ICDT 2016*, pages 14:1–14:18, 2016.
- 16 Sushant Patnaik and Neil Immerman. Dyn-FO: A parallel, dynamic complexity class. *Journal of Computer and System Sciences*, 55(2):199–209, 1997.
- 17 Liat Peterfreund, Dominik D. Freydenberger, Benny Kimelfeld, and Markus Kröll. Complexity Bounds for Relational Algebra over Document Spanners. In *Proceedings of PODS 2019*, pages 320–334, 2019.
- 18 Liat Peterfreund, Balder ten Cate, Ronald Fagin, and Benny Kimelfeld. Recursive Programs for Document Spanners. In *Proceedings of ICDT 2019*, pages 13:1–13:18, 2019.

- 19 Markus L. Schmid. Characterising REGEX Languages by Regular Languages Equipped with Factor-Referencing. *Information and Computation*, 249:1–17, 2016.
- 20 Thomas Zeume. *Small dynamic complexity classes*. Springer, 2017.
- 21 Thomas Zeume and Thomas Schwentick. Dynamic conjunctive queries. *Journal of Computer and System Sciences*, 88:3–26, 2017.