Efficient Incremental Validation of XML Documents

Denilson Barbosa Alberto O. Mendelzon Leonid Libkin Laurent Mignet
Marcelo Arenas
University of Toronto
{dmb,mendel,libkin,mignet,marenas}@cs.toronto.edu

Abstract

We discuss incremental validation of XML documents with respect to DTDs and XML Schema definitions. We consider insertions and deletions of subtrees, as opposed to leaf nodes only, and we also consider the validation of ID and IDREF attributes. For arbitrary schemas, we give a worst-case $n \log n$ time and linear space algorithm, and show that it often is far superior to revalidation from scratch. We present two classes of schemas, which capture most real-life DTDs, and show that they admit a logarithmic time incremental validation algorithm that, in many cases, requires only constant auxiliary space. We then discuss an implementation of these algorithms that is independent of, and can be customized for different storage mechanisms for XML. Finally, we present extensive experimental results showing that our approach is highly efficient and scalable.

1. Introduction

The problems of storing and querying XML documents have attracted a great deal of interest. Other aspects of XML data management, however, have not yet been satisfactorily explored; among them is the problem of checking that documents are valid with respect to their specifications, and that they remain valid after updates. Both integrity checking and incremental integrity checking are classical topics in the relational setting (see, e.g., [6, 5] and references therein). In the XML context, researchers are just beginning to look at update languages [17], validation [14], and incremental validation of specifications [11, 16]. In this paper, we propose and evaluate efficient incremental validation algorithms for XML data.

One popular form of XML document specification is the Document Type Definition (DTD). A DTD D is a grammar that defines a set of documents $L(D)$; each document in $L(D)$ is said to be valid with respect to D. The *validation* problem is: given a DTD D and an XML document X, is it the case that $X \in L(D)$? When a valid document is updated, it must be revalidated; let U be some update operation. The *incremental validation* problem is: given $X \in L(D)$, is it the case that $\mathsf{U}(X) \in L(D)$?

We start by discussing the validation problem with respect to DTDs in Section 2. We assume, following the XML standard, that the regular expressions used in DTDs are *1-unambiguous*; informally, this means that documents can be parsed with a single symbol lookahead. Although the literature tends to view XML documents as trees, the existence of node identifiers (called *ID attributes*) and references to identifiers (called *IDREF attributes*) turns these trees into graphs. We separate the validation problem into two parts: validating the structural constraints imposed by the DTD, which amounts to parsing; and validating the attribute constraints, which amounts to checking that the ID attributes specified in the DTD have unique values across the document and that the IDREF references do not dangle.

In Section 3, we show that both structural and attribute constraints can be validated in $O(n \log n)$ time and linear space. Then we show that, not surprisingly, incremental validation can be highly beneficial by avoiding complete revalidation after each update. We start by giving a general incremental validation algorithm with respect to a reasonable class of updates; although the worst-case bounds on the algorithm match those of revalidation, we argue why it is likely to perform better than revalidation in practice. We then present two classes of DTDs, called *1,2-conflict-free* (1,2-CF) and *conflict-free* (CF), for which incremental validation can be done efficiently in practice.

The class of 1,2-CF DTDs is defined by regular expressions in which no two different occurrences of a symbol in the expression match string positions that occur at distance 1 and 2 from a position matched by some other symbol in the expression. For example, the 1-unambiguous regular expression $a(b|cb)$ is not 1,2-CF because the first b appears immediately after the a in the string ab, and the second b appears at distance 2 from the a in the string acb. The definition of CF DTDs further restricts this, by disallowing repeated symbols in the regular expressions. The time complexity of incremental validation for both classes is essen-

Proceedings of the 20th International Conference on Data Engineering (ICDE'04)
1063-6382/04 $ 20.00 © 2004 IEEE

tially logarithmic, per update; the space complexity differs for these classes: 1,2-CF DTDs require linear space, while CF DTDs only require constant auxiliary space.

The definitions of 1,2-CF and CF DTDs are somewhat technical, and it is natural to ask how common such DTDs are. The empirical evidence to date [4] suggests that in fact most DTDs found on the web belong to the class of CF DTDs (and thus 1,2-CF as well). Also, as we show in Section 3, our approach applies to a large subset of XML Schema [20] as well.

Section 4 describes an implementation of our algorithms and discusses how they could be coupled with storage mechanisms for XML. In Section 5 we present experimental results on our prototype implementation. We show that the advantages of incremental over complete revalidation are substantial. Finally, we discuss related work in Section 6 and conclude in Section 7.

In summary, the contributions of this paper are: an analysis of incremental validation of XML documents with respect to DTDs and XML Schema specifications, including structural as well as ID/IDREF constraints; the introduction of two restricted classes of DTDs that allow very efficient incremental validation; practical methods for incremental validation of documents; and an extensive experimental evaluation that shows the feasibility of incremental validation for "realistic" DTDs.

2. DTDs and regular expressions

As mentioned in the Introduction, DTDs specify two kinds of constraints: *structural* constraints, given by element declaration rules, and *attribute* constraints, given by attribute declaration rules. (See [19] for details.) We treat validation of these two kinds of constraints separately.

As customary, the structural constraints of DTDs are abstracted as extended context-free grammars, that is, context-free grammars where the right hand side of each production contains a regular expression. An XML document is valid with respect to the structural constraints of a DTD if its abstraction as a tree represents a derivation tree of the extended CFG corresponding to that DTD. Unlike structural constrains which deal with the labels of nodes in the XML tree, attribute constraints deal with the *values* of (attribute) nodes. In particular, one can specify uniqueness of certain attributes and inclusion dependencies among them.

2.1. Validation of structural constraints

Elements are declared in a DTD by rules of the form `<!ELEMENT l` c`>`, which specify that valid elements of type[1] `l` have content conforming to c, which is called a *content model*; five content models are defined in the XML standard. The validation of the #PCDATA, ANY and EMPTY content models can be done trivially. Of greater interest to us is the validation of *element* and *mixed* content models.

An element e has *element* content if it has only other elements (i.e., child elements) as its content. An element content model is specified by a regular expression E whose vocabulary is the set of element labels declared in the DTD. The validity of elements under this content model is decided as follows: an element e whose content model is defined by E is valid if and only if the string formed by concatenating the tags of its children belongs to $L(E)$, the language denoted by E.

An element x has *mixed* content if it has both text (i.e., #PCDATA) and other elements as its content. A mixed content model is specified by a regular expression that matches (#PCDATA(|*Name*)$*$)$*$, where *Name* can be any element label declared in the DTD [19]. It is easy to see that the validation of mixed content models also amounts to testing membership in regular languages, if we denote occurrences of `#PCDATA` content by a special symbol. Thus, without loss of generality, we focus on validation of element content models only throughout the paper.

An XML document is valid with respect to the structural constraints of a DTD if all elements in the document are valid.

The standard procedure for testing membership in a regular language is to simulate the automaton that accepts the language on the input strings. For an input string of length n and an automaton with s states, such simulation can be done in time $O(ns^2)$ and $O(s^2)$ space if the automaton is non-deterministic, and in $O(ns)$ time and constant space if the automaton is deterministic [7]. If the language is described by a regular expression, as is the case with DTDs, one has to produce the corresponding automaton for performing the simulations, and there are standard procedures for doing so as well [7].

2.2. 1-unambiguous regular expressions

The specification of XML DTDs restricts the regular expressions used for defining element content models to be 1-unambiguous [3]. Informally, a regular expression is 1-unambiguous if one can uniquely match an occurrence of a symbol in the regular expression to a character in the input string without looking beyond that character. In other words, 1-unambiguous regular expression require a lookahead of one symbol only.

Let Σ be a finite alphabet of symbols. Our regular expressions are given by the grammar $E := \epsilon \mid a \mid E|E \mid EE \mid E^*$, where a ranges over Σ, with ϵ being the empty

[1] Elements with the same label have the same type in DTDs [19].

string, and $E|E, EE, E^*$ being the union, concatenation, and the Kleene star, respectively.

We use the definition of 1-unambiguous expressions from [3]. First, we mark symbols with subscripts to indicate different occurrences of the same symbol in a regular expression. For instance, a *marking* of the regular expression $a(b \mid cb)$ is $a(b_1 \mid cb_2)$. For expression E, we denote its marking by E'. Each subscripted symbol is called a *position*; we denote by $pos(E)$ the set of all positions in regular expression E. For a given position x, $\chi(x)$ denotes the corresponding (unmarked) symbol in Σ. Finally, the subscripting method used is such that if $F \mid G$ or FG are regular expressions, then $pos(F)$ and $pos(G)$ are disjoint.

Note that we can view a marked regular expression E' as a regular expression over the alphabet of subscripted symbols $pos(E)$, such that each subscripted symbol occurs at most once in E'.

Definition 1 *A regular expression E is* 1-unambiguous *if and only if for all words u, v, w over the subscripted alphabet $pos(E)$ and all x, y in $pos(E)$, the conditions $uxv, uyw \in L(E')$ and $x \neq y$ imply $\chi(x) \neq \chi(y)$.*

In other words, for each word w denoted by a 1-unambiguous regular expression E, there is exactly one marked word in $L(E')$ that corresponds to w, and this word can be constructed incrementally by examining the next symbol of w.

2.2.1. The Glushkov automaton of a regular expression.

One way of representing regular expressions by finite automata was proposed by Glushkov [21]. In the Glushkov automaton of a regular expression E, states correspond to positions of E and transitions connect those positions that can be consecutive in a word in $L(E')$.

First we define, for each regular expression E, the sets $first(E)$, the set of positions that appear as the first symbol of some word in $L(E')$; $last(E)$, similarly for last positions; and $follow(E, x)$, the set of positions that appear immediately after position x in some word in $L(E')$. For technical reasons we define a "virtual" position $\pi, \pi \notin pos(E)$, and $follow(E, \pi) = first(E)$.

Definition 2 *The Glushkov automaton $G_E = (Q, \Sigma, \delta, q_I, F)$ of a regular expression E is defined as follows:*

1. $Q = pos(E) \cup \{q_I\}$;
2. *For* $a \in \Sigma$, *let* $\delta(q_I, a) = \{x | x \in first(E), \chi(x) = a\}$;
3. *For* $x \in pos(E), a \in \Sigma$, *let* $\delta(x, a) = \{y | y \in follow(E, x), \chi(y) = a\}$;
4. $F = \begin{cases} last(E) \cup \{q_I\}, & \textit{if } \varepsilon \in L(E), \\ last(E), & \textit{otherwise} \end{cases}$

It is known [21] that for any regular expression E, $L(E) = L(G_E)$.

Note that there is a 1-1 correspondence between states in the Glushkov automaton and positions in the regular expression.

The Glushkov automaton provides a characterization of 1-unambiguous regular expressions:

Proposition 1 ([3]) *A regular expression E is 1-unambiguous if and only if its Glushkov automaton G is deterministic.*

As it turns out, the Glushkov automaton can be computed in time quadratic in the size of the expression, and for 1-unambiguous regular expressions the size of the automaton is linear in the size of the expression. Thus, testing whether a regular expression is 1-unambiguous can be done in quadratic time [3].

For the rest of paper we will assume that every regular expression is 1-unambiguous.

Extended Transition Function. As customary, we extend the transition function to operate on strings rather than single symbols. Let $G = (Q, \Sigma, \delta, q_I, F)$ be a DFA. The function $\hat{\delta}$ maps a string w in Σ^* into the state of G after reading w. We represent strings by their symbols $w_1 \ldots w_t$, with the understanding that a string is empty if $t < 1$. Then $\hat{\delta}$ is defined inductively by $\hat{\delta}(\epsilon) = q_I$, and $\hat{\delta}(w_1 \ldots w_t) = \delta(\hat{\delta}(w_1 \ldots w_{t-1}), w_t)$. Clearly, $w \in L(E)$ iff $\hat{\delta}(w) \in F$.

2.3. Validation of attribute constraints

Attribute specifications in a DTD are declared by rules of the form `<!ATTLIST l` $\mathtt{a}_1\ t_1\ p_1 \ldots \mathtt{a}_n\ t_n\ p_n$ `>`, specifying that elements of type `l` may have an attribute labeled $\mathtt{a}_i$ of type t_i and participation constraint $p_i, 1 \leq i \leq n$. The type of an attribute a must be one of the following: ID, IDREF, IDREFS, *enumeration* or CDATA. Type CDATA specifies that the value of a is text, while enumeration types explicitly define the set of values that a can assume. Again, the validation of attributes of such types can be done trivially. The ID, IDREF and IDREFS types are used for identifying and establishing references among elements within a document and require more validation effort.

In a valid XML document the following must hold: (1) the values of all ID attributes are unique, (2) the value of each IDREF attribute must be equal to the value of some ID attribute. The IDREFS type is just a multivalued version of the IDREF type; that is, an IDREFS attribute can be viewed as a set of attributes with the same label.

The participation constraint of an attribute is either `#REQUIRED` or `#IMPLIED`. If an attribute is `#REQUIRED`, then it must be defined for all elements of type `l`. Checking the validity of participation constraints is also trivial, thus we will not consider such constraints further.

2.4. XML Schema specifications

XML Schema [20] is another popular language for defining XML specifications proposed by the W3C, which extends DTDs in several ways. The most notable extension is the decoupling of element *labels* and element content models. In a DTD, there can be only one content model specified for elements of a given label; that is, there can be only one regular expression associated with each label in the DTD. In XML Schema, on the other hand, one can specify different content models for elements of the same label depending on the *context* in which the element occurs. Note that element content models in XML Schema are also defined by 1-unambiguous regular expressions [20].

In XML Schema, the context of an element is determined by the label of the element and the context of the parent of the element. For instance, we could specify that c elements have r_1 as their content model whenever they are children of a elements, and r_2 as their content model whenever they are children of b elements. This limited notion of context makes the validation of XML Schema specifications very similar to the validation of DTDs: the only distinction is that, in order to decide which content model to use, one has to consider the context in which the elements appear.

The attribute constraints defined in XML Schema are also a superset of those in DTDs. In particular, XML Schema also defines ID and IDREF attributes, with the same semantics as in DTDs. Therefore, our work applies directly to this subset of XML Schema. For clarity of exposition, we will refer to DTDs only throughout the paper.

3. Incremental validation

We now turn our attention to the incremental validation problem. That is, given an XML document X, valid with respect to a DTD D, and an update operation U, is $\mathsf{U}(X)$ valid with respect to D? Before we do that, we first introduce some notation, state some assumptions and briefly consider the problem of static validation. Static validation becomes a concern when updating XML documents since practical update languages will inevitably allow the insertions of subtrees, which must be (statically) validated.

It turns out that the complexity of validating structural constraints in a DTD depends on whether one navigates the documents using a pointer structure (i.e., a DOM-like interface) or as a string (i.e., in a SAX-like interface) [14]. This discrepancy is important to us for the following reasons. We assume the document will be stored in a database that provides indexed access to elements in the document, as in DOM. However, the SAX interface seems more reasonable for statically validating those XML trees being inserted in the document, since it does not require one to materialize the entire tree in memory prior to validation. We assume that the documents are stored as in DOM, and that the subtrees being inserted are statically validated using a SAX-like interface.

Notation and assumptions. Let X be an XML document and D be a DTD. Throughout the paper we use $n = |X|$, $d = |D|$, and we denote by h the *height* of X; note that $h = O(n)$.

We make the following assumptions about the data structures used for storing the documents: given an address of a node i in X, we have logarithmic access and update time (on n) to i; the parent of i; the label of i; the left and right siblings of i; and the first and last children of i. With respect to the DTDs, we assume that the time to determine the next state of an automaton, given a state and a symbol is logarithmic on d.

Under the standard complexity model for incremental recomputation [5, 12, 9], the complexity per update is the complexity of recomputing both the result and the auxiliary data from the input, the update, and the old result and auxiliary data. We will consider as data the documents (i.e., element labels, element ordering information, etc.) and the DTDs.

3.1. The complexity of validating XML documents: the static case

We now briefly present the complexity for the static version of the problem; i.e., given a DTD D and an XML document X, is it the case that $X \in L(D)$?

Static validation of structural constraints can be done as follows. In the SAX model, elements in the XML document are "visited" in depth-first search order, thus we have to validate $O(h)$ elements simultaneously; i.e., simulate one automaton for each "open" element. Therefore, static validation amounts to advancing states in the automata as we see the closing tags of elements. Using a stack to keep the current states of the automata being simulated, the complexity of static validation using SAX is $O(n \log d)$ time and space. For the DOM model, we can validate each element at a time, at a cost of $O(n(\log n + \log d))$ time and $O(\log n + \log d)$ space. Note that it is possible to build a DOM-like representation of a document while performing a SAX validation, at the added $O(\log n)$ cost for each insertion into the data structures.

For validation of attribute constraints, checking that ID attribute values are unique and that there are no dangling references can be done in $O(n \log n)$ time using linear space. It is not hard to see that these are worst-case optimal space and time bounds.

3.2. Update operations

To talk about incremental validation, we need an update language. Proposing a proper update language for XML is outside the goals of this paper. Instead, we present a minimal set of operations, consisting basically of insertions and deletions of subtrees. We point out that, as was the case with static validation, incremental validation of the #PCDATA, EMPTY and ANY content models is trivial. Thus, we focus on incremental validation of elements whose content models are specified by 1-unambiguous regular expressions. We use the following update primitives:

- **Append**(p,y), where both p and y are elements, results in inserting y as the last child of p;
- **InsertBefore**(x,y), where both x and y are elements, results in inserting y as the immediate left sibling of x; this operation is not defined if x is the root of the document being updated;
- **Delete**(x), where x is an element, results in deleting x from the document. We assume that x is not the root of the document (note that if x is the root then the operation is trivially valid).

The cost of insertion operations has three components: statically validating the subtrees being inserted, incrementally validating the document resulting from the insertion, and updating the document to reflect the insertion. In general, only structural constraints can be validated statically on the subtree alone, since the validity of ID and IDREF attributes is defined with respect to the entire document. As discussed earlier, the cost of statically validating the structural constraints of an XML tree y is $O(|y| \log d)$ and the cost of inserting y into the document is $O(|y| \log n)$.

With respect to deletes, the costs are: incrementally validating the document resulting from the deletion, and updating the document to reflect the operation. The cost of deleting a subtree y from the document is $O(|y| \log n)$. In the remainder, we consider the complexity of incremental validation of the document only. In other words, we assume that all subtrees being inserted are structurally valid.

One important observation that should be made here is that the incremental validation concerns only the content of the element where the update takes place. For example, after an **Append**(p,y) operation only the content of p needs to be revalidated. Therefore, one can expect that even a full revalidation of the content of p after that operation should outperform revalidation from scratch of the entire document.

3.3. Incremental validation of structural constraints: the general case

Assume that E is a 1-unambiguous regular expression that defines the content of some element p, and G is the Glushkov automaton of E. Let $w = w_1 \ldots w_k$ be the string formed by concatenating the labels of the children of p. For clarity of exposition, we will refer to the i-th child of p as the i-th symbol in w (i.e., w_i). Assume that the document is valid (i.e., $w \in L(E)$) before the update. The problem then is to determine whether w', the updated version of w, belongs to $L(E)$.

Our approach is as follows. Together with the i-th child of p, we store the value of $\hat{\delta}(w_1 \ldots w_i)$ for the automaton that validates the content model of p; note that this requires an auxiliary storage of size $O(n \log d)$. Whenever an update operation modifies w, we check whether we can modify the stored values of $\hat{\delta}(w_1 \ldots w)$ accordingly, in a way that it still yields an accepting computation. It is easy to see that $w' \in L(G)$ if and only if this procedure succeeds. We now discuss the complexity of doing so.

3.3.1. Appends and deletions at the end. The cases where the updates occur at the last child of an element are very simple. For example, in an **Append**(p,y) operation, $w' = w_1 \ldots w_k y \in L(E)$ iff $\delta(\hat{\delta}(w_1 \ldots w_k), y) \in F$. Checking such condition can be done as follows. Finding $\hat{\delta}(w_1 \ldots w_k)$ can be done in $O(\log n)$ time, and determining whether $\delta(\hat{\delta}(w_1 \ldots w_k), y) \in F$ requires $O(\log d)$ time; thus, the cost of incremental validation of appends is $O(\log n + \log d)$ time.

The validation after a **Delete**(x) operation when $x = w_k$ requires only checking whether $\hat{\delta}(w_1 \ldots w_{k-1}) \in F$ and can be handled in $O(\log n + \log d)$ time as well.

3.3.2. Arbitrary insertions and deletions. Now consider the incremental validation after a **Delete**(x) operation, where $x = w_i, i < k$. In this case, note we need not recompute $\hat{\delta}(w_1 \ldots w_j), j < i - 1$; it suffices to revalidate $w_{i+1} \ldots w_k$ starting from $\delta(\hat{\delta}(w_1 \ldots w_{i-1}), w_{i+1})$. This can be further improved by checking if at any point $w_l, l > i$, the new $\hat{\delta}$ function has the same value as the one previously computed; since we started with a string in $L(E)$, this would indicate that w' is in $L(E)$ as well.

Dealing with the **InsertBefore**(x,y) operation is similar, as only the part of the string after the insertion needs to be revalidated. Thus, the algorithms for both deletion and insertion require $O(|w|(\log n + \log d))$ time; however, note $|w| = O(n)$.

This bound is tight: consider the marked regular expression $a(b_1 * |cb_2*)$, and assume $w = acb \ldots b$ and that we delete the c from w. Initially, all b's in w match state b_2; however, the deletion requires that all b's now match b_1. A similar argument applies for inserting a c when $w = ab \ldots b$.

3.4. Incremental validation of structural constraints for restricted DTDs

We now introduce two classes of DTDs for which the incremental validation can be done very efficiently. The intuition for introducing our restricted DTDs is as follows. Consider again the example of the regular expression $a(b_1 * |cb_2 *)$ for which the insertion or deletion of a c might require the re-validation of the entire string. This happens because there are two positions in the regular expression (b_1 and b_2) which are "close" and correspond to the same symbol. Thus, inserting or deleting a c element results in "flipping" all b_1's into b_2's or vice-versa. To avoid this problem, we limit the proximity in which positions corresponding to the same symbol may occur in the regular expression.

3.4.1. 1,2-conflict-free DTDs. Let E be a regular expression over an alphabet Σ. Recall that for a position x in E, $\mathit{follow}(E, x)$ is the set of positions in E that can follow x in some path through E. We define $\mathit{follow}_2(E, x)$ as $\{y \in \mathit{pos}(E) \mid \exists z \in \mathit{follow}(E, x)$ such that $y \in \mathit{follow}(E, z)\}$.

Definition 3 *E is a* 1,2-conflict-free *regular expression if (1) E is 1-unambiguous; (2) for every $x, y, z \in \mathit{pos}(E) \cup \{\pi\}$, if $y \in \mathit{follow}(E, x)$ and $z \in \mathit{follow}_2(E, x)$, then $\chi(y) = \chi(z) \;\Leftrightarrow\; y = z$.*

A DTD D is 1,2-conflict free (1,2-CF) if every regular expression in D, defining an element content model, is 1,2-conflict-free.

Next, we show how to do the incremental validation for 1,2-CF DTDs. Let E be a 1-unambiguous regular expression and $w = w_1 \ldots w_k \in L(E)$. We say that w is *i-contractible under E* ($i \in [1, k]$) if it is possible to remove w_i from w without affecting the computation of the Glushkov automaton for E. Formally, w is i-contractible if $\hat{\delta}(w_1 \ldots w_{i-1} w_{i+1}) = \hat{\delta}(w_1 \ldots w_{i-1} w_i w_{i+1})$. We say that w is *i, a-expansible under E*, where a is a symbol in the alphabet of E, if it is possible to insert a in the i-th position of w without affecting the computation of the Glushkov automaton for E. Formally, w is i, a-expansible if $\hat{\delta}(w_1 \ldots w_{i-1} a w_i) = \hat{\delta}(w_0 \ldots w_{i-1} w_i)$.

The following theorem shows that incremental validation of 1,2-conflict-free regular expression can be reduced to the problem of checking either i-contractibility (for deletions) or i, a-extensibility (for insertions).

Theorem 1 *Let E be a 1,2-conflict free regular expression and $w = w_1 \ldots w_k \in L(E)$.*

1. *If $w' = w_1 \ldots w_{i-1} w_{i+1} \ldots w_k$, then $w' \in L(E)$ iff w is i-contractible under E.*
2. *If $w' = w_1 \ldots w_{i-1} a w_i \ldots w_k$, then $w' \in L(E)$ iff w is i, a-expansible under E.*

Testing for i-contractibility for a deterministic Glushkov automaton can be done as follows. Assume that E is a 1-unambiguous regular expression and $w = w_1 \ldots w_k$ is a string in $L(E)$ such that $\hat{\delta}(w_1 \ldots w_{i-1}) = q$. To check whether w is i-contractible, we just need to verify that $\delta(q, w_{i+1}) = \delta(\delta(q, w_i), w_{i+1})$. Note that, given w_i, finding q, and w_{i+1} are both $\log n$ operations; testing the i-contractibility condition amounts to two lookups in the transition function, at a cost of $\log d$ time each. Thus, the total cost is $O(\log n + \log d)$ time. Checking i, a-expansibility can be done in a similar manner.

In summary, we have the following.

Corollary 1 *The per-update complexity of the incremental validation of XML documents with respect to 1,2-conflict free DTDs is $O(\log n + \log d)$ time and $O(n \log d)$ auxiliary space.*

Note that for the restricted case of updates at the leaves (like those studied in [11]), and for *fixed* DTDs, the complexity of our algorithms matches that of [11]. However, the constants involved in our time and space bounds are much smaller than those in [11].

3.4.2. Conflict-free DTDs. We now focus on a further restricted form of 1,2-conflict free DTDs for which the incremental validation can be done without the need for auxiliary space. The condition we impose on the DTDs is simply that the regular expressions have no repeated symbols.

Definition 4 *E is a* conflict-free *regular expression if: (1) E is 1-unambiguous, and (2) for every $y, z \in \mathit{pos}(E)$, $\chi(y) = \chi(z)$ iff $y = z$. A DTD D is* conflict-free *if all regular expressions in D are conflict-free.*

A recent study [4] shows that most "real" DTDs define only EMPTY, ANY and conflict-free regular expressions as content models for elements.

Evidently, CF expressions are also 1,2-CF. Moreover, note that for CF regular expressions we have that $\mathit{pos}(E) = \Sigma$. Consequently, there is a 1-1 mapping between symbols in w and states in the Glushkov automaton of G. Thus, testing for i-contractibility or i, a-expansibility under CF regular expressions can be done by inspecting w_{i-1} and w_{i+1}, which eliminates the necessity of precomputing and storing the values of $\hat{\delta}$ for the incremental validation. Therefore, we have the following:

Corollary 2 *The incremental validation of XML documents with respect to CF DTDs can be done with the same time complexity as in Corollary 1, and with only constant auxiliary space.*

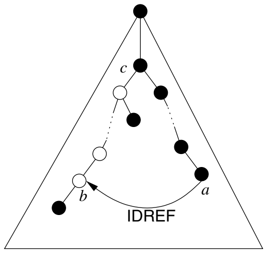


Figure 1. XML document with a single element reference.

3.5. Incremental validation of attribute constraints

We now briefly describe the incremental validation of the attribute constraints, focusing on ID and IDREF attributes, since the validation of all other types is trivial. An update operation is valid if after its execution the following hold:

- No two distinct elements have the same value for their ID attribute;
- No IDREF attribute refers to a non-existing ID value (i.e., there are no "dangling" references in the document).

Checking the validity after **Append**(p,y) or **InsertBefore**(x,y) operations amounts to verifying that no two ID attributes are the same, and every IDREF attribute in y refers to some values (either in y or the rest of the document). Both are easily achieved by maintaining a data structure on the set of ID attribute values, that allows both logarithmic lookup and insertion. In this case the incremental validation of attribute constraints takes $O(|y| \log n)$ time and linear auxiliary space.

3.5.1. Deletions. Checking the validity after a **Delete**(x) operation is more challenging. One has to check whether the subtree rooted at x contains a node that has an ID attribute referenced by some other node that is not a descendant of x. To illustrate the problem, consider the XML tree in Figure 1, in which there is only one IDREF reference from element a to element b. Let c be the closest common ancestor of a and b. It is easy to see that the removal of any element in the path cb, except for c itself (i.e., the elements shown as hollow nodes in the figure), results in dangling references in the document. Put more precisely:

Definition 5 *Let a, b, c be nodes in an XML tree s.t. a has an IDREF attribute that references b, and c is their closest common ancestor. The a, b*-reference path *is the path resulting from removing c from the $c \rightsquigarrow b$ path.*

It is easy to see that a **Delete**(x) operation results in dangling references if and only if x lies in some reference path in the document. This also applies for the cases when c is equal to a or b in Definition 5. Thus, the incremental validation of attribute constraints can be done as follows. Since a node can be in several reference paths, we store a reference counter for each node in the tree (which requires $O(n \log n)$ auxiliary space). Whenever an IDREF is inserted, we increment the reference counters for all nodes in the corresponding reference path; conversely, these counters are decremented whenever an IDREF attribute is removed.

In this way, checking whether **Delete**(x) yields dangling references amounts to checking whether the reference counter of x is equal to 0, and can be done in $O(\log n)$ time. The costs for inserting/removing an IDREF attribute now has to take into account the number of reference counters that need to be incremented/decremented (which is $O(h)$), which yields a cost of $O(h \log n)$ time for each operation. Thus, the final cost for checking IDREF attribute constraints is $O(|y| h \log n)$. The empirical evidence to date shows that the vast majority of XML documents found on the web are shallow [8].

4. Implementation

In this section we present a prototype implementation of our algorithms, and discuss how our methods could be coupled with storage methods for XML. Our implementation was done in C++; we used the Berkeley DB[2] libraries for storage, and the Xerces SAX parser[3] for "shredding" the documents and loading them into our data structures.

We assign a unique *type* identifier to each occurrence of an element label in the DTD (or XML Schema specification) taking the context into account; we use integers to represent element types. Also, we rewrite the regular expressions in the DTD in terms of element types, as opposed to labels. By doing so, we not only capture the context information, but also avoid costly string comparisons when simulating the automata.

4.1. Data structures

Figure 2 shows the data structures we use. The fields shown in bold are keys for the corresponding relations; all such relations are indexed by their keys. The underlined attributes are index fields for the relations that have no keys.

The structure of the XML documents is captured by the following relations. The element relation stores, for each element, the id of its parent, its type, the value of its reference counter (see Section 3.5.1), and the value of $\hat{\delta}$ for

[2] Available at http://www.sleepycat.com.

[3] Available at http://xml.apache.org.

element (B-tree): {**id**, p_id, type, ref_cnt, state}
LS (B-tree): {**id**, id_left}
RS (B-tree): {**id**, id_right}
FLC (B-tree): {**id**, id_first, id_last}
ID (Hash table): {**value**, id}
IDR (Hash table): {**id**, value}
IDREF (Hash table): {id, id_begin, id_end}
transition (B-tree): {p_type, from, to, type, label}

Figure 2. Data structures used in our implementation.

that element assigned during the static validation (see Section 3.3). LS and RS store the ordering of the elements in the document: given an element id, LS (respect., RS) stores the id of the left (respect., right) sibling of that element. The FLC relation gives the ids of the first and last children of a given element.

With respect to ID and IDREF attributes, the ID relation stores the id of the elements given the value of the corresponding ID attribute. The IDR relation holds the inverse relation of ID, and is necessary when removing elements from the document that contain ID attributes. Finally, the IDREF relation stores, for each node that contains an IDREF attribute, the begin and end points of the corresponding reference path; we use the b element as the starting point of a a, b- reference path (recall Figure 1). In this way, navigating the path amounts to finding the start element and recursively finding its ancestors, until we reach the end of the path.

The last relation in Figure 2, transition, stores the transition functions of all automata defined in the DTD. In order to store all transition functions in a single relation, we add the type of the element whose content model is given by the regular expression as part of the key for that relation (the p_type column). As usual, each transition is identified by a pair of states (from and to), and a symbol (type).

The type of an element being inserted is determined by its label and the type of its parent (which specifies the context of the insertion). Thus, in order to perform insertions, we proceed as follows. Let $\mathcal{T}$ denote the set of element types in the DTD; we maintain the mapping $\mu : \mathcal{T} \times \Sigma \to \mathcal{T}$ which gives the type of an element of a given label depending on the type of its parent. We materialize μ by simply adding an extra column (label) to the transition relation; note that we do not add extra tuples by doing so, since the functional dependency {p_type,type} → {label} holds. Also, note we gain in performance, since we avoid extra table lookups for each insert and append operation.

```
<!ELEMENT catalog (book+,review+)>
<!ELEMENT book (title,author+,price)>
<!ATTLIST book isbn ID #REQUIRED
     genres CDATA #IMPLIED>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!ATTLIST price currency CDATA #IMPLIED>
<!ELEMENT review (user,p*)>
<!ATTLIST review isbn IDREF #REQUIRED
     rating CDATA #REQUIRED
     date CDATA #IMPLIED>
<!ELEMENT user (#PCDATA)>
<!ELEMENT p (#PCDATA)>
```

Figure 3. Catalog DTD.

4.2. Integration with other systems

We conclude this section with a brief discussion of how our algorithms could be used by XML data management systems. We consider two scenarios for integration: implementing the algorithms inside the storage engine; or using them as an external service to the storage manager.

Obviously, implementing the algorithms in the storage engine has the potential for better performance, provided that the data structures used by the storage engine allow the access patterns we use. In this regard, we note that, as shown in this section, our algorithms can be implemented using standard data structures. Also, all the information we use from the documents is “essential”, in the sense that it is reasonable to expect any storage mechanism for XML will capture them; therefore, the storage overhead of our algorithms will be essentially storing the values of $\hat{\delta}$. Another good reason for implementing these algorithms inside the storage engine is to avoid having duplicate information about the structure of the document.

The second approach seems more appealing for DTD-aware relational mappings of XML documents (e.g., [1]), in which the relational schema (i.e., the data structures) is not uniform across all DTDs. However, we note that the validity of certain simple content models (e.g., `name(first,last)`) can be enforced directly by the relational schema. Thus, our algorithms could be used to provide incremental validation only for those content models which cannot be captured directly by the relational schema. In this scenario, an update operation would only be processed by the storage engine after it is incrementally validated using our algorithms. In order for this scheme to work, a 1-1 correspondence between element identifiers in both systems is required; however, this should not represent a problem in practice.

Size	Book elements	Elements	Validation time	Database size
64K	50	1.1K	22 ms	144K
512K	400	9K	73 ms	548K
4M	3.2K	73K	386 ms	3.75M
32M	25.6K	583K	2.9 s	29.2M
256M	204K	4.6M	24.2 s	234M
2G	1.6M	37M	479 s	1.9G

Table 1. Catalog datasets used.

5. Experimental analysis

We now present the results of a preliminary experimental analysis of our algorithms. We ran two sets of experiments: in the first, we used synthetic documents describing book catalogs and conforming to the CF DTD in Figure 3;for the second set, we used documents conforming to the XMark benchmark [13]. All tests were run on a Pentium 4 2.4GHz machine with 1G of RAM, 20G of disk and running Linux. All results reported here were obtained using a memory buffer of 4MB (this was the default value set by Berkeley DB).

We use the following conventions for presenting the results. The times for validation from scratch (of the contents of the element being modified) are reported as Full, while the times for the incremental methods are reported as Incr. We compare the times for validation of CF, 1,2-CF and arbitrary regular expressions separately.

All times are reported in microseconds and all graphs are in log-log scale. In general, each workload in our experiments consists of 120 operations of the same type (e.g., valid insertions). We ran the first 20 operations as a "warm-up" procedure, and report the average time of the remaining 100 operations.

5.1. Catalog experiments

For these experiments, we used 6 documents varying in size from 64KB to 2GB. The size of the documents is a function of the number of books in the catalog: on average there are 3 reviews per book; the number of authors per book is uniformly distributed in the interval $[1, 10]$; the number of p elements per review follows a normal distribution with mean 3 and variance 2; and the length of the PCDATA values for each p follows an exponential distribution with mean 100. The isbn values used as ID and IDREF are 10-digit long strings. Table 1 shows, for each document: the number of book elements; the total number of elements; the time for static validation; and the size of all data structures materialized. We performed experiments to measure the revalidation and the update times for the catalog documents.

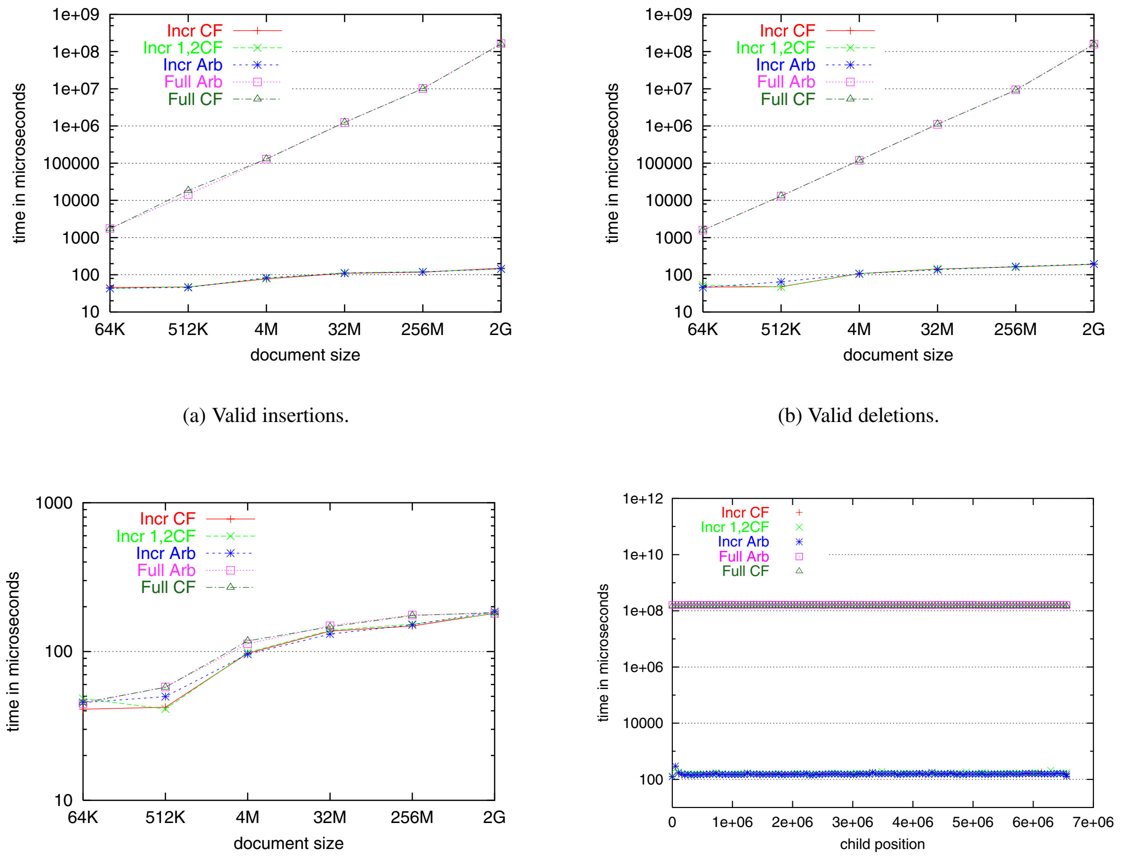
Revalidation times. The goal of these experiments is to show the performance of our approach on a per-operation basis. By using a CF DTD we are able to run all validation methods and, thus, compare the advantages of incremental validation over full revalidation.

Figure 4 summarizes the behavior of our algorithms for incremental validation of structural constraints for the catalog documents. The workloads are as follows. Half of the operations we perform modify the content of the catalog element (i.e., modify a long string), while the other half modify the content of some other node chosen at random (i.e., a short string). Figures 4(a) and 4(b) compare full revalidation to incremental validation for the catalog documents. The results for append operations were identical, and thus are omitted. Two observations can be made here: the full revalidation of (only) the contents being modified (e.g., Full Arb in Figure 4(a)) represents a substantial gain over static validation of the document (see Table 1); and, as expected, incremental validation outperforms that by several orders of magnitude.

Figure 4(c) shows the revalidation times after invalid delete operations. We note that all invalid deletions in the catalog documents are those that modify the content of elements other than catalog. Thus, the graph shows that, for contents with few elements (i.e., short strings), all algorithms perform very similarly. Contrasting Figures 4(c) and 4(b) shows the impact of the length of the strings being updated on the revalidation times. Also, a careful look at Figures 4(a) shows the impact the buffering. Thee times for 64K and 512K (which fit in memory) are almost identical, while the highest slopes of the curves occur between the 512K and the 4M documents (4M does not fit in memory). As expected, increasing the buffer size has the effect of having more documents at the lower "valley" shown for 64K and 512K.

In our final revalidation experiment for catalog, we used the 2G document (whose catalog element has roughly 6M children), and varied the position in which we perform an insertion operation. Figure 4(d) shows the revalidation times (regardless of whether the operations were successful) for this experiment. That figure shows not only the gap between the full revalidation and the incremental methods, but also that the revalidation times do not depend of the position where we update the string.

Updating the documents. The times for updating the documents do not depend of the revalidation method used; thus we report them separately in Figure 5. For this experiment, we consider the insertion and deletion of elements, ID and IDREF attributes, one at a time (i.e., we do not insert subtrees). Note that only books have ID attribute while only reviews have IDREF attributes, and both kinds of elements occur at the same depth in the documents. The figure shows that the times for individual operations scale very well with



(a) Valid insertions.

(b) Valid deletions.

(c) Invalid deletions.

(d) Varying the position of insertion for children of `catalog` in 2GB document.

Figure 4. Revalidation of structural constraints.

the size of the documents, as was the case for the times for incremental validation.

5.2. XMark experiments

For the final set of experiments we used 4 documents for the XMark benchmark, varying in size from 4M to 2G (i.e., no document fits in memory). The XMark DTD is larger than the one for catalog; however, it is also a CF DTD.

We report here the times for performing two "complete" operations: inserting a new item for auction and removing an auction currently open. We chose an item at random, and used it as the subtree (with 23 elements in total) for the insert operations, with a new value for its ID attribute. The insertions were performed in random places inside the `europe` element; the deletions were performed in random places inside the `open_auction` element. We note that each open auction has 8 IDREF attributes on average. Figure 6 shows the times for these operations separately. The graphs show relatively higher costs for deleting subtrees over inserting subtrees. This happens because deleting the subtrees requires deleting several elements that might reside in different pages on disk. Not surprisingly, increasing the buffer size for performing delete operations reduces the number of page misses. As one can see, the behavior of our algorithms is essentially identical to that for the catalog documents; most notably, the algorithms seem to scale very well with document size. We note this the expected behavior, since both DTDs are CF.

6. Related work

There is extensive literature on the subject of relational view/integrity maintenance, including a book [6] and a recent survey [5], both of which provide a good survey of the field. Some of the view maintenance techniques have been

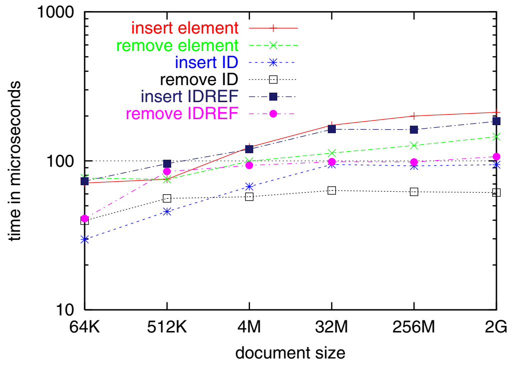


Figure 5. Updating the database.

extended to semi-structured models that were precursors of XML [16].

Static validation of XML documents has been studied in the literature and shown to have low complexity: for DOM-like pointer structures, the validation problem is in LogSpace, while for SAX-like interfaces, the problem is complete for uniform NC^1 [14]. The validation of streaming XML under constrained memory (depending on the DTD only) has also been considered [15]; the paper shows conditions under which validation can be done (in general, streaming validation can be done only for non-recursive DTDs).

Incremental validation of XML documents is closely related to incremental maintenance of regular languages, which was previously studied in the literature [12]. It is known that checking membership in a regular language is complete for the complexity class NC^1, which roughly corresponds to *logarithmic* parallel time [18]. On the other hand, such a membership query can be checked incrementally with AC^0 complexity [12], and AC^0 $\subsetneq$ NC^1 corresponds to *constant* parallel time [18]. 1-unambiguity leads to particularly simple incremental validation techniques.

A recent paper [11] uses algorithms closely related to those in [12] and in our work, and shows that incremental validation of XML documents can be done in logarithmic time with respect to (fixed) DTDs, and in $O(\log^2 n)$ time with respect to Specialized DTDs (an abstraction of XML Schemas [10]). We note that *constant* access time to elements in the database is assumed in [11], as opposed to $O(\log n)$ time in our work; intuitively, doing so allows one to interpret the complexity of the algorithms in terms of number of database accesses, as opposed to time. In the sequel, we adjust our results accordingly, to allow a better comparison between the methods. In [11], the logarithmic time incremental validation for arbitrary DTDs is achieved by using a separate balanced tree for storing the symbols in each string (i.e., the children of each element) in the document. Thus, the the storage costs in that work are $O(nd^2)$ for DTDs and $O(nd^4)$ for Specialized DTDs, where d is the size of the DTD. In contrast, our method yields *constant* incremental validation time for the restricted classes of DTDs and XML Schema specifications, and *linear* worst-case time for arbitrary DTDs. On the other hand, the storage costs in our approach are $O(n \log d)$ in general, and *constant* for CF DTDs.

The update operations we use are similar to those in [11], except that they consider only insertions and deletions of leaf nodes, as opposed to subtrees in our case; on the other hand, they allow the renaming of nodes in the XML documents. In order to support the renaming of arbitrary nodes in the tree (i.e., not only leaf nodes) [11] performs the incremental validation of the contents of *all* elements against *all* regular expressions in the DTD. In our approach, renaming of nodes must be replaced by deletions followed by insertions. For the restricted DTDs we present, the complexity of such operations matches that of the renamings in [11].

Finally, [11] does not consider attribute constraints and does not present any experimental validation.

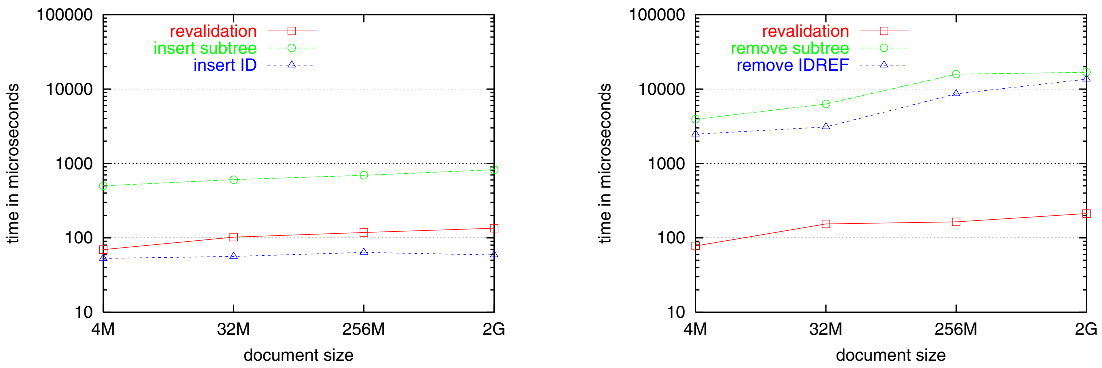
For this study, we had to fix an update language; we settled for a simple language, as the goal was to verify the feasibility of our approach rather than propose a new update language. For a discussion on updating XML, see [17]. We use extensively 1-unambiguous regular expressions, which are discussed in [2, 3]. Further restrictions of DTDs correspond to those identified in an empirical study [4].

7. Conclusion

We have discussed the incremental validation of XML documents with respect to DTDs and XML Schemas , considering insertions and deletions of subtrees, as opposed to leaf nodes only, as well as validation of ID and IDREF attributes. We have characterized a class of DTDs, appearing to capture most real-life DTDs, that admits a logarithmic time and constant space incremental validation algorithm. Membership of a DTD in this class is testable in polynomial time. We have discussed how our algorithms could be used with storage mechanisms for XML, and we have shown through experimental results that the method is practical for large documents and behaves much better than full revalidation.

There are several directions for future work, such as handling complex updates involving several insertions and deletions as a single transaction and studying the cost of incremental validation for various relational mappings.

Acknowledgments. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada, Bell University Laboratories and a PREA grant. D. Barbosa was supported in part by an IBM PhD. Fellowship. We thank Victor Vianu and Yannis Papakonstantinou



(a) Insertion of an item.

(b) Deletion of an open auction.

Figure 6. Insertion and deletion of subtrees in XMark.

for providing an early copy of their paper and for their useful comments.

References

[1] P. Bohannon, J. Freire, P. Roy, and J. Siméon. From XML schema to relations: A cost-based approach to XML storage. In *Proceedings of the 18th International Conference on Data Engineering*, pages 64–75, 2002.

[2] A. Brüggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, 120(2):197–213, 1993.

[3] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 142:182–206, 1998.

[4] B. Choi. What are real DTDs like? In *Proceedings of the 5th International Workshop on the Web and Databases*, pages 43,48, Madison, Wisconsin, United States, 2002.

[5] G. Dong and J. Su. Incremental Maintenance of Recursive Views Using Relational Calculus/SQL. *SIGMOD Record*, 29(1):44–51, 2000.

[6] A. Gupta and I. S. Mumick, editors. *Materialized Views - Techniques, Implementations and Applications*. MIT Press, 1998.

[7] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2nd edition, 2001.

[8] L. Mignet, D. Barbosa, and P. Veltri. The XML Web: A First Study. In *Proceedings of the 12th International Conference on World Wide Web*, 2003.

[9] P. B. Miltersen, S. Subramanian, J. S. Vitter, and R. Tamassia. Complexity Models for Incremental Computation. *Theoretical Computer Science*, 130(1):203–236, 1994.

[10] Y. Papakonstantinou and V. Vianu. DTD Inference for Views of XML Data. In *Proceedings of the 19th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 35–46, 2000.

[11] Y. Papakonstantinou and V. Vianu. Incremental validation of XML documents. In *Proceeedings of The 9th International Conference on Database Theory*, 2003.

[12] S. Patnaik and N. Immerman. Dyn-FO: A Parallel, Dynamic Complexity Class. In *Proceedings of the 13th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 210–221, 1994.

[13] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proceedings of the 28th International Conference on Very Large Data Bases*, 2002.

[14] L. Segoufin. Typing and Querying XML Documents: Some Complexity Bounds. In *Proceedings of the 22nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 167–178, 2003.

[15] L. Segoufin and V. Vianu. Validating streaming XML documents. In *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 53–64, 2002.

[16] D. Suciu. Query Decomposition and View Maintenance for Query Languages for Unstructured Data. In *Proceedings of 22th International Conference on Very Large Data Bases*, pages 227–238, 1996.

[17] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pages 413 – 424, 2001.

[18] H. Vollmer. *Introduction to Circuit Complexity*. Springer Verlag, 1999.

[19] Extensible markup language (XML) 1.0 - second edition. W3C Recommendation, 2000. Available at: `http://www.w3.org/TR/2000/REC-xml-20001006`.

[20] XML Schema part 1: Structures. W3C Recommendation, 2001. Available at: `http://www.w3.org/TR/xmlschema-1/`.

[21] S. Yu. Regular languages. In *Handbook of Formal Languages*, volume 1, pages 41–110. Springer, 1997.