Incremental Schema Validation for XML Databases

Damien K. Fisher Raymond K. Wong
School of Computer Science & Engineering
University of New South Wales
Sydney, NSW 2052, Australia
{damienf,wong}@cse.unsw.edu.au

Technical Report
UNSW-CSE-TR-0423
June 2004

SCHOOL OF COMPUTER SCIENCE & ENGINEERING
THE UNIVERSITY OF NEW SOUTH WALES

Abstract

An important feature of any database system is the ability to perform consistency checks on the data being manipulated in the system. For XML databases, the most fundamental such check is validation with respect to a fixed data schema, such as a DTD or XML Schema. While it is straightforward to perform such validation on an entire XML document, it would be extremely inefficient to revalidate the entire database from scratch upon every modification. Hence, it is natural to ask whether efficient incremental schema validation techniques exist.

In this paper, we investigate the complexity bounds of such techniques for a variety of schema languages. As with previous work on this subject, we mainly study the related problem of dynamic membership in a regular language. We define a class of regular expressions for which dynamic membership can be determined in constant time, and a larger class for which dynamic membership can be determined in $O(\log \log n)$ time. We also show that, in general, validation can be performed in time $O(\log n / \log \log n)$, and that this bound is tight for some schemas.

1 Introduction

The management of XML data has been an active research topic over the past few years, spurned on by the success of XML in the industry. Significant effort has been expended on many of the fundamental aspects of XML data management, such as storage, efficient querying, and selectivity estimation, with generally positive results. Another important area — and the work most relevant to this paper — is that of incremental integrity and constraint checking.

Incremental problems are well studied in the relational literature, as *incremental evaluation systems* [15, 27]. However, very little of this theory carries over to the unique case of XML. In the context of XML databases, integrity checking has been investigated in a few ways. Forms of integrity constraints available in relational databases, such as keys, have also been generalized to the case of XML [9].

The work most relevant to this paper focuses on validation against a schema. Alon et al [1] demonstrated the difficulty of the typechecking problem for XML views of relational databases: in general, determining whether a transformation from the underlying database to an XML view satisfies a given DTD is undecidable. The complexity of validating a fixed XML document is understood [31], but in many contexts we would like to extend this to more flexible models. Such work has been carried out in the streaming model [32] and in the model of an XML document undergoing insertions and deletions [2, 6, 25].

Incremental schema validation is a vital component in a wide range of XML applications. Here a few examples:

1. Many interactive XML editors and collaborative tools provide on-the-fly validation. As the size of XML documents increases, such tools would benefit greatly from efficient algorithms to perform this check as the document is edited.

2. Incremental schema validation is obviously a fundamental problem for the many commercial and research XML databases available. While some applications make use of completely unstructured XML, the vast majority require conformance to a particular schema.

3. As described above, it has been shown that in the general case, typechecking XML views (whether over XML or relational databases) is undecidable [1]. While we cannot hope to perform this check statically, it is still possible to perform it at runtime: as the view is updated, we can check for its conformance to the given schema. An efficient validation algorithm would yet again play a vital role.

In addition to this, our results apply to the incremental validation of regular languages, which are fundamental to many areas of computer science. It thus would not be surprising to find additional, and unexpected, applications of incremental validation in areas far removed from XML databases.

In this paper, we further investigate the existence of theoretically and practically sound algorithms for this problem, extending previous work [2, 6, 25] significantly. The main results of this paper are:

- We find a class of schemas for which validation can be performed in constant time and space. This class strictly contains one found by Barbosa et al [2].

- We find another class of schemas for which validation can be performed in $O(1)$ time and $O(n)$ space, where n is the number of nodes in the XML document.

- More generally, we give a validation algorithm using $O(\log\log n)$ time and $O(n)$ space for updating *star-free* schemas. For a small class of schemas (not contained in the two classes mentioned above), the time complexity drops to $O(1)$. Star-free schemas cover most practical schemas, and hence this is a very significant result for the use of XML. The algorithm is especially practical as there are some useful synergies between its data structures and the popular region algebra approaches to storing XML.

- For non-star-free schemas we give an $O(\log n/\log\log n)$ algorithm and show that this bound is tight for some schemas, hence resolving a question of Papakonstantinou and Vianu [25].

- For non-recursive schemas, we provide a practical improvement which, although it does not affect the asymptotic results, is considerably easier to maintain in practice, and reduces the space usage substantially.

- Our algorithms apply to most popular schema languages, including DTDs, XML Schema, and non-recursive RELAX NG schemas.

The rest of this paper is organized as follows. In Section 2, we provide the basic definitions used in the rest of the paper. We discuss the strengths and limitations of related work in Section 3. Sections 4 through 7 discuss our techniques for several classes of regular expressions. In Section 8, we discuss a practical optimization that simplifies the implementation for non-recursive schemas. Finally, Section 9 concludes the paper.

2 Background and Basic Framework

2.1 Regular Languages

Let Σ be a finite, non-empty alphabet, Σ^* be the set of all words over Σ, and $\epsilon \in \Sigma^*$ be the word of length zero. Then a regular expression e over Σ, defining a language $L(e) \subseteq \Sigma^*$, is defined recursively as:

- $e = \emptyset$ denotes $L(e) = \emptyset$;

- $e = \epsilon$ denotes $L(e) = \{\epsilon\}$;

- $e = a$ for any $a \in \Sigma$ denotes $L(e) = \{a\}$;

- **Concatenation**: For any regular expressions e_1, e_2, $e = e_1 \cdot e_2$ denotes $L(e) = \{w_1 w_2 \mid w_1 \in L(e_1), w_2 \in L(e_2)\}$;

- **Union**: For any regular expressions e_1, e_2, $e = e_1|e_2$ denotes $L(e) = L(e_1) \cup L(e_2)$; and

- **Kleene closure**: For any regular expression e', $e = e'^*$ denotes $L(e) = \{w_1 \dots w_n \mid \forall n \geq 0, \forall w_i \in L(e')\}$.

A *regular language* is a language defined by a regular expression. The above definition of regular expressions is minimal, but two additional operators are often defined:

- **Intersection**: For any regular expressions e_1, e_2, $e = e_1 \cap e_2$ denotes $L(e) = L(e_1) \cap L(e_2)$; and

- **Complement**: For any regular expression e', $e = \neg e'$ denotes $L(e) = \Sigma^* - L(e')$.

We call the syntax which includes these additional operators the *extended regular expression syntax*. Any extended regular expression can be expressed by an equivalent standard regular expression, but the additional power of extended regular expressions allows us to easily define an important class of regular languages:

Definition 1 (Star-Free Regular Languages) *A star-free regular language is a language which can be defined by an extended regular expression without using Kleene closure.*

Star-free regular languages are also known as *counter-free languages*, because they do not have the ability to count modulo some number[1]. For instance, the regular language $(000)^*$, which consists of all strings of zeros with a length of zero modulo three, is not star-free. However, the language $(01)^*$ is star-free, as an equivalent extended regular expression for this language is $\epsilon \cup (\neg(\neg\emptyset \cdot (11 \cup 00) \cdot \neg\emptyset) \cap (0 \cdot \neg\emptyset \cdot 1))$. Note that, as the last example shows, the term *star-free* does not imply that *all* expressions describing the language be star-free, only that one such extended regular expression exists.

For any $k \in \mathbb{N}$ and $w = a_1 a_2 \ldots a_n \in \Sigma^*$ of length $n \geq k$, define:

- $l_k(w) = a_1 \ldots a_k$
- $r_k(w) = a_{n-k+1} \ldots a_n$
- $I_k(w) = \{a_i \ldots a_{i+k-1} \mid i \in \{2, \ldots, n-k\}\}$

Several important subclasses of star-free regular languages, first introduced by McNaughton and Papert [24], can now be defined:

Definition 2 (Strictly k-Testable Language) *A regular language L is strictly k-testable if there are sets $L_k, R_k, I_k \subseteq \Sigma^k$, such that for all words $w \in \Sigma^*$ of length k or more, $w \in L$ if and only if $l_k(w) \in L_k$, $r_k(w) \in R_k$, and $I_k(w) \subseteq I_k$.*

Definition 3 (Strictly Locally Testable Language) *A regular language L is strictly locally testable if it is strictly k-testable for some k. The order of strict local testability is the smallest k for which L is strictly k-testable.*

Definition 4 (k-Testable Language) *A regular language L is k-testable if, for all $w_1, w_2 \in \Sigma^*$, $l_k(w_1) = l_k(w_2)$, $r_k(w_1) = r_k(w_2)$, and $I_k(w_1) = I_k(w_2)$, implies that $w_1 \in L$ if and only if $w_2 \in L$.*

Definition 5 (Locally Testable Language) *A regular language L is locally testable if it is k-testable for some k. The order of local testability is the smallest k for which L is k-testable.*

Note that no restrictions are placed on words of length less than k: in other words, whether a language L is k-testable or not is completely independent of the words in L of length less than k. The above classes of languages are most intuitively understood by considering an acceptor for them. Figure 1 demonstrates the idea in the case of a 3-testable language. A k-testable language is accepted by a machine which only has a fixed amount of memory, and which has a head which can read k consecutive symbols (in the figure, the head of the machine is currently placed over the first occurrence of the substring 010). This machine moves over the input tape in an arbitrary order, and determines the set of substrings of length k in the input string, as well as the strings of length k with which the input begins and ends. It then feeds this data into a decision procedure, which determines whether the string belongs to the language. The acceptor in Figure 1 could accept any 3-testable language, for example:

[1] They do, however, have the ability to count up to some fixed limit.

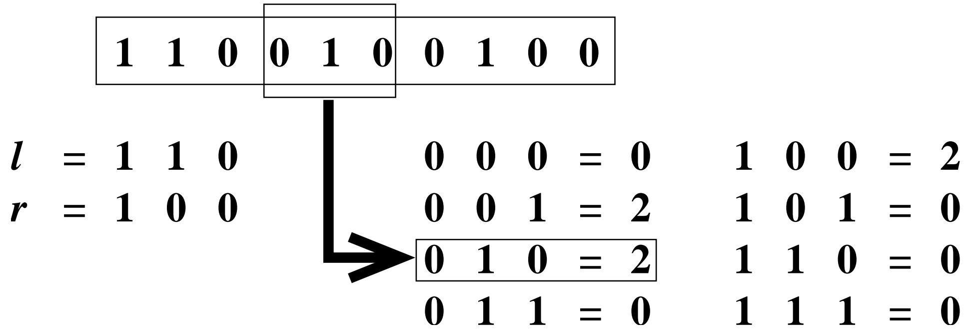

Figure 1: An acceptor for a 3-testable language over $\Sigma = \{0, 1\}$.

- The language consisting of all strings containing either both 000 and 111, or neither of them, is 3-testable. The input string in Figure 1 is a member of this language, because the counters for the substrings 000 and 111 are both zero, and $l, r \notin \{000, 111\}$.
- The language consisting of all strings containing 000 is strictly 3-testable. The input string in Figure 1 is not a member of this language.

The set of strictly locally testable languages is closed under intersection, but not under union or negation. The locally testable languages are closed under all Boolean operations, but not under concatenation. The relationship between these classes is captured by the following results from [24]:

Theorem 6 *The closure of the strictly k-testable languages under Boolean operations is the k-testable langauges.*

Theorem 7 *The closure of the locally testable languages under Boolean operations and concatenation is the star-free regular languages.*

2.2 Deterministic Finite Automata

A deterministic finite automaton (DFA) A is a quintuple $\langle Q, \Sigma, \delta, q_I, F \rangle$, where:

- Q is a finite set of states;
- Σ is a finite alphabet;
- $\delta : Q \times \Sigma \to Q$ is the transition function;
- $q_I \in Q$ is the initial state; and
- $F \subseteq Q$ is the set of final states.

We define the extended transition function $\hat{\delta} : Q \times \Sigma^* \to Q$ in the natural way:

$$\begin{aligned} \hat{\delta}(q, \epsilon) &= q \\ \hat{\delta}(q, aw) &= \hat{\delta}(\delta(q, a), w) \quad \forall a \in \Sigma, \forall w \in \Sigma^* \end{aligned}$$

A word w is accepted by a DFA if and only if $\hat{\delta}(q_I, w) \in F$. Kleene's theorem states that a language is accepted by a DFA if and only if it is regular. The process of converting between a regular expression and a DFA is well-known, see, e.g., the survey of Yu [39].

2.3 Syntactic Monoid

A *monoid* is a set S equipped with a binary operator (represented here by multiplication) and an identity element 1, so that $\forall x \in S$, $1s = s1 = s$. As an example of a monoid, the *free monoid* over Σ is simply Σ^* equipped with concatenation and identity element ϵ.

Amongst all DFAs accepting a particular regular language L, there exists one (unique up to isomorphism) using a minimal number of states, $A_L = \langle Q_L, \Sigma, \delta_L, q_I, F_L \rangle$. It is straightforward to obtain this DFA (again, see, e.g., Yu [39]). The *syntactic monoid* M_L of L is given by the action of $\hat{\delta}_L$ on Q_L. More precisely, we define a function $\phi : \Sigma^* \to Q_L \to Q_L$ as $\phi(w)(q) = \hat{\delta}_L(x, q)$, $\forall w \in \Sigma^*$ and $\forall q \in Q_L$. The syntactic monoid is then the image of the free monoid under ϕ. Intuitively, the syntactic monoid contains all possible ways of moving between states in the minimal automaton.

Syntactic monoids yield a surprising and beautiful characterization of star-free regular languages:

Theorem 8 (Schützenberger [30]) *A regular language is star-free if and only if its syntactic monoid contains no non-trivial subgroups.*

2.4 Alternating Finite Automata

An alternating finite automaton (AFA) is a useful generalization of the typical finite automaton which allows boolean functions to be used in transitions. First, we define some needed notation: if S_1, S_2 are finite sets, then $S_1^{S_2}$ is the set of vectors over S_1 of length $|S_2|$, with coefficients of the vectors indexed by elements of S_2; for a vector $v \in S_1^{S_2}$ and $x \in S_2$, we denote by v_x the coefficient of v indexed by x.

An alternating finite automaton A is a quintuple $\langle Q, \Sigma, \delta, q_I, F \rangle$, where Q, Σ, q_I, and F are defined as in the case of a DFA. For each $q \in Q$, we define a function $\delta_q : \Sigma \times \{0,1\}^Q \to \{0,1\}$, which gives the transition function for each state q. The transition function $\delta : Q \times \Sigma \times \{0,1\}^Q \to \{0,1\}$ is then defined by $\delta(q, a, v) = \delta_q(a, v)$. As in the case of a DFA, we can extend δ to $\hat{\delta} : Q \times \Sigma^* \times \{0,1\}^Q \to \{0,1\}$:

$$\begin{aligned} \hat{\delta}(q, \epsilon, x) &= x_q \\ \hat{\delta}(q, aw, x) &= \delta(q, a, \hat{\delta}(q, w, x)) \end{aligned}$$

Define a vector $f \in \{0,1\}^{|Q|}$ as $f_q = 1$ if and only if $q \in F$. Then a word w is accepted by A if and only if $\hat{\delta}(q_I, w, f) = 1$.

An alternative representation, which is easier to visualize, is the *equational representation.* We can represent an AFA by a set of equations of the following form:

$$X_q = \sum_{a \in \Sigma} a \cdot \delta_q(a, X) + f_q, \forall q \in Q$$

In the above, X is a vector of length $|Q|$ of variables (over words in Σ^*) indexed by the states $q \in Q$. The language consisting of words satisfying the above set of equations is equal to the language of the corresponding AFA (see Yu [39]). The expressive power of alternating finite automata is exactly equivalent to deterministic automata, but they can be considerably more concise.

As an example, Figure 2 gives the equational representation of an AFA accepting the set of all words over $\{a, b, c\}$ in which some a precedes a b, and no b precedes a c.

$$\begin{aligned}
X_1 &= a \cdot (X_3 \wedge \neg X_4) + b \cdot (X_2 \wedge \neg X_5) + c \cdot X_1 + 1 \\
X_2 &= a \cdot X_3 + b \cdot X_2 + c \cdot X_2 + 0 \\
X_3 &= a \cdot X_3 + b \cdot X_6 + c \cdot X_3 + 0 \\
X_4 &= a \cdot X_4 + b \cdot X_5 + c \cdot X_4 + 0 \\
X_5 &= a \cdot X_5 + b \cdot X_5 + c \cdot X_6 + 0 \\
X_6 &= a \cdot X_6 + b \cdot X_6 + c \cdot X_6 + 1
\end{aligned}$$

Figure 2: The equational representation for $\Sigma^* a \Sigma^* b \Sigma^* \cap \neg(\Sigma^* b \Sigma^* c \Sigma^*)$, $\Sigma = \{a, b, c\}$.

2.5 1-Unambiguous Regular Languages

1-Unambiguous regular languages are used in several XML schema languages [7, 33]. For each regular expression e, let $\Sigma(e)$ denote the symbols used in e. In most regular expressions, one symbol is used more than once (e.g., in $(00)^*$, 0 is used twice). We thus associate with each regular expression e, a *marked* regular expression $m(e)$ which disambiguates the symbols occurring in e. For example, we associate with $e = (00)^*$ the expression $e' = (0_1 0_2)^*$, so that $\Sigma(e') = \{0_1, 0_2\}$. The following definition is taken from Brüggemann-Klein and Wood [8]:

Definition 9 (1-Unambiguous Regular Expression) *A regular expression is 1-unambiguous if we can unique determine which position of a symbol in a regular expression should match a symbol in an input word without looking beyond that symbol in the input word.*

We will call a language 1-unambiguous if we can find a 1-unambiguous regular expression for that language.

2.6 XML

For the purposes of this paper, it suffices to model XML documents as unranked, finite, labelled, ordered trees. We neglect features such as attributes and data values, which can be trivially checked incrementally. We also ignore cross-referencing features such as the ID/IDREF feature of DTDs, as this issue has been investigated previously by Barbosa et al [2], and is also relatively easy to handle efficiently incrementally.

Therefore, we model an XML document as a tuple $\langle \Sigma, t, \lambda \rangle$, where Σ is an alphabet (possibly countably infinite), t is an ordered tree, and λ is a mapping from the nodes of t to Σ, giving the label of each node in the tree. Even though Σ can be infinite, in the sequel we will only concern ourselves with the case when the alphabet is finite. This poses no problem, since we are only interested in element labels that occur in the document schema under consideration; since the schema is finite, this subset of Σ must also be finite.

In the sequel, we will assume a RAM machine model with word size $O(\log|D| + \log|S|)$, where D is the XML document and S is the schema under consideration. We will assume a representation of the tree where, given a specific node n, we can access in $O(1)$ steps: the label $\lambda(n)$, the parent of n, the immediate siblings of n, and the first child of n. This is the same model assumed by previous work [2, 25], and is quite realistic. In fact, for most of our work, we will only require that we can access the nearest k nodes in document order; this has the advantage of mapping more naturally onto a representations where the document is stored in document order over a series of disk blocks, e.g., the B-tree representation used in several XML systems [19, 21].

$$\begin{aligned} a &= b^* \\ b &= c \\ b &= d \end{aligned}$$

(a) DTD

$$\begin{aligned} a &= b^* c^* \\ b &= d_1 \\ c &= d_2 \end{aligned}$$

(b) XML Schema

$$\begin{aligned} a &= b^* \\ b_1 &= c_1 \\ b_2 &= c_2 \end{aligned}$$

(c) RELAX NG

Figure 3: Examples of various schema languages

2.7 XML Schema Languages

There are numerous XML schema languages, which varying degrees of power. We briefly outline here the most popular along with corresponding theoretical models. We will confine our discussion to the structural facilities of these languages, as the incremental validation of non-structural properties of an XML document is trivial and uninteresting.

2.7.1 Document Type Definition

The Document Type Definition (DTD) schema language is the schema language given in the original XML specification [7]. DTDs are best modelled by *extended context free grammars* (ECFG), in which the productions are given as regular expressions over an alphabet Σ (the same alphabet is used for both terminals and non-terminals). Given an ECFG S, an XML document t is accepted by $L(S)$ if t is a valid parse tree of S.

We note one important practical restriction: the XML specification requires that the regular expressions used in a DTD are 1-unambiguous. This restriction was introduced into the specification in order to simplify validation; however, as we will show, a much larger class of regular expressions can be validated efficiently. In this paper, all ECFGs are unrestricted in this regard.

Figure 3(a) gives the ECFG for a very simple DTD. In this example, an a element's children must be one or more bs, and each of these b elements can have either c or d as a child.

Throughout this paper, we will generalize definitions of regular languages to ECFGs in the natural way. For instance, we will say that an ECFG S is locally testable if all productions in S are locally testable.

2.7.2 XML Schema

One major limitation of DTDs is that the type of an element cannot be separated from its name. This can be problematic when we wish the type of an element to depend on its context;

in a DTD, all types for a particular element are always considered. The XML Schema language [33] lifts this restriction.

XML Schemas have two relevant restrictions. As with DTDs, all regular expressions must be 1-unambiguous; as noted above, we ignore this restriction in this work. Second, the type of an element can only depend on its context. More specifically, XML Schema requires that the type of the element can be determined given the type of the parent. For instance, if one has two types a_1, a_2 for an element with label a, then if this element occurs as a child of an element of type b, the type definition must refer deterministically to either a_1 or a_2. With this restriction, XML Schemas can be easily modeled as ECFGs, and hence give no real additional expressive power over DTDs. As most of our results apply only to ECFGs, we will enforce this second restriction throughout our work.

Figure 3(b) gives the ECFG for an XML Schema. In this example, two different element types refer to different types for elements with the label d; however, these references are deterministic. For example, when the XML parser sees a b element, it immediately knows that any d elements it sees must satisfy type d_1, and not type d_2.

2.7.3 RELAX NG

The most powerful schema language in prevalent use is RELAX NG [12]. This is essentially as expressive as XML Schema, except that it imposes neither of the two conditions described above. The most important restriction, in terms of incremental schema validation, is allowing elements to have types which are context-independent; this means that such schemas can no longer be represented as ECFGs (instead they are usually modelled as specialized DTDs [26]). The best known algorithm for these schemas takes linear space and $O(\log^2 n)$ time [25]. The work described in this paper does not improve the general situation for these schemas; however, we do provide improvements for non-recursive RELAX NG schemas in Section 8.

Figure 3(c) demonstrates the additional power of RELAX NG over XML Schema. In this case, when an XML parser sees a b element, it does not know immediately whether it is of type b_1 or b_2; this is dependent on the type of its child.

2.8 Problem Definition

The problem we investigate in this paper is precisely defined as follows. Let $\mathcal{D}$ be the set of all XML documents, S be some schema, and $L(S) \subseteq \mathcal{D}$ the set of documents satisfying the schema S. We define the set of valid updates to be functions $u : \mathcal{D} \to \mathcal{D}$, corresponding to the insertion and deletion of subtrees anywhere in a document, subject to the constraint that the new document is still a tree.

There are actually two variants of the problem. The first was the one considered by Barbosa et al [2]:

Definition 10 (Strict Incremental Schema Validation) *Given some schema S, document $D \in L(S)$, and update u, determine whether $u(D) \in L(S)$.*

The second is more general and more difficult. It was previously considered by Papakonstantinou and Vianu [25]:

Definition 11 (Lazy Incremental Schema Validation) *Given some schema S, document $D \in L(S)$, and series of updates $u_1, u_2, \ldots, u_n$, determine whether $(u_n \circ u_{n-1} \circ \cdots \circ u_1)(D) \in L(S)$.*

The reason lazy schema validation is more difficult is because a document can become invalid, have several updates applied to it, and then become valid again due to one "critical" update. We believe that lazy validation is much more useful than strict validation, as we now demonstrate with the three applications we listed in the introduction:

1. In an editor or collaborative tool, the user will generally want immediate feedback as to whether his changes are valid or not. However, the user will also want to allow the document to remain in an inconsistent state during editing. Therefore, lazy validation is to be preferred.
2. In a database, it is often the case that integrity constraint checking is suspended whilst inside a transaction, due to the fact that it is often impossible to update the database otherwise. Therefore, lazy validation is necessary, so that integrity checking can be performed at commit time, regardless of the number of changes made.
3. In a materialized XML view, one change to the underlying database might change several parts of the view, which need to be considered together during validation. Therefore, lazy validation is again required.

Another serious problem with strict validation is that sometimes one can get into a situation where no updates are possible. For instance, consider the regular language $(00)^*$. If we have some valid string from this language, then in order to obtain another valid string we must either add or delete at least two characters. However, strict validation requires that after every insertion or deletion, the string remain valid; this means that we can never actually insert the second character.

In this paper, we will focus our attention on lazy validation; note that any solution to the lazy validation problem automatically solves the strict validation problem. All of our results will be stated in terms of regular languages, and hence the problem we will be mainly interested in is the dynamic membership problem for regular languages:

Definition 12 (Dynamic Membership Problem for Regular Languages — DynReg) *Let L be a regular language. The dynamic membership problem for L is to maintain a data structure supporting the following operations on a word $w \in \Sigma^*$:*

- Insert(x, y)*: insert the character $y \in \Sigma$ after position x in the word w*
- Delete(x)*: delete the character at position x in the word w*
- Is-Member*: return whether the current word w belongs to L*

Frandsen et al [17] briefly considered the issue of dynamic membership in regular languages. However, there is an important distinction between their formulation and ours: we allow the length of the string to vary, whereas they fix the length and only allow updates. In database applications, allowing insertions and deletions is an important feature. Surprisingly, for counter-free languages we obtain similar bounds to those of Frandsen et al; however, we will describe in Section 7.2 a regular language for which the two models provide different answers.

A solution to DynReg can be used to solve the incremental schema validation problem for ECFGs, as described in [2, 25]. Briefly, for each node in the document, we store a data structurue for DynReg which we apply to that node's children, using the appropriate regular expression from the schema. At any point, the document is accepted by the schema if and only if the structure at each node is accepting (we can determine this in constant time by maintaining a counter giving the number of invalid nodes). This clearly uses linear space if the solution to DynReg uses linear space, and yields an identical worst case time complexity.

3 Related Work

Incremental schema validation is a topic which has been of interest recently. Papakonstantinou and Vianu [25] were the first to study the problem, and presented linear space schemes for both ECFGs and specialized DTDs, with $O(\log n)$ and $O(\log^2 n)$ update cost respectively. While these are quite efficient, the worst case cost is independent of structural properties of the underlying schema, and hence there are cases where one can do better. One of the contributions of this paper is to show that the $O(\log n)$ algorithm presented in [25] is optimal for some regular languages, up to a factor of $O(\log \log n)$.

Barbosa et al [2] considered incremental updating for DTDs and XML Schemas, based upon Glushkov automata (see [39]). For a small class of schemas ($1,2$-*conflict-free* schemas), they gave an algorithm solving the *strict* validation problem with constant time updates and linear space. However, this result does not carry over to the more important lazy validation case. For instance, in Section 7.2, we show the language $(00)^*10^*$ only admits solutions taking at least $\Omega(\log n/\log\log n)$ update time. The same proof holds for the language $(0101)^*2(01)^*$, which is $1,2$-conflict-free. Note also that, under strict validation, it is impossible to change a valid word of this language to another valid word. Hence, in terms of lazy validation, the class of $1,2$-conflict-free regular languages is not very interesting.

For an even smaller class of schemas (*conflict-free* schemas), Barbosa et al gave an algorithm with constant time updates and constant space. A *conflict-free* regular expression is 1-unambiguous, and has the additional restriction that no letter of the alphabet occurs more than once in the regular expression. We will say that a conflict-free regular language is a language for which we can find a conflict-free regular expression. Barbosa et al proved that incremental validation of conflict-free regular languages (given a conflict-free regular expression for that language) can be done in constant time and constant space. In fact, conflict-free regular languages are a reformulation of the idea of *local languages* [16]. In the terminology of this paper, we have:

Theorem 13 *A regular language is conflict-free if and only if it is strictly 2-testable.*

Thus, conflict-free regular languages are simply a special case of our algorithm for strict locally testable languages. In this paper, we provide constant time and space algorithms for both the strict and lazy cases, which neatly generalizes the work of Barbosa et al. We also find a constant time algorithm for a class of schemas significantly larger than those considered in previous work.

A final limitation of the work of Barbosa et al is that for more general regular languages, their methods degrade in the worst case to update cost linear in the size of the document. This limitation is also found in the work of Bouchou and Alves [6].

Schema validation in other models has also been considered in the literature. Segoufin and Vianu [32] studied the problem of validating streaming XML documents, and showed that non-recursive DTDs can be validated in such a model using finite state automata. The case of static schema validation has also been studied by Segoufin [31]: for a DOM-like representation of the tree (similar to the model presented in this paper), the problem is shown to be in LogSpace; for a SAX-like representation, the problem is in uniform NC^1.

More general dynamic problems have been studied in the literature. Patnaik and Immerman [27] defined a complexity class Dyn-FO, consisting of those problems that can be dynamically maintained in first-order logic: most importantly for our purposes, they demonstrated that the regular languages are in Dyn-FO.

As demonstrated in Section 2.3, there is a close link between regular languages and finite monoids. Frandsen et al [17] studied dynamic word problems over finite monoids, and related

this problem to dynamic membership testing over regular languages. We will use some of their results later in this paper; however, their lower bound results are not directly applicable to our problem because they only consider words of a fixed length, whereas in our case the length of the word can be varied through insertions and deletions. Also, their results depend heavily on the syntactic monoid of a regular language. While this is a theoretically convenient edifice, it suffers from two problems: it can be exponentially larger than the minimal automaton, and it is unlikely to be of use in other parts of an XML system. Algorithms based on automata are more likely to leverage existing code, and also yield intuitive insights into their running time. The majority of the results presented here are expressed in terms of automata.

4 Dynamic Membership for Strictly Locally Testable Regular Languages

4.1 Validation for Strictly Locally Testable Schemas

Let L be a strictly k-testable regular language, and assume we have the sets L_k, R_k, and I_k for L. We wish to solve DynReg over L for some word w. For a position x in w, let $I_{k,x}(w)$ be the multiset of internal substrings of w of length k which contain position x. We also define $L_{k,x}(w) = \{l_k(w)\}$ if x is one of the first k positions, and $L_{k,x}(w) = \emptyset$ otherwise ($R_{k,x}(w)$ is defined analogously). For instance, in Figure 1, if x is the third character, then $L_{3,x}(w) = \{110\}$, $I_{3,x} = \{100, 001\}$, and $R_{3,x}(w) = \emptyset$. Note that $S_{k,x}(w)$ is a multiset, so that multiple copies of the same string can appear.

To solve DynReg, we maintain a counter c, initialized to zero. While the length of w is less than k, we use c as a Boolean value indicating whether or not $w \in L$ (since k is a constant, validating w upon every update is fast). When the length of w reaches k, we implement the operations as follows[2]:

- Insert(x, y): Before inserting y after x, we first perform the following actions:
  - For each $l \in L_{k,x}(w) \cap L_{k,z}(w)$, if $l \notin L_k$, then decrement c.
  - For each $i \in I_{k,x}(w) \cap I_{k,z}(w)$, if $i \notin I_k$, then decrement c.
  - For each $l \in R_{k,x}(w) \cap R_{k,z}(w)$, if $l \notin R_k$, then decrement c.

  We then insert y after x, and do the following:

  - For each $l \in L_{k,y}(w)$, if $l \notin L_k$, then increment c.
  - For each $i \in I_{k,y}(w)$, if $i \notin I_k$, then increment c.
  - For each $l \in R_{k,y}(w)$, if $l \notin R_k$, then increment c.

- Delete(x): Let y and z be the positions immediately preceding and following x. Before deleting x, we do the following:
  - For each $l \in L_{k,x}(w)$, if $l \notin L_k$, then decrement c.
  - For each $i \in I_{k,x}(w)$, if $i \notin I_k$, then decrement c.
  - For each $l \in R_{k,x}(w)$, if $l \notin R_k$, then decrement c.

[2] The steps given are not optimal for some corner cases. The optimizations are omitted for ease of presentation, as they are straightforward.

After deleting x, we do the following:

- For each $l \in L_{k,y}(w) \cap L_{k,z}(w)$, if $l \notin L_k$, then increment c.
- For each $i \in I_{k,y}(w) \cap I_{k,z}(w)$, if $i \notin I_k$, then increment c.
- For each $l \in R_{k,y}(w) \cap R_{k,z}(w)$, if $l \notin R_k$, then increment c.

- IS-MEMBER: Return true if and only if $c = 0$.

Intuitively, c counts the number of "bad" substrings, and hence $w \in L$ if and only if $c = 0$. It is easy to see that the above procedure is correct, and takes only constant time and space. Hence we have the following result:

Theorem 14 DYNREG *can be solved in constant time and space for strictly locally testable regular languages.*

For the more general problem of schema validation for an ECFG, directly applying this algorithm would yield a constant time and linear space algorithm, since we would have to maintain one counter for each node in the document. However, since the document is valid only if all the counters are equal to zero, we can simply combine the counters into a single counter, and check that counter for zero. Hence, we have the following theorem:

Theorem 15 *Lazy validation can be performed in constant time and space for strictly locally testable ECFGs.*

4.2 Determining Strict Local Testability

For a regular language L with minimal automaton $A = \langle Q, \Sigma, \delta, q_I, F \rangle$, Caron [10] gives an algorithm for testing whether L is strictly locally testable in time $O(|\Sigma||Q|^2)$. Once we have ascertained that a language L is indeed strictly locally testable, we need to determine the order k of strict local testability. While this question has not specifically addressed in the literature to date, some of the results in [10] can be adapted to this task:

Definition 16 (Local Automaton) *A deterministic automaton $\langle Q, \Sigma, \delta, q_I, F \rangle$ is k-local for some positive integer k if for every $w \in \Sigma^k$, $|\{\hat{\delta}(q, w) \mid q \in Q\}| = 1$.*

Theorem 17 *A language is strictly $(k+1)$-testable if and only if it is recognized by a trimmed minimal k-local automaton.*

Proof: The proof follows a construction similar to that given by Caron [10]. □

It is easy to test whether an automaton is k-local, and hence this gives us an effective decision procedure for determining the order of strict testability. First, test for strict 1-testability, by checking that for all $a \in \Sigma$, a either takes all states in the automaton to a state which can reach a final state, or takes all states to a state which cannot. If it is not strictly 1-testable, then test the automaton for k-locality for increasing $k \in \{1, 2, \ldots\}$. Since we know that L is strictly locally testable, this procedure must halt. In practice, it is rare for k to be much larger than 3, and this procedure runs very quickly.

Once we have the order of strict local testability k, the final problem is to determine the sets L_k, R_k, and I_k. This can be easily done by finding all strings of length k that take some state in the automaton to a state that can reach a final state, and classifying them appropriately.

5 Dynamic Membership for Locally Testable Regular Languages

5.1 Validation for Locally Testable Languages

Let L be a k-testable regular language, over which we wish to solve DYNREG for some word w. Validation proceeds in a manner similar to that for strictly testable languages. However, instead of maintaining one counter, we now maintain $|\Sigma|^k + 2$ words (one counter for each possible internal substring of length k, and one word identifying the prefix and suffix of length k of w). These variables are maintained in an analogous fashion to the procedure outlined in Section 4.1.

In order to determine whether $w \in L$, we first determine $l_k(w)$, $r_k(w)$, and $I_k(w)$, which is trivial to do from the words we maintain. We then feed this into the appropriate decision procedure to determine whether $w \in L$. For small values of k, this can best be done by simply maintaining an array of Boolean values indexed by $\langle l_k(w), I_k(w), r_k(w)\rangle$. For larger values of k, we can simply check directly in the automaton whether the sets are valid for a word in L. Both of these checks take constant time with respect to the length of w, and hence we have:

Theorem 18 DYNREG *can be solved in constant time and space for locally testable regular languages.*

In generalizing to lazy ECFG validation, however, we cannot combine the counters, as we could with strictly locally testable languages. Therefore:

Theorem 19 *Lazy validation can be performed in constant time and linear space for locally testable ECFGs.*

5.2 Determining the Order of Local Testability

Kim et al [23] gave an algorithm which determines in time $O(|Q|^2)$ whether a DFA accepts a locally testable language. However, the problem of determining the *order* of local testability is, in fact, NP-hard, as demonstrated by Kim and McNaughton [22].

Two points should be made. First, it is not strictly necessary for us to find the order k of local testability, since clearly for any $k' \geq k$, the language is also k'-testable. Second, in practice, k is generally small, and hence we could simply test for increasing values of $k \in \{1, 2, \ldots\}$; once we reached some *a priori* bound K, we could switch to a more general validation algorithm.

To compute the order of local testability, we have several options, all due to various results of Trahtman:

- If we have the syntactic monoid of the language, then we can use the algorithm in [34] to determine the order of testability. This algorithm is quite efficient ($O(n^2)$, where n is the order of the monoid), but the size of the monoid can be exponential in the size of the DFA, which means that in practice this is quite expensive.

- Alternatively, if we have the DFA for the language in question (a reasonable assumption in practice), we can use another algorithm [35] to find the exact order. However, this algorithm is also quite expensive in the worst case — $O(n^{n+2})$ time, where n is the number of states in the DFA. In the same paper, Trahtman gives an algorithm to test k-testability in time $O(n^{k+1})$; hence, we can reasonably quickly test testability for small values of k, which are the values in which we are primarily interested in practice.

- Another option is to compute an upper bound on the order, and use this. There is a tight upper bound of $(n^2 - n)/2 + 1$ on a DFA with n states, again given by Trahtman [36, 37]. However, in most cases, we can find better bounds, with the algorithms of [35]. If this bound is fairly small, then it is probably good enough for practical use.

Even though the above procedures seem extremely expensive, the size of schemas in practice means that local testability can be determined very quickly.

6 Dynamic Membership for Star-Free Regular Languages

6.1 A Dynamic Predecessor/Successor Structure

In this section, we define a dynamic $O(\log \log n)$ predecessor/successor structure required for the incremental validation of star-free regular languages. Suppose we have a linked list of n items, and a finite set of markers M. The problem is to maintain a data structure that allows the following operations:

- $\textsc{Insert}(x, y)$: insert y after the item pointed to by x in the linked list.
- $\textsc{Delete}(x)$: delete the item pointed to by x from the linked list.
- $\textsc{Mark}(x, m)$: mark the pointer x with the value $m \in M$.
- For each $m \in M$, $\textsc{Predecessor}_m(x)$: return a pointer y to the first item marked with m to the left of x.
- For each $m \in M$, $\textsc{Successor}_m(x)$: return a pointer y to the first item marked with m to the right of x.

We will use the symbols $-\infty$ and ∞ to denote the extremities of the linked list. Thus, $\textsc{Successor}_m(-\infty)$ returns the first item in the list marked with m.

Our solution to this problem is based upon a combination of van Emde Boas priority queues [38] and list labelling algorithms [4]. Briefly, the van Emde Boas priority queue solves the predecessor/successor problem for a fixed universe of size u in time $O(\log \log u)$ and space $O(u)$. List labelling algorithms label elements in a dynamic linked list with integers such that the relative order of items in the list is reflected by the order of the integers. We will use the algorithm of Bender et al [4], which, with a tag universe size of size $O(n^{1+\epsilon})$, performs list labelling in amortized $O(\log n)$ time per insertion.

We now define a data structure which solves this problem in $O(\log \log n)$ amortized time and $O(n)$ space. For simplicity, we first consider the case where n varies between some fixed bounds $u/2 \leq n \leq 2u$. We divide the linked list of n items into $\Theta(n/\log n)$ blocks with $\Theta(\log n)$ consecutive items in each block. For each item x in the linked list, we maintain $\textsc{Parent}(x)$, a pointer to the block it resides in, and $\textsc{Tag}(x)$, its relative position in the block. For each block B and each $m \in M$, we maintain $\textsc{Tag}(B)$, the block's position relative to other blocks, and a balanced binary tree, $\textsc{Items}_m(B)$, indexed by the tags of the items, which store the items of type m in the block B. Finally, for each $m \in M$, we maintain a van Emde Boas structure $\textsc{Blocks}_m$, storing the blocks in which an item marked with m occur.

We implement the operations as follows:

- $\textsc{Insert}(x, y)$: Insert y into the block B containing x. If this block is full, then split it into two blocks of equal size (also splitting $\textsc{Items}_m(B)$ for each $m \in M$), and insert the

new block into the list of blocks. Block overflow only occurs every $\Theta(\log n)$ insertions, and splitting a block costs $O(\log n \log\log n)$ time on an amortized basis, as the list labelling algorithm will generate $\Theta(\log n)$ relabellings. Hence, the total amortized cost is $O(\log\log n)$.

- DELETE(x): Delete x from its block B, and from $\text{ITEMS}_m(B)$ for each $m \in M$. If B becomes empty, delete B from $\text{BLOCKS}_m(B)$. Each of these steps takes at most $O(\log\log n)$ time.
- MARK(x): If B is the block in which x occurs, add x to items $\text{ITEMS}_m(B)$, and add B to $\text{BLOCKS}_m(B)$.
- $\text{PREDECESSOR}_m(x)$: First, check if a predecessor exists in the parent block B of x, by traversing $\text{ITEMS}_m(B)$ in time $O(\log\log n)$; if not, look in the van Emde Boas structure, again with cost $O(\log\log n)$.
- $\text{SUCCESSOR}_m(x)$: Similar to predecessor.

The only remaining difficulty is to handle when n moves outside the fixed range $u/2 \leq n \leq 2u$. In this case, we choose a new tag universe size $u = O(n^{1+\epsilon})$ and reconstruct the entire data structure using the new value of u. It is clear that this only adds $O(\log\log n)$ overhead on an amortized basis.

6.2 Dynamic Membership for Piecewise Testable Languages

As an introduction to our techniques for general star-free regular languages, we first consider the DYNREG problem for another important subclass of star-free regular languages:

Definition 20 (Piecewise Testable Language) *A regular language L is piecewise testable if and only if it is a finite Boolean combination of sets of the form:*

$$\Sigma^* a_1 \Sigma^* a_2 \Sigma^* \ldots \Sigma^* a_n \Sigma^*, \text{ for } a_i \in \Sigma \tag{1}$$

As we will be using these languages as an example only, we will not be concerned with determining piecewise testability (this is a well-researched problem). Let us assume we have a language as specified in Equation 1. In this case, we maintain a predecessor/successor data structure, as described in Section 6.1, with the set of markers $M = \{a_1, \ldots, a_n\}$. Upon insertion and deletion of characters in the string, we update this structure, marking new characters as appropriate. The string is then accepted if and only if:

$$\text{SUCCESSOR}_{a_n}(\ldots(\text{SUCCESSOR}_{a_2}(\text{SUCCESSOR}_{a_1}(-\infty)))\ldots) \neq \infty$$

For more general piecewise testable languages, which consist of Boolean combinations of languages of the form given in Equation 1, we simply maintain the above data structure for each component, and combine the results from each structure using the appropriate Boolean function. Thus, for piecewise testable languages, we can perform validation in amortized time $O(\log\log n)$ and linear space.

We will be interested in a slightly more general type of language, which we will call a *weakly piecewise testable* language:

Algorithm 1 Determine whether the current word $w = a_1 a_2 \dots a_l$ belongs to a language L of the form given in Equation 2.

```
IS-MEMBER
 1  if w does not start with w_0 or end with w_n then
 2      return false
 3  x ← a_{|w_0|} (this can be maintained incrementally)
 4  for i ∈ {1, n} do
 5      y ← min{SUCCESSOR_a(x) | a ∈ Σ − Σ_{i−1}} (minimum taken with respect to tag value)
 6      z ← SUCCESSOR_{w_i}(x)
 7      if TAG(y) < TAG(z) then
 8          return false
 9      x ← z advanced |w_i| − 1 times
10  return true
```

Definition 21 (Weakly Piecewise Testable Language) *A regular language L is weakly piecewise testable if and only if it is a finite Boolean combination of sets of the form:*

$$w_0 \Sigma_0^* w_1 \Sigma_1^* w_2 \Sigma_2^* \dots \Sigma_{n-1}^* w_n \tag{2}$$

where each $\Sigma_i \subseteq \Sigma$, $w_i \in \Sigma^*$, and $w_i \neq \epsilon$ for $i \in \{1, \dots, n-1\}$.

Let us consider a language L of the form in Equation 2. We can validate such languages incrementally by maintaining a predecessor/successor data structure with marker set $M = \{w_1, \dots, w_{n-1}\} \cup \bigcup_{i=0}^{n-1} (\Sigma - \Sigma_i)$. As characters are inserted into the string, we again insert them into the data structure. A new position x is marked either if their value v lies in $\Sigma - \Sigma_i$ for some $i \in \{0, \dots, n-1\}$, or if there is a substring beginning at x which coincides with one of the words w_i for $i \in \{1, \dots, n-1\}$. Finally, we also maintain two Boolean flags, indicating whether the string begins and ends with w_0 and w_n respectively. Testing whether the string is in L can then be determined by running the procedure given in Algorithm 1. This procedure clearly runs in time independent of the length of the string in question. Note also that several optimizations are possible, but have been omitted for clarity.

Boolean combinations of such languages can be handled in an analagous fashion to piecewise testable languages. Thus, we have the following:

Theorem 22 DYNREG *can be solved for weakly piecewise testable languages in $O(\log \log n)$ time and $O(n)$ space.*

6.3 Dynamic Membership for General Star-Free Regular Languages

There are two approaches we can take when handling dynamic membership for star-free expressions. The first is to obtain the syntactic monoid for the expression, and use the algorithm of Frandsen et al [17] (replacing the use of a van Emde Boas priority queue with the structure of Section 6.1). However, their method requires the use of the Krohn-Rhodes decomposition of an aperiodic finite monoid, which is non-trivial, and potentially very large. Thus, their method is not amenable to efficient or straightforward implementation. The approach we give is automata-based, and provides greater intuitive insight into the complexity results we obtain,

as well as providing a more straightforward implementation. Along the way, we find a new characterization of star-free regular languages.

Our method is based upon the following class of alternating finite automata:

Definition 23 (Almost Loop-Free AFA) *An alternating finite automaton* $\langle Q, \Sigma, \delta, q_I, F\rangle$ *is almost loop-free if and only if both of the following two conditions are met:*

1. *We can find a mapping* $\phi : Q \to \mathbb{Z}_{|Q|}$ *such that for all* $q \in Q$*, the function* δ_q *does not depend on any* q' *such that* $\phi(q') < \phi(q)$*. More specifically,* δ_q *depends on a state* q' *if for some* $a \in \Sigma$*, there exists* $v, v' \in \{0,1\}^Q$ *which agree on all coefficients except* $v_{q'}$*, such that* $\delta_q(a, v) \neq \delta_q(a, v')$*.*
2. *Let* X_q *be the variable representing state* q*. Then for all* $q \in Q$ *such that* δ_q *depends on state* q*,* δ_q *is equivalent to some function* $X_q \vee \delta'_q$*, where* δ'_q *is not dependent on state* q*.*

Intuitively, in an almost loop-free automaton we can reorder the states such that the transition function for a particular state does not depend on any of the states that come before it. This type of automaton is "almost" loop-free because the only loops that can occur in the dependency graph are self-loops of a certain kind. (We will discuss the need for the second condition momentarily.) For example, the AFA given in Figure 2 is almost loop-free (the states are already appropriately ordered in the figure).

We are interested in almost loop-free AFAs because of the following theorem[3]:

Theorem 24 (Salomaa and Yu, corrected [29]) *A regular language is star-free if and only if can be represented by an almost loop-free alternating finite automaton.*

This theorem sheds light on the second condition in the definition above. If we did not impose this condition, then the following AFA would be almost loop-free:

$$X = a \cdot \neg X + 1$$

This accepts the language $(aa)^*$, which is not star-free. Fortunately, it also fails to satisfy the second constraint in the above definition.

The proof of Theorem 24 in [29] is constructive, and hence, given a star-free regular expression e, we can easily obtain an equivalent almost loop-free AFA. However, suppose we have a star-free regular language L and some non-star-free regular expression e for L; how do we obtain a star-free regular expression e' for L? There exist automated procedures for doing this, see, e.g., Perrin [28]. However, this is a fundamentally difficult problem, regardless of whether we start with a regular expression for L [5] or an automata for L [11]. Fortunately, in practice we can often apply heuristics to obtain a star-free expression. For example, for some subset $A \subseteq \Sigma$, a star-free expression equivalent to Σ'^* is $\neg\emptyset \cdot (\Sigma - \Sigma') \cdot \neg\emptyset$.

Thus, given a star-free regular language L, it is relatively straightforward to obtain a corresponding almost loop-free AFA. The next theorem shows why an almost loop-free AFA is useful:

Theorem 25 *An almost loop-free AFA accepts a weakly piecewise testable language.*

[3]While the procedure outlined in [29] is correct, the stated result and corresponding proof is not; we present here a refined statement for which their proof holds.

Proof: Our proof will be an induction on the number of equations in the equational representation E of an almost loop-free automaton. In the following, we will number the equations in $E = \{X_1, \ldots, X_n\}$ such that X_i depends only on equations X_j for $j \geq i$.

First, suppose $|E| = 1$. In this case, we have only one equation, $X = \sum_{a \in \Sigma} a \cdot \delta(a, X) + f$. The Boolean functions $\delta(a, X)$ are over only the variable X, and hence can only be equivalent to X, 1, or 0 ($\neg X$ is not possible due to the second condition in the definition of almost loop-free AFA). Let Σ_X, Σ_1, and Σ_0 be the subsets of a corresponding to the three different types of functions. Then it is easy to see that the language accepted by E is:

$$(\Sigma_X^* \cdot \epsilon(f)) \cup (\Sigma_X^* \cdot \Sigma_1) \cup \neg(\Sigma_X^* \cdot \Sigma_0) \tag{3}$$

In the above, $\epsilon(1) = \epsilon$, $\epsilon(0) = \emptyset$. This language is clearly weakly piecewise testable.

Now suppose that $|E| = n$, and that the inductive hypothesis is true for all values $|E| < n$. We have:

$$X_1 = \sum_{a \in \Sigma} a \cdot \delta_1(a, X_1, \ldots, X_n) + f_1$$

By the second condition in the definition of an almost loop-free automaton, we can rewrite this as:

$$X_1 = \sum_{a \in \Sigma} a \cdot \delta_1'(a, X_1) + \sum_{a \in \Sigma} a \cdot \delta_1''(a, X_2, \ldots, X_n) + f_1$$

As we did in the base case, let Σ_{X_1}, Σ_1, and Σ_0 be the three subsets corresponding to the possible definitions of $\delta_1'(a, X_1)$. Let L_1 be the language constructed from these subsets, as defined by Equation 3.

The sets $E_i = \{X_i, \ldots, X_n\}$ are all of size less than n, and each represent an almost loop-free AFA. Therefore, we may apply the inductive hypothesis to obtain weakly piecewise testable languages L_i accepted by E_i. We can combine L_1 with $L_2, L_3, \ldots, L_n$ into the following language L:

$$L_1 + \sum_{a \in \Sigma} \Sigma_{X_1}^* \cdot a \cdot \delta_1(a, L_2, \ldots, L_n)$$

It is easy to verify that the above language is the solution to E. Moreover, it is weakly piecewise testable, as can be observed by applying standard transformations to L (e.g., $(A \cap B) \cdot C = (A \cdot C) \cap (B \cdot C)$). Thus we have proved the inductive hypothesis. □

The above theorem can be restated to reveal the following interesting characterization of star-free languages:

Corollary 26 *A regular language is star-free if and only if it is weakly piecewise testable.*

As the proof of the above theorem is constructive, we can thus obtain a weakly piecewise testable representation of any star-free regular language, and use the results of the previous section to solve DYNREG incrementally in $O(\log \log n)$ time and $O(n)$ space. This carries over directly to the schema validation problem:

Theorem 27 *Lazy validation can be performed on star-free schemas in amortized $O(\log \log n)$ time and linear space.*

From an efficiency point of view, it would be useful to minimize the number of Boolean operations involved in the decomposition of a star-free language. This problem is strictly harder than the minimization of a Boolean function, which is coNP-hard [20]. In practice, *ad hoc* techniques can be used to reduce the size of the expression substantially.

Finally, we note a few optimizations are possible for some special cases. Consider a weakly piecewise component $L = w_0\Sigma_0^* \ldots \Sigma_{n-1}^* w_n$. We can find a constant time algorithm for this component in the following cases:

- If $n < 3$, then L is locally testable. In some cases when $n = 3$, L is also locally testable. More generally, we can sometimes use techniques similar to those used for locally testable languages (e.g., the language $\Sigma^*01\Sigma^*01\Sigma^*$ consists of all strings in which 01 occurs at least twice).
- If the characters occuring in each word w_i are mutually exclusive with each Σ_i, then instead of maintaining the structures listed above, we can simply maintain order maintenance information [4, 14], and a linked list of all occurrences of each $w \in W$ (order does not matter). Upon a query, if this list has more than n items, return false. If it has less than n items, then sort the list and ensure the words appear in the correct order.

7 A General Solution for DynReg

7.1 An Optimal Algorithm for DynReg

The only incremental validation algorithm for DTDs with good worst-case behavior for *all* DTDs is that of Papakonstantinou and Vianu [25]. This algorithm, which has $O(\log n)$ worst and average case performance, is quite practical. We now demonstrate that a more complicated algorithm can yield an improvement by a factor of $O(\log\log n)$.

The basic approach was outlined by Frandsen et al [17]. Let L be a regular language, $A_L = \langle Q_L, \Sigma, \delta_L, I_L, F_L\rangle$ be its minimal automaton, M_L be its syntactic monoid, and ϕ be the natural map from Σ^* to M_L. Then, for any word w in L, $x \in L$ if and only if $\phi(w)(I_L) \in F_L$. Hence, any solution to the dynamic word problem for finite monoids can be used to solve the dynamic membership problem for regular languages. The most general solution to the dynamic word problem is based on the prefix sum problem, which has a $\Theta(\log n/\log\log n)$ solution. In our case, we wish to vary n, which is not typically considered in the literature.

Our solution is a modification of the data structure given by Dietz [13] to solve the list indexing problem, closely related to the prefix sum problem. First, let ϵ be a constant strictly less than 1. For a machine with word size $O(\log n)$, Dietz gave a simple solution for the partial sum problem for lists of length $O(\log^\epsilon n)$, taking amortized $O(1)$ time per operation. The idea is to cache updates in a separate data structure, and use a lookup table to allow these updates to be inserted into the main structure quickly.

A *weight balanced tree* is a tree with branching factor $\Theta(\log^\epsilon n)$ ($\epsilon < 1$), such that the leaves of the tree are at the same depth. Each node n in the tree is assigned a weight $w(n)$, such that the number of descendants of n is $O(w(n))$. Dietz outlined how such a structure could be maintained under insertions and deletions, such that the height remained $\Theta(\log n/\log\log n)$, in amortized time $\Theta(\log n/\log\log n)$.

Thus, we store our word w in a weight balanced tree, with the letters of w making up the leaves. At each internal node of the tree, we maintain two prefix sum data structures over the children $x_1, \ldots, x_b$ of the node. In the first prefix sum data structure, we maintain $w(x_i)$, the number of descendants of x_i. These values are used as the weight function for the tree. The

second prefix sum data structure maintains the product of syntactic monoid elements. More precisely, for a child x, which has leaf nodes $a_1, \ldots, a_m$, we store in this data structure the value $\phi(a_1 \ldots a_m)$.

Upon an insertion or a deletion, we update both of these data structures for each node on the root-to-leaf path in question. Each update takes only constant time (since the data structures are over lists of size $\Theta(\log^\epsilon n)$), and hence the total time is $\Theta(\log n / \log\log n)$. To determine membership of the word w in the language L, it suffices to test the product stored in the root node of the tree.

7.2 A Tight Lower Bound for Incremental DTD Validation

In this section, we demonstrate that for some regular languages, the algorithm of Section 7.1 is the best possible. We do this by giving a reduction to the $\mathbb{Z}_2$ prefix sum problem: given an array $A[n]$ of length n of entries from $\mathbb{Z}_2$, for any i compute $\sum_{j=1}^{i} A[j]$ while also allowing entries to be changed.

Suppose we can incrementally validate $L = (00)^*10^*$ in $O(f(n))$ time. Then we can compute a solution to the $\mathbb{Z}_2$ prefix sum problem as follows. Let $A[n]$ be the array of values. We maintain a linked list of 0s, and an array $P[n]$ of pointers into the linked list. The linked list is initialized to the string $(00)^n$ (that is, the string of $2n$ zeros), and pointer $P[i]$ points to the $2i$-th entry in the list. Here is how we handle each operation:

- $A[i] \leftarrow A[i] + \delta$: If $\delta \equiv 0 \bmod 2$, do nothing. If $P[i-1]$ points to the item preceding $P[i]$, then insert a new 0 before $P[i]$. Otherwise, delete the 0 before $P[i]$.
- Compute $\Sigma_{j=0}^{i} A[j]$: Insert a 1 at the location pointed by by $P[i]$. Check if the string belongs to L: if it does, we return 0, otherwise we return 1. We then remove the 1 from the list.

The length of the list is $O(n)$, and if m prefix sum operations are performed, then $O(m)$ DynReg operations are performed. Hence, we can solve the $\mathbb{Z}_2$ prefix sum problem in $O(f(n))$ time. Therefore, L cannot be validated in less than $\Omega(\log n / \log\log n)$ time, by the lower bound of Fredman and Saks [18].

We note in passing that L is 1-unambiguous, showing that this is not a useful concept when considering incremental validation. We also note that this result only holds in the insertion/deletion model of membership testing; if we follow Frandsen et al [17], and allow only updates, then DynReg can be easily solved in constant time for this language.

8 Non-Recursive Schemas

The above schemes have good theoretical performance. However, an implementation of one of the above schema validation algorithms can end up using a substantial amount of space. This is because each node in the database must maintain an appropriate data structure for the validation information pertaining to its children. If nodes have very few children, then the constant factor in the space cost can become significant.

For non-recursive schemas, we can do much better, without affecting the overall theoretical performance. In practice, the vast majority of schemas are non-recursive, and hence this is a useful strategy. In fact, our solution can be even be applied to recursive schemas, by restricting its use to only the non-recursive portions of the schema.

The basic idea is that, for non-recursive schemas, we can validate the document as a string rather than as a tree. Suppose we have a document over the alphabet of element labels Σ. Let

$\{\uparrow_i \mid i \in \mathbb{Z}+\}$ be some set of symbols disjoint from Σ. Then for each document over Σ, we can find a canonical string representation over $\Sigma \cup \{\uparrow_i \mid i \in \mathbb{Z}+\}$. For instance, consider the document:

```
<a>><b><c/><c/></b><d/></a>
```

We can represent this document as the string:

$$abc \uparrow_3 c \uparrow_3\uparrow_2 d \uparrow_2\uparrow_1$$

For a non-recursive ECFG, we can find an equivalent regular language L over the set of strings constructed in the above manner. The transformation simply entails appending to each regular expression in the ECFG the appropriate $\uparrow_i$ (where i is the level at which this type occurs in documents), and then reducing the ECFG to a single regular expression through substitutions. If a particular production in the ECFG occurs at different levels in the schema, then we simply make copies. For example, consider the ECFG:

$$\begin{aligned} a &= b^* \\ b &= cc \end{aligned}$$

We can reduce this to the regular language:

$$a(bc \uparrow_3 c \uparrow_3\uparrow_2)^* \uparrow_1$$

Validation for XML documents conforming to this schema then reduces to validation of strings over this regular expression. We can apply all the techniques of the previous sections, but this time using only a single set of data structures, instead of one set per node.

The following theorem demonstrates why this technique is useful:

Theorem 28 *The above procedure applied to a star-free ECFG yields a star-free regular language.*

Proof: Since the concatenation, negation, union, and intersection of star-free languages is again star-free, we only need to consider the Kleene closure. Suppose we have some star-free regular language L, and we construct $L' = (L \uparrow)^*$, where $\uparrow$ is a symbol not occurring in any word in L. We wish to show L' is star-free.

If L' were not star-free, then for any positive integer n, we can find words v, w, x, such that $vw^n x \in L'$ but $vw^{n+1}x \notin L'$. Let us consider any $n > 1$, and consider v, w, x satisfying this condition. For some k, we have:

$$vw^n x = w_1 \uparrow w_2 \uparrow \ldots w_k \uparrow \tag{4}$$

Let us first suppose that w^n does not contain a $\uparrow$, and hence must be a substring of some w_i. In this case, we can then find words v', x', such that $w_i = v'w^n x' \in L$, but $w_i = v'w^{n+1}x' \notin L$, which could not hold for *all* values of $n > 1$, because L is star-free. Hence, we can always find an $n > 1$ such that we can find some n, v, w, x such that w^n contains a $\uparrow$. Clearly, if w^n contains an occurrence of a $\uparrow$, then so must w. Therefore, w must be of the following form:

$$w = y \uparrow \ldots \uparrow z$$

In the above, neither y nor z contain an $\uparrow$. Because $n > 1$, we must thus have that $zy \in L$, because $vw^n x \in L'$. But this implies that $vw^{n+1} x \in L'$, which is a contradiction. Hence L' must be star-free. □

Of course, this optimization is not always useful. If the ECFG is strictly locally testable, then we can already validate with only constant space. There is also the possibility that the regular language we construct requires the use of a more complicated validation algorithm. However, there are still several advantages to using this technique. First, there is a nice synergy with the region algebras used in many XML databases, because the $O(\log \log n)$ data structure makes use of the region tags which are generally used heavily in such systems. Secondly, the above technique allows one to apply our techniques to non-recursive RELAX NG schemas: any reference an element t can be replaced with the regular expression $t_1 | \ldots | t_n$, where the t_i are the types defined for elements with label t.

9 Conclusions and Open Problems

In this paper, we have continued the investigation of incremental schema validation initiated by previous work [2, 25]. We have given four levels in the complexity hierarchy for this problem, and effective solutions for determining in which category an arbitrary schema belongs. We have also highlighted a method which greatly simplifies the practical implementation of these ideas for non-recursive schemas, which are the most common type in practice.

The problem of dynamic regular language membership testing is an interesting one, and one we believe is fundamental to several areas of computer science. There are several interesting open problems:

- A complete classification of the complexity of DynReg would be of interest. There are several classes of languages that we have not considered here for which constant time algorithms exist. For instance, non-star-free languages such as $(00)^*$ can be validated incrementally very easily (these lie within the class of languages with commutative syntactic monoids [17]). There also exist star-free languages which we have not considered, for example, the *locally threshold testable* languages [3], for which constant time algorithms can be used, using similar techniques to those given in this paper. More generally, we can verify incrementally in constant time any language which is completely determined by some Boolean function acting on the counters we maintain for $l_k(w)$, $I_k(w)$, and $r_k(w)$. Characterizing these languages and finding a suitable Boolean function would likely be an interesting result.

- The differences between the updates only model of Frandsen et al [17] and the insertion/deletion model considered here should be determined.

- It would be useful to know how much easier strict validation is than lazy validation.

- Finally, an investigation of the incremental schema validation problem for specialized DTDs, such as RELAX NG schemas, would also have practical applications.

10 Acknowledgements

The author would like to thank Matthew Gebski for several helpful discussions on topics related to this paper.

References

[1] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. Typechecking XML views of relational databases. *ACM Transactions on Computational Logic*, 4(3):315–354, July 2003.

[2] D. Barbosa, A. O. Mendelzon, L. Libkin, L. Mignet, and M. Arenas. Efficient incremental validation of XML documents. In *ICDE 2004*, pages 671–682. IEEE Computer Society, 2004.

[3] Danièle Beauquier and Jean-Eric Pin. Factors of words. In *ICALP 1989*, volume 372 of *Lecture Notes in Computer Science*, pages 63–79. Springer, 1989.

[4] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *ESA 2002*, volume 2461 of *Lecture Notes in Computer Science*, pages 152–164. Springer, 2002.

[5] László Bernátsky. Regular expression star-freeness is PSPACE-complete. *Acta Cybernetica*, 13(1):1–22, 1997.

[6] B. Bouchou and M. H. F. Alves. Updates and incremental validation of XML documents. In *DBPL 2003*, volume 2921 of *Lecture Notes in Computer Science*, pages 216–232. Springer, 2004.

[7] T. Bray et al. Extensible markup language (XML) 1.0 (third edition). http://www.w3.org/TR/2004/REC-xml-20040204/, February 2004.

[8] Anne Brüggemann-Klein and Derick Wood. One-unambiguous regular languages. *Information and Computation*, 142(2):182–206, 1998.

[9] Peter Buneman, Susan B. Davidson, Wenfei Fan, Carmem S. Hara, and Wang Chiew Tan. Keys for XML. *Computer Networks*, 39(5):473–487, August 2002.

[10] P. Caron. Families of locally testable langauges. *Theoretical Computer Science*, 242(1–2):361–376, July 2000.

[11] S. Cho and Dung T. Huynh. Finite-automaton aperiodicity is PSPACE-complete. *Theoretical Computer Science*, 88(1):99–116, 1991.

[12] J. Clark and M. Murata. RELAX NG specification. http://www.relaxng.org/spec-20011203.html, December 2001.

[13] P. F. Dietz. Optimal algorithms for list indexing and subset rank. In *WADS 1989*, volume 382 of *Lecture Notes in Computer Science*, pages 39–46. Springer, 1989.

[14] Paul F. Dietz and Daniel D. Sleator. Two algorithms for maintaining order in a list. In *STOC 1987*, pages 365–372. ACM Press, 1987.

[15] Guozhu Dong and Jianwen Su. Incremental maintenance of recursive views using relational calculus/SQL. *SIGMOD Record*, 29(1):44–51, 2000.

[16] Samuel Eilenberg. *Automata, Languages, and Machines*, volume A. Academic Press, 1974.

[17] G. S. Frandsen, P. B. Miltersen, and S. Skyum. Dynamic word problems. *Journal of the ACM*, 44(2):257–271, 1997.

[18] M. L. Fredman and M. E. Saks. The cell probe complexity of dynamic data structures. In *STOC 1989*, pages 345–354. ACM Press, May 15–17 1989.

[19] Alan Halverson et al. Mixed mode XML query processing. In *VLDB 2003*, pages 225–236. Morgan Kaufmann, 2003.

[20] E. Hemaspaandra and G. Wechsung. The minimization problem for boolean formulas. *SIAM Journal on Computing*, 31(6):1948–1958, 2002.

[21] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A native XML database. *The VLDB Journal*, 11(4):274–291, 2002.

[22] S. M. Kim and R. McNaughton. Computing the order of a locally testable automaton. *SIAM Journal on Computing*, 23(6):1193–1215, December 1994.

[23] S. M. Kim, R. McNaughton, and R. McCloskey. A polynomial time algorithm for the local testability problem of deterministic finite automata. *IEEE Transactions on Computers*, 40(10):1087–1093, October 1991.

[24] R. McNaughton and S. Papert. *Counter-Free Automata*. MIT Press, 1971.

[25] Y. Papakonstantinou and V. Vianu. Incremental validation of XML documents. In *ICDT 2003*, volume 2572 of *Lecture Notes in Computer Science*, pages 47–63. Springer, 2002.

[26] Yannis Papakonstantinou and Victor Vianu. DTD inference for views of XML data. In *PODS 2000*, pages 35–46. ACM Press, 2000.

[27] Sushant Patnaik and Neil Immerman. Dyn-FO: A parallel, dynamic complexity class. In Victor Vianu, editor, *PODS 1994*, pages 210–221. ACM Press, 1994.

[28] D. Perrin. Finite automata. In *Handbook of Theoretical Computer Science*, volume B, chapter 1, pages 1–57. Elsevier and MIT Press, 1990.

[29] K. Salomaa and S. Yu. Alternating finite automata and star-free languages. *Theoretical Computer Science*, 234(1–2):167–176, March 2000.

[30] Marcel Paul Schützenberger. On finite monoids having only trivial subgroups. *Information and Control*, 8(2):190–194, April 1965.

[31] L. Segoufin. Typing and querying XML documents: some complexity bounds. In Frank Neven, editor, *PODS 2003*, pages 167–178. ACM Press, 2003.

[32] L. Segoufin and V. Vianu. Validating streaming XML documents. In *PODS 2002*, pages 53–64. ACM Press, 2002.

[33] H. S. Thompson et al. XML schema part 1: Structures. http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/, May 2001.

[34] A. N. Trahtman. A polynomial time algorithm for local testability and its level. *International Journal of Algebra and Computation*, 9(1):31–39, 1999.

[35] A. N. Trahtman. Algorithms finding the order of local testability of deterministic finite automaton and estimations of the order. *Theoretical Computer Science*, 235(1):183–204, March 2000.

[36] A. N. Trahtman. Optimal estimation on the order of local testability of finite automata. *Theoretical Computer Science*, 231(1):59–74, January 2000.

[37] A. N. Trahtman. Erratum to "optimal estimation on the order of local testability of finite automata". *Theoretical Computer Science*, 255(1–2):697, March 2001.

[38] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, June 1977.

[39] S. Yu. Regular languages. In *Handbook of Formal Languages*, volume 1, chapter 1, pages 41–110. Springer-Verlag New York, Inc., 1997.