Incremental Validation of XML Documents

ANDREY BALMIN*
IBM Almaden Research Center
and
YANNIS PAPAKONSTANTINOU and VICTOR VIANU
University of California at San Diego

We investigate the incremental validation of XML documents with respect to DTDs, specialized DTDs and XML Schemas, under updates consisting of element tag renamings, insertions and deletions. DTDs are modeled as extended context-free grammars. "Specialized DTDs" allow the decoupling of element types from element tags. XML Schemas are abstracted as specialized DTDs with limitations on the type assignment. For DTDs and XML Schemas, we exhibit an $O(m \log n)$ incremental validation algorithm using an auxiliary structure of size $O(n)$, where n is the size of the document and m the number of updates. The algorithm does not handle the incremental validation of XML Schema wrt renaming of internal nodes, which is handled by the specialized DTDs incremental validation algorithm. For specialized DTDs, we provide an $O(m \log^2 n)$ incremental algorithm, again using an auxiliary structure of size $O(n)$. This is a significant improvement over brute-force re-validation from scratch.

We exhibit a restricted class of DTDs called "local" that arise commonly in practice and for which incremental validation can be done in practically constant time by maintaining only a list of counters. We present implementations of both general incremental validation and local validation on an XML database built on top of a relational database.

Our experimentation includes a study of the applicability of local validation in practice, results on the calibration of parameters of the auxiliary data structure, and results on the performance comparison between the general incremental validation technique, the local validation technique, and brute-force validation from scratch.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Query Processing*; *Relational Databases*; *Transaction Processing*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Information Filtering*

General Terms: Algorithms, Experimentation

Additional Key Words and Phrases: Update, validation, XML

Author's address: Andrey Balmin, IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120. e-mail: abalmin@us.ibm.com. Yannis Papakonstantinou and Victor Vianu, U.C. San Diego, Computer Science and Engineering Department, La Jolla, CA 92093-0114. e-mail: {yannis, vianu}@cs.ucsd.edu

Yannis Papakonstantinou was supported in part by the NSF under grants IRI-9734548 and Digital Government 9983510. Victor Vianu was supported in part by the NSF under grants ITR-0225676 (SEEK) and ITR-0313384.

1. INTRODUCTION

The emergence of XML as a standard representation format for data on the Web has led to a proliferation of databases that store, query, and update XML data. Typically, valid XML documents must conform to a specified type that places structural constraints on the document. When an XML document is updated, it has to be verified that the new document still satisfies its type. Doing this efficiently is a challenging problem that is central to many applications. Brute-force validation from scratch is not practical when the data are large, because it requires reading and validating the entire database following each update. Instead, it is desirable to develop algorithms for incremental validation. However, this approach has been largely unexplored. In this paper we investigate the efficient incremental validation of updates to XML documents.

An XML document can be viewed abstractly as a tree of nested elements. The basic mechanism for specifying the type of XML documents is provided by Document Type Definitions (DTDs) [W3C 1998]. DTDs can be abstracted as extended context-free grammars (CFGs). Unlike usual CFGs, the productions of extended CFGs have regular expressions on their right-hand sides. An XML document satisfies a DTD if its abstraction as a tree is a derivation tree of the extended CFG corresponding to the DTD. XML Schema [W3C 2001] and, more recently, RELAX NG [RELAX NG][1] schemas provide XML typing mechanisms that extend DTDs in several ways. Most notable is the ability to decouple the type of an element from its label. In this paper we use *specialized* DTDs [Papakonstantinou and Vianu 2000], that capture the decoupling of element tags from types. Indeed specialized DTDs allow the type of an element to depend on the full set of node labels (tags) of the XML tree (as is the case in RELAX NG), while XML Schema is abstracted as a restriction of specialized DTDs where the type of an element only depends on its label and the type of its parent. It is a well-known and useful fact that specialized DTDs define precisely the regular languages of unranked trees, and so are equivalent to top-down (and bottom-up) non-deterministic tree automata.

Verifying that a word satisfies a regular expression[2] is the starting point in checking that an XML document satisfies a DTD. An obvious way to do this following an update is to verify it from scratch, i.e. run the updated sequence of labels through the non-deterministic finite automaton (NFA) corresponding to the regular expression. However, this requires $O(n)$ steps, under any reasonable set of unit operations, where n is the length of the sequence (note that, in complexity-theoretic terms, membership of a word in a regular language is complete in NC^1 under DLOGTIME reductions [Vollmer 1999].) We can do better by using incremental validation, relying on an appropriate auxiliary data structure. Indeed, we provide such a structure and corresponding incremental validation algorithm that, given a regular expression r, a string s of length n that satisfies r, and a sequence of m updates (inserts, deletes, label renamings) on s, checks[3] in $O(m \log n)$ whether the

[1]RELAX NG is a schema language for XML, designed within OASIS. It is currently at the final stage of ISO standardization.

[2]A word *satisfies* a regular expression if it belongs to the corresponding language.

[3]For readability, we provide here the complexity with respect to the string and update sequence, for fixed (specialized) DTD or regular expression. The combined complexity is spelled out in the

updated string satisfies r. The auxiliary structure we use materializes in advance relations that describe state transitions resulting from traversing certain substrings in s. These are placed in a balanced tree structure that is maintained similarly to B-trees and is well-behaved under insertions and deletions. The size of the auxiliary structure is $O(n)$. In addition, we provide an $O(m \log n)$ time algorithm that maintains the auxiliary structure, so that subsequent updates can also be incrementally validated.

Our approach to incremental validation of trees with respect to DTDs, specialized DTDs and XML Schemas builds upon the incremental validation algorithm for strings. DTDs turn out to be easier to validate than specialized DTDs, whereas XML Schemas fall between specialized DTDs and DTDs in difficulty. Indeed, based on the algorithm for string validation, incremental validation of m updates to a tree T with respect to a DTD can be done in time $O(m \log |T|)$ using an auxiliary structure of size $O(|T|)$ which can also be maintained in time $O(m \log |T|)$. The same complexity results apply to XML Schemas for all update operations, except internal node renamings, which are handled using the algorithm for the incremental validation of specialized DTDs.

We then consider a restricted class of DTDs called "local", that arises very frequently in practice. Although we did not pursue this, we expect that "local" XML Schemas are equally frequent and their incremental validation can benefit in the same way. Intuitively, these are DTDs using regular expressions for which membership after an update can be determined locally, by examining only substrings within bounded distance from the update position. This allows for a very efficient incremental validation algorithm. Although the theoretical worst-case data complexity of validating m updates is still $O(m \log |T|)$, and the auxiliary structure has size $O(i \log(|T|/i))$, where i is the number of internal nodes of T, the algorithm can be implemented very efficiently. Furthermore, if no internal nodes may be renamed, there is no need for an auxiliary data structure. In practice, the technique provides constant time validation, with space overhead of $O(i)$ counters that are represented very efficiently by usual 32-bit integers, which are sufficient when the maximum size of element lists in the database is 2^{32}. In addition to its simple implementation and efficiency, local validation is very frequent. We tested 60 DTDs from OASIS (see [Choi 2002]) and only 10 DTDs were not local.

We proceed with the incremental validation of specialized DTDs, which is also our method for the incremental validation of XML Schema wrt internal node renamings. Specialization introduces another degree of complexity. Intuitively, this is due to the fact that an update to a single node may have global repercussions for the typing of the tree. This stands in contrast with DTDs without specialization, where a single update to a node needs to be validated only with respect to the type of its parent and the sequence of its children, so has local impact on type checking. In XML Schemas a single update to a node has impact only on the types of its descendants.

We first attempt a rather straightforward extension of the incremental validation for DTDs and obtain an algorithm of time complexity $O(m \ \mathit{depth}(T) \log |T|)$ using an auxiliary structure of size $O(|T|)$. However, this is not satisfactory when the tree is narrow and deep. In the worst case, $\mathit{depth}(T) = |T|$. To overcome this,

paper.

we develop a more subtle approach that has the following main ingredients: First, the unranked tree T representing the XML document is mapped into a binary tree encoding that allows us to unify the horizontal and vertical components of validation. Then the specialized DTD is translated into a bottom-up non-deterministic tree automaton accepting precisely the encodings of valid documents. Finally, an incremental validation algorithm for binary trees with respect to tree automata is developed, based on a divide-and-conquer strategy that focuses on computations along certain paths in the tree chosen to appropriately divide the work. Auxiliary structures are associated to each of these paths. The resulting incremental validation algorithm has time complexity $O(m \log^2 |T|)$ and uses an auxiliary structure of size $O(|T|)$.

Finally, in the last part of the paper, we evaluated the general DTD incremental validation algorithm, the validation algorithm for local DTDs, and a brute-force (re-)validation algorithm on an XML database that operates on top of a commercial RDBMS system. We describe the XML database and the implementation of the necessary data structures on top of an RDBMS. We discuss the applicability of the validation algorithm for local DTDs and a set of performance results that indicate its superiority over general incremental validation in the case of local DTDs. The experiments also quantify the significant superiority of both incremental validation algorithms over the brute-force technique. Finally, the experiments provide useful data for the optimization of various parameters of the data structures.

Related Work. As mentioned earlier, XML databases need to efficiently validate updates on their content. Ipedo's XML database [Ipedo] validates update commands with respect to XML Schemas; however, to our knowledge no technical information is publicly available on the underlying structures and algorithms. Another application where efficient validation is useful is XML editors (see [XML Edt] for a survey of available products). Some XML editors like XMLMind [XMLmind] and XMLSpy [XMLSpy] feature incremental validation of DTDs. Recently, XMLSpy also included validation of XML Schemas [XMLSpy]. No information is provided on their incremental validation algorithms.

Note that our abstraction of the content models of DTDs [W3C 1998] by arbitrary regular expressions removes the requirement for 1-unambiguous regular expressions. The incremental validation algorithm of [Barbosa et al. 2004] utilizes the fact that 1-unambiguousness leads to deterministic Glushkov automata for the regular expressions of DTDs. Consequently [Barbosa et al. 2004] use the Glushkov automata to develop a local incremental validation algorithm. Our definition of "locality" is more general in two aspects. First, the "CF" concept of locality in [Barbosa et al. 2004] corresponds to particular cases of 1-local of our development. We define a more general concept of "k-local", whose significance is that an update can be validated by inspecting only the siblings within distance k from the update. Second, our definition of locality is based on the locality of the minimal automata of the regular expressions, while [Barbosa et al. 2004] base the definition on the Glushkov automata. We prove that if an automaton (including potentially a Glushkov automaton) recognizes a regular expression is local then the minimal automaton is also local, but the converse does not necessarily hold. Hence detecting locality using the minimal automata provides a wider definition. Our locality property is

orthogonal to the "1,2 CF" property of [Barbosa et al. 2004], which was designed exclusively to validate individual atomic updates. Instead, our validation algorithm supports transactions consisting of multiple updates.

Closely related to incremental validation is incremental parsing, which is key to incremental program compilation. Research on incremental parsing has focused on LR parsing [Ghezzi and Mandrioli 1980; Wagner and Graham 1998; Jalili and Gallier 1982; Larcheveque 1995; Petrone 1995] and LL (recursive descent parsing) [Murching et al. 1990; Li 1995; Linden 1993], since programming languages are typically described by LR(0), LR(1), LL(1), LALR(1) and LL(1) grammars. All techniques start by parsing the input text and producing a parse tree, which is typically annotated with auxiliary information. The parse tree is updated as a result of the updates to the input text. A typical theme of the incremental parsing techniques is identifying minimal structural units of the parse tree that are affected by the modifications (see [Ghezzi and Mandrioli 1980] for LR(0) parsing and [Larcheveque 1995] for a generalization to LR(k).) However, the performance of the incremental parsing algorithms is hard to compare to our validation algorithm because of the differences in settings and goals, which typically involve minimization of the changes on the parse tree. Indeed, the best-case performance of incremental parsers will generally beat the one of our regular expression validation algorithm, which always takes $O(\log n)$ steps for a single update. This is because incremental parsers take advantage of natural "termination points" used in programming languages syntax [Linden 1993], that typically occur close to the update. Logarithmic complexity in the size of the string is achieved for LALR grammars by [Wagner and Graham 1998] but only if the grammar is such that its parse trees have depth $O(\log n)$ for a string of length n. One can easily see that there are LALR grammars that do not meet this property, and neither do the CFGs corresponding to DTDs. Furthermore, [Wagner and Graham 1998] require that the interpretation of iterative sequences be independent of the context. In particular, [Wagner and Graham 1998] provide the following "bad grammar", which recognizes the regular expression $(a|b)x^*$

$$S \to aC^+|bD^+$$
$$C \to x$$
$$D \to x$$

This grammar is problematic for their algorithm because the reduction of an x to either a C or a D is determined by the initial symbol in the sentence, which is arbitrarily distant. In this case their algorithm needs $O(n)$ recomputation, where n is the size of the string. Notice that our divide-and-conquer algorithm for the incremental validation of regular expressions does not pose any restriction on the regular expression.

The complexity of validation is related to that of membership of a word in a regular language, and of a tree in a regular tree language. The problem of word membership in a regular language is known to be complete in uniform NC^1 under DLOGTIME reductions [Vollmer 1999] and acceptance of a tree over a ranked alphabet by a tree automaton is complete in uniform NC^1 under DLOGTIME reductions if the tree is presented in prefix notation [Lohrey 2001], and complete in LOGSPACE if the tree is presented as a list of its edges [Segoufin 2002]. To our knowledge,

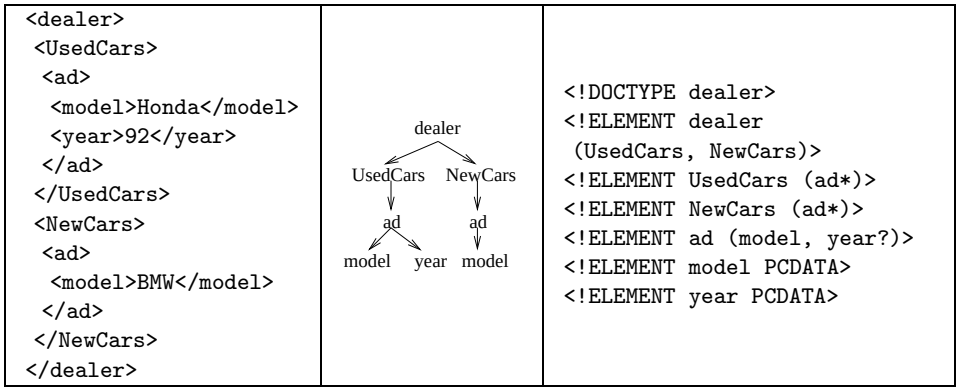
no complexity results exist on the incremental variants of these problems, with the exception of a result of [Patnaik and Immerman 1997] discussed below.

Incremental evaluation of queries by first-order means is studied by [Dong and Su 1995] using the notion of first-order incremental evaluation systems (FOIES) A related descriptive complexity approach to incremental computation is developed by Patnaik and Immerman in [Patnaik and Immerman 1997]. They define the dynamic complexity class Dyn-FO (equivalent to FOIES), consisting of properties that can be incrementally verified by first-order means. They exhibit various problems in Dyn-FO, such as multiplication, graph connectivity, and bipartiteness. Most relevant to our work, they show that membership of a word in a regular language is in Dyn-FO. For label renamings, they sketch an approach similar to ours. The incremental algorithm and auxiliary structure for node insertions and deletions that modify the length of the string are not spelled out. Also, no extension to regular tree languages is discussed. The study in [Patnaik and Immerman 1997] is pursued in [Hesse and Immerman 2002], where an extension of Dyn-FO is introduced and it is shown that the single-step version of the circuit value problem is complete in Dyn-FO under certain reductions. Complexity models of incremental computation are considered in [Miltersen et al. 1994]. The focus is on the classes *incr*-POLYLOGTIME (*incr*-POLYLOGSPACE) of properties that can be incrementally verified in polylogarithmic time (space). Interesting connections to parallel complexity classes are exhibited, as well as complete problems for classical complexity classes under reductions in the above incremental complexity classes.

Organization. The paper is organized as follows. Section 2 presents our abstraction of XML documents and DTDs. It also presents specialized DTDs, their restriction capturing XML Schemas, and their connection to tree automata. We also spell out formally the incremental validation problem and the assumptions made in our complexity analysis. In Section 3 we examine the incremental validation of strings with respect to regular expressions and develop the core divide-and-conquer strategy used later for DTD and XML Schema validation. Section 4 presents an $O(m \log |T|)$ validation algorithm for DTDs and an $O(m)$ algorithm for local DTDs. Those algorithms are also applicable to the incremental validation of XML Schemas wrt insertions, deletions and leaf node renamings. Section 5 presents the algorithm for incremental validation of specialized DTDs, also used for incremental validation of XML Schemas wrt internal node renamings, yielding $O(m \log^2 |T|)$ incremental validation. Section 6 describes our implementation of incremental validation for DTDs. Section 7 presents an evaluation of the applicability of local validation, and experimental results comparing our general incremental algorithm for DTDs, the algorithm for local DTDs, and a brute-force revalidation algorithm. Section 8 contains concluding remarks and future work.

2. BASIC FRAMEWORK

We introduce here the basic formalism used throughout the paper, including our abstractions of XML documents, DTDs, and XML Schemas. We also recall basic definitions relating to tree automata.



$$
\begin{array}{lcl}
root & : & dealer \\
dealer & \rightarrow & UC\ NC \\
UC & \rightarrow & ad^* \\
NC & \rightarrow & ad^* \\
ad & \rightarrow & model\ (year|\epsilon) \\
model & \rightarrow & \epsilon \\
year & \rightarrow & \epsilon
\end{array}
$$

$root : d^t$

$$
\begin{array}{ll}
d^t \rightarrow UC^t\ NC^t & \mu(d^t) = dealer \\
UC^t \rightarrow (ad^u)^* & \mu(UC^t) = UC \\
NC^t \rightarrow (ad^n)^* & \mu(NC^t) = NC \\
ad^u \rightarrow m^t\ y^t & \mu(ad^u) = ad \\
ad^n \rightarrow m^t & \mu(ad^n) = ad \\
m^t \rightarrow \epsilon & \mu(m^t) = model \\
y^t \rightarrow \epsilon & \mu(y^t) = year
\end{array}
$$

Fig. 1. XML, DTD and specialized DTD (UC and NC stand for `UsedCars` and `NewCars`)

Labeled ordered trees. We abstract XML documents as labeled ordered trees. Our abstraction ignores data values present in XML documents, because their validation with respect to an XML Schema is trivial. For example, an XML document holding ads for used cars and new cars is shown in Figure 1 (left), together with its abstraction as a labeled tree.

An *ordered labeled tree* over finite alphabet Σ is a pair $T = \langle t, \lambda \rangle$, where t is an ordered tree and λ is a mapping associating to each node n of t a label $\lambda(n) \in \Sigma$. Trees are assumed by default to be unranked, i.e. there is no fixed bound on the number of children each node may have. The set of all labeled ordered trees over Σ is denoted by $\mathcal{T}_\Sigma$. We sometimes denote a tree consisting of a root v with subtrees $T_1 \ldots T_k$ by $v(T_1 \ldots T_k)$. We will also consider binary trees, where each node has at most two children. If every internal node has *exactly* two children, the binary tree is called *complete*.

We assume that finding (i) the label, (ii) the parent, (iii) the immediate left (right) sibling, and (iv) the first child of a specified node, are unit operations, i.e., they can be accomplished in $O(1)$.

Types and DTDs. As usual, we define XML document types in terms of the document's structure alone, ignoring data values. The basic specification method is (an abstraction of) DTDs. A DTD consists of an extended context-free grammar over alphabet Σ (we make no distinction between terminal and non-terminal symbols). In an extended CFG, the right-hand sides of productions are regular expressions over Σ. An ordered labeled tree $\langle t, \lambda \rangle$ over Σ satisfies a DTD d if the tree $\langle t, \lambda \rangle$ is a derivation tree of the grammar. For example, the tree is valid with respect to the DTD in Figure 1.

The start symbol of a DTD d is denoted by $root(d)$. We can assume without loss of generality that for each $a \in \Sigma$ the DTD has a single rule $a \rightarrow r_a$ with a on the left-hand side. and we denote by N_a a standard non-deterministic finite-state automaton (NFA) recognizing the language r_a. The set of labeled trees satisfying a DTD d is denoted by $sat(d)$.

We use the following notation for NFA. An NFA is a 5-tuple $N = \langle \Sigma, Q, q_0, F, \delta \rangle$ where Σ is a finite alphabet, Q is a finite set of *states*, $q_0 \in Q$ is the *start state*, $F \subseteq Q$ is the set of *final states*, and δ is a mapping from $\Sigma \times Q$ to $\mathcal{P}(Q)$. A string $a_1 \ldots a_n$ is accepted by N iff there exists a mapping $\sigma : \{1, \ldots, n\} \rightarrow Q$ such that $\sigma(a_1) \in \delta(a_1, q_0)$, $\sigma(a_n) \in F$, and for each $i, 1 \leq i < n$, $\sigma(a_{i+1}) \in \delta(a_{i+1}, \sigma(a_i))$. The set of strings accepted by N is denoted $L(N)$. N is a *deterministic finite-state automaton* (DFA) iff δ returns singletons on each input. Recall that for each regular expression r there exists an NFA N whose number of states is linear in r, such that N accepts the regular language r. In general, a DFA accepting r requires exponentially many states wrt r. However, for certain classes of regular expressions, the corresponding DFA remains linear in the expression. One such class consists of the 1-unambiguous regular languages [Bruggemann-Klein and Wood 1998]. This is relevant in the context of XML types, since DTDs and XML Schemas require the regular expressions used to specify the contents of elements to be 1-unambiguous.

An important limitation of DTDs is the inability to separate the *type* of an element from its *name*. For example, consider the dealer document in Figure 1. Used cars have model and year while new cars have model only. There is no mechanism to specify this using DTDs, since rules depend only on the name of elements, and not on its context. To overcome this limitation, XML Schema provides a mechanism to decouple element names from their types and thus allow context-dependent definitions of their structure. We abstract and extend this mechanism using the notion of *specialized DTD* (studied in [Papakonstantinou and Vianu 2000] and equivalent to formalisms proposed in [Beeri and Milo 1999; Cluet et al. 1998]).

DEFINITION 2.1. (**Specialized DTD**) A *specialized DTD* is a 4-tuple $\langle \Sigma, \Sigma^t, d, \mu \rangle$ where Σ is a finite alphabet of labels, Σ^t is a finite alphabet of *types*, d is a DTD over Σ^t and μ is a mapping from Σ^t to Σ. ♢

Intuitively, Σ^t provides, for each $a \in \Sigma$, a set of types associated to a, namely those $a^t \in \Sigma^t$ for which $\mu(a^t) = a$. In our specialized DTD example (lower right corner of Figure 1) we create two types for the element *ad*: a type ad^n whose content is just a "model" type, and a type ad^u whose content is "model" and "year". Note that μ induces a homomorphism on words over Σ^t, and also on trees over Σ^t (yielding trees over Σ). We also denote by μ the induced homomorphisms.

Let $\tau = \langle \Sigma, \Sigma^t, d, \mu \rangle$ be a specialized DTD. A tree t over Σ satisfies τ (or is *valid*

wrt τ) if $t \in \mu(sat(d))$. Thus, t is a homomorphic image under μ of a derivation tree in d. Equivalently, a labeled tree over Σ is valid if it can be "specialized" to a tree that is valid with respect to the DTD over the alphabet of types. The set of all trees over Σ that are valid w.r.t. τ is denoted $sat(\tau)$. When τ is clear from the context, we simply say that a tree is *valid*.

An XML Schema is abstracted as a specialized DTD $\langle \Sigma, \Sigma^t, d, \mu \rangle$ where two additional constraints apply: For every rule $a \rightarrow r_a$ of d, the regular expression r_a does not contain two types a^t and $a^{t'}$ such that $\mu(a^t) = \mu(a^{t'})$. Intuitively, the constraint implies that for every node v of a tree t that satisfies an XML Schema, the type of v is a function of its label and of the type of its parent.

Tree automata. There is a powerful connection between specialized DTDs and *tree automata*: they are precisely equivalent, and define the *regular tree languages* [Bruggemann-Klein et al. 2001]. We will make use of this connection in the paper.

Tree automata are devices whose purpose is to accept or reject an input tree. Classical tree automata are defined on ranked trees, i.e., trees whose internal nodes have a fixed number of children. As in the case of string automata, there are several equivalent variants: top-down nondeterministic automata are equivalent to bottom-up (non)-deterministic ones. In contrast to string automata, top-down deterministic automata are weaker than their non-deterministic counterpart and they capture the key limitation of XML Schemas with respect to specialized DTDs: An XML Schema assigns a unique type to each node and this type can be inferred from the label of the node and the labels of the ancestors of the node. In contrast, specialized DTDs assign to each node a set of possible types, which depends on the context provided by the full set of nodes of the tree. We next review bottom-up non-deterministic automata on complete binary trees. For technical reasons that become clear in Section 5, we assume that all leaves have the same label #.

DEFINITION 2.2. (**Bottom-up non-deterministic tree automaton**)
A bottom-up non-deterministic tree automaton (BNTA) is a 5-tuple $A = \langle \Sigma, Q, Q_0, q_f, \delta \rangle$ where Σ is a finite alphabet, Q is a finite set of *states*, Q_0 is the set[4] of *start states* ($Q_0 \subseteq Q$), q_f is the *accept state* ($q_f \in Q$) and δ is a mapping from $\Sigma \times Q \times Q$ to $\mathcal{P}(Q)$.

A tree $T = \langle t, \lambda, \rangle$ is *accepted* by the automaton A iff there is a mapping σ from the nodes of t to Q such that: (i) if n is a leaf then $\sigma(n) \in Q_0$, (ii) if n is an internal node with children n_1, n_2 then $\sigma(n) \in \delta(\lambda(n), \sigma(n_1), \sigma(n_2))$, and (iii) if n is the root then $\sigma(n) = q_f$. The set of trees accepted by A is denoted by $\mathcal{T}(A)$. ♢

There is a *prima facie* mismatch between DTDs and tree automata: DTDs describe unranked trees, whereas classical automata describe binary trees. There are two ways around this. First, unranked trees can be encoded in a standard way as binary trees. Alternatively, the machinery and results developed for regular tree languages can be extended to the unranked case, as described in [Bruggemann-Klein et al. 2001]. For technical reasons, it will be useful to adopt here the first approach.

[4]Some definitions of BNTA require a single start state for each leaf symbol, and allow a set of final states. Having multiple start states and a single final state is a harmless variation, convenient here for technical reasons.

The incremental validation problem. Given a (specialized) DTD τ, a tree $T \in sat(\tau)$, and a sequence s of updates to T yielding another tree T', we wish to efficiently check if $T' \in sat(\tau)$.[5] In particular, the cost should be less than re-validation of T' from scratch. The individual updates are the following:
(a) replace the current label of a specified node by another label,
(b) insert a new leaf node after a specified node,
(c) insert a new leaf node as the first child of a specified node, and
(d) delete a specified node; if the node is an internal one, the subtree rooted at the node is also deleted.

We allow some cost-free one-time pre-processing to initialize incremental validation, such as computing the NFA corresponding to the regular expressions used by the DTDs. We will also initialize and then maintain an auxiliary structure $\mathcal{A}(T)$ to help in the validation. The cost of the incremental validation algorithm is evaluated with respect to:
(a) the time needed to validate T' using T and $\mathcal{A}(T)$, as a function of $|T|$ and $|s|$
(b) the time needed to compute $\mathcal{A}(T')$ from T, s, and $\mathcal{A}(T)$,
(c) the size of the auxiliary structure $\mathcal{A}(T)$ as a function of $|T|$.

The complexity analysis is provided in terms of the number of update operations and will also make explicit the combined complexity taking into account the specialized DTD.

The algorithms can be trivially extended to accomodate insertions of subtrees. In this case the provided algorithmic complexity results are modified to account for the straightforward non-incremental validation of the subtree.

3. WARMUP: INCREMENTAL VALIDATION OF STRINGS

As warmup to the validation problem, we consider in this section the incremental validation of *strings* with respect to a regular language specified by an NFA. We first consider the case when all updates consist of label renamings. We discuss inserts and deletes later.

Consider an NFA $N = \langle \Sigma, Q, q_0, F, \delta \rangle$, and a string $a_1 \dots a_n \in L(N)$. For compatibility with our tree formalism, we view a string as a sequence of nodes (or elements) each of which has a label. When there is no confusion we sometimes blur the distinction between an element and its label.

Consider a sequence of element renamings $u(a_{i_1}, b_1), \dots, u(a_{i_m}, b_m)$, where $i_1 < i_2 < \dots < i_m$. The renaming $u(a_{i_j}, b_j)$ requires that the label of element a_{i_j} be renamed to b_j. We would like to efficiently check whether the updated string

$$a_1 \dots a_{i_1-1} b_1 a_{i_1+1} \dots a_{i_m-1} b_m a_{i_m+1} \dots a_n \in L(N).$$

Validating the new string from scratch by running it through the automaton N takes $O(n|Q|^2 \log |Q|)$. We can easily do better by maintaining some auxiliary information. For simplicity in the presentation, we assume that we can find the

[5]Notice that a subsequence (prefix) of s may produce a tree $T'' \notin sat(\tau)$, while the complete sequence produces a consistent tree $T' \in sat(\tau)$.

rank of a specified node among its siblings in $O(1)$. This assumption is removed later.

Consider the case of a single renaming $u(i, b)$ for $1 \leq i \leq n$. Suppose that we have pre-computed, for each i, $1 < i < n$, the sets $\mathsf{Pre}(i) = \delta(q_0, a_1 \dots a_{i-1})$ and $\mathsf{Post}(i) = \{s \mid \delta(s, a_{i+1} \dots a_n) \in F\}$. If we precompute Pre and Post in arrays then we can retrieve $\mathsf{Pre}(i)$ or $\mathsf{Post}(i)$ in $O(|Q|)$. An $O(|Q|^2)$ algorithm for checking whether the string is in $L(N)$ following the update $u(i, b)$ is now obvious: If there is a state $s_1 \in \mathsf{Pre}(i)$, a state $s_2 \in \mathsf{Post}(i_1 + 1)$ such that $s_2 \in \delta(b, s_1)$ then the updated string is in $L(N)$.

However, the Pre and Post technique does not scale to m updates. Furthermore, maintaining Pre and Post is problematic because, following each update $u(i, b)$, we need to recompute all $\mathsf{Pre}(j)$ for $j > i$ and $\mathsf{Post}(j)$ for $j < i$. This requires $O(n|Q|^2 \log |Q|)$ time.

As the next step in the warmup, we can try to keep some additional auxiliary information in order to better handle multiple updates. For each i, j, $1 \leq i < j \leq n$, let T_{ij} be the *transition relation* $\{\langle p, q\rangle \mid p, q \in Q, q \in \delta(p, a_i \dots a_j)\}$. Note that $T_{ij} = T_{ik} \circ T_{kj}$, $i < k < j$, where $\circ$ denotes composition of binary relations. We also denote by δ_a the relation $\{\langle p, q\rangle \mid q \in \delta(p, a)\}$ for $a \in \Sigma$. If all T_{ij} are available, then checking validity of the updated string $a_1 \dots a_{i_1-1} b_1 a_{i_1+1} \dots a_{i_m-1} b_m a_{i_m+1} \dots a_n$ reduces to verifying that

$$\langle q_0, f\rangle \ \in \ T_{0(i_1-1)} \circ \delta_{b_1} \circ T_{(i_1+1)(i_2-1)} \circ \dots \circ T_{(i_m+1)(n)}$$

for some $f \in F$. This takes time $O(m|Q|^2 \log |Q|)$, if we assume that we have precomputed in a 2-dimensional array all relations T_{ij}. In particular, the composition of two relations is a join operation. It can be accomplished in $O(|Q|^2 \log |Q|^2) = O(|Q|^2 \log |Q|)$ by employing sort-merge join. Each relation is sorted in $O(|Q|^2 \log |Q|)$ and then they are merged in $O(|Q|^2)$. The same complexity can be derived if we assume binary tree indices on each attribute of the relations and we employ index-based join [Garcia-Molina et al. 2001]. The size of the array required for the precomputation is $n^2|Q|^2$. However, maintaining the precomputed structure is prohibitively expensive, since we have to recompute every relation T_{ij} if there is an update between the ith and jth position of the string. We are therefore led to consider a more promising approach, which provides the basis for the solution we adopt.

Divide-and-conquer validation. We describe a divide-and-conquer approach that allows validating a sequence of m renamings to a string of length n, as well as update the auxiliary structure, in $O(m|Q|^2 \log |Q| \log n)$ time. The size of the auxiliary structure is $O(|Q|^2 n)$. Note that the approach below is similar to that briefly sketched in [Patnaik and Immerman 1997].

For simplicity, assume first that n is a power of 2, say $n = 2^k$. The main idea is to keep as auxiliary information just the T_{ij} for intervals $[i, j]$ obtained by recursively splitting $[1, n]$ into halves, until $i = j$. More precisely, consider the *transition relation tree* $\mathcal{T}_n$ whose nodes are the sets T_{ij}, defined inductively as follows:

- the root is $T_{1,2^k}$
- each node T_{ij} for which $j - i > 0$ has children T_{ik} and $T_{(k+1)j}$ where $k = \frac{j-i+1}{2}$,

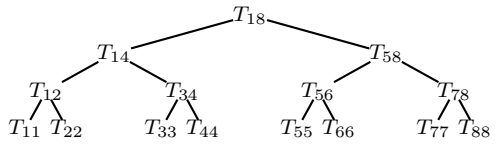


Fig. 2. The tree $\mathcal{T}_{18}$

• T_{ii} are leaves, $1 \leq i \leq n$.

For example, $\mathcal{T}_8$ is shown in Figure 2.

Note that $\mathcal{T}_n$ has $n + (n/2) + \ldots + 2 + 1 = 2n - 1$ nodes and has depth $\log n$. Thus, the size of the auxiliary structure is $O(n|Q|^2)$.

Consider now a string $a_1 a_2 \ldots a_n \in L(N)$, and a sequence of renamings $u(i_1, b_1), u(i_2, b_2), \ldots, u(i_m, b_m)$, where $i_1 < i_2 < \ldots < i_m$. The updated string is $a_1 \ldots a_{i_1-1} b_1 a_{i_1+1} \ldots a_{i_m-1} b_m a_{i_m+1} \ldots a_n$. Note that the relations T_{ij} that are affected by the updates are those laying on the path from a leaf $T_{i_v i_v}$ $(1 \leq v \leq m)$ to the root of $\mathcal{T}_n$. Let $\mathcal{I}$ be the set of such relations, and note that its size is at most $m \log n$.

The tree $\mathcal{T}_n$ can now be updated by recomputing the T_{ij}'s in $\mathcal{I}$ bottom-up as follows: First, the leaves $T_{i_v i_v} \in \mathcal{I}$ are set to δ_{b_v}, $1 \leq v \leq m$. Then each $T_{ij} \in \mathcal{I}$ with children T_{iv} and T_{vj} for which at least one has been recomputed is replaced by $T_{iv} \circ T_{vj}$. Thus, at most $m \log n$ T_{ij}'s have been recomputed, each in time $O(|Q|^2 \log |Q|)$, yielding a total time of $O(m|Q|^2 \log |Q| \log n)$.

The validation of the string $a_1 \ldots a_{i_1-1} b_1 a_{i_1+1} \ldots a_{i_m-1} b_m a_{i_m+1} \ldots a_n$ is now trivial: it is enough to check, in the updated auxiliary structure, that $\langle q_0, f \rangle \in T_{1n}$ for some $f \in F$. Thus, validation is also done in time $O(m|Q|^2 \log |Q| \log n)$.

The above approach can easily be adapted to strings whose length is not a power of 2 (for example, by appropriately truncating $\mathcal{T}_{2^k}$ where $k = \lceil \log n \rceil$).

Dealing with inserts and deletes. We next extend the divide-and-conquer approach outlined for renamings to the case when node inserts and deletes are also allowed. The above approach no longer works, for two reasons: First, inserts and deletes cause the position of nodes in the string to change. Second, the length n of the string, and therefore the set of relevant intervals used to construct $\mathcal{T}_n$, are now dynamic. Due to these differences, inserts and deletes would require recomputing the entire tree $\mathcal{T}_n$, which is inefficient. Instead, we would like to use a tree structure $\mathcal{T}$ that can be incrementally maintained under inserts and deletes, as well as renamings, while preserving the properties that enabled our divide-and-conquer approach. Most importantly, the tree should continue to be balanced and have depth $O(\log n)$. This suggests adopting an approach based on B-trees, that we describe next. We assume basic familiarity with B-trees (e.g., see [Garcia-Molina et al. 2001]).

The B-tree variant we use is the 2-3 tree, which was a precursor to B-trees [Cormen et al. 1990]. Each node contains 3 cells. Each cell is either empty or contains a transition relation T_s corresponding to some subsequence s of $v_1 \ldots v_n$, conforming to the rules described below. At most one of the 3 cells in a node can be empty, assuming $n \geq 2$. Each nonempty cell is either contained in a leaf node or has one node (with three cells) as a child. The following rules apply to the

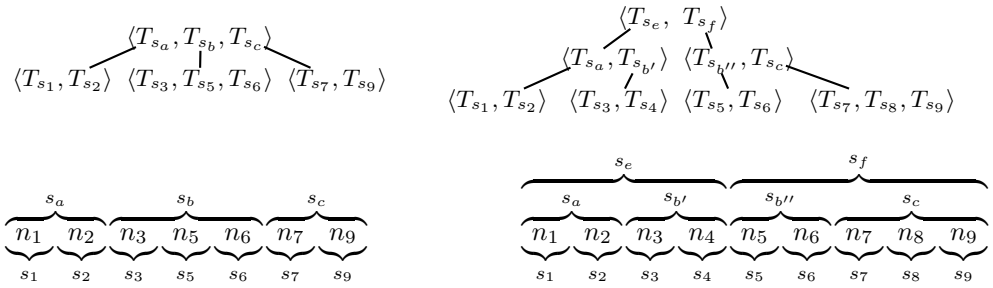


Fig. 3. A $\mathcal{T}$ tree before and after the insertion of nodes n_4 and n_8

transition relations stored in the cells:

—if the root has two nonempty cells containing the relations T_{s_1} and T_{s_2} (resp. three cells containing the relations T_{s_1}, T_{s_2} and T_{s_3}) then $T_{s_1} \circ T_{s_2} = T_{[v_1 \ldots v_n]}$ (resp. $T_{s_1} \circ T_{s_2} \circ T_{s_3} = T_{[v_1 \ldots v_n)]}$);

—if an internal cell contains a relation T_s and its child node contains T_{s_1}, T_{s_2} (resp. T_{s_1}, T_{s_2}, and T_{s_3}) then $T_s = T_{s_1} \circ T_{s_2}$ (resp. $T_s = T_{s_1} \circ T_{s_2} \circ T_{s_3}$);

—the sequence of non-empty leaf cells is $T_{s_1} \ldots T_{s_n}$ and $T_{s_i} = T_{[v_i]}$, $1 \leq i \leq n$.

We also maintain pointers providing in $O(1)$, for each element v in the input string, the leaf cell T_s for which the singleton s consists of v. Note that the position of the element is never recorded explicitly.

For example, the left part of Figure 3 shows a sequence of seven nodes, several subsequences, and the corresponding tree. Note that the subscript of a node does not necessarily indicate its position in the string. Each sequence s_i is the singleton sequence n_i, for $i \in \{1, 2, 3, 5, 6, 7, 9\}$.

The requirement of having 3 cells per node of which at least 2 are non-empty ensures that the tree $\mathcal{T}$ remains balanced and of depth $O(\log n)$ as it is updated. This follows from the standard analysis of B-tree behavior under the maintenance algorithm [Garcia-Molina et al. 2001], which we describe here. In a disk-based implementation one should increase the maximum number of cells per node. Furthermore, the leaf node cells need not correspond to singleton element lists. Indeed, we reduce storage requirements by associating each leaf node cell with a substring. Section 7 provides an evaluation of the performance effect of the number of cells in nodes and of the substring size corresponding to leaf node cells.

Recall that we wish to validate strings with respect to an NFA $N = \langle \Sigma, Q, q_0, F, \delta \rangle$. We describe below the maintenance algorithm for $\mathcal{T}$. Once $\mathcal{T}$ is computed for the current string, validation is easy: check that for some $f \in F$, $\langle q_0, f \rangle$ belongs to the composition of the sets T_s in the cells of the root node of $\mathcal{T}$, at cost $O(|Q|^2 \log |Q|)$.

The auxiliary structure $\mathcal{T}$ corresponding to a valid string w is initialized by starting from the empty string and constructing w by a sequence of inserts, using the maintenance algorithm. Then $\mathcal{T}$ is maintained incrementally as follows. If the update is a renaming of element v, $\mathcal{T}$ is updated much like $\mathcal{T}_n$: we use the index to find the leaf cell of T_v corresponding to v, then update all sets T_s along the path from T_v to the root. This involves $O(\log n)$ updates.

If the update is the insertion or deletion of a new labeled element, the maintenance algorithm mimicks the one for B-trees. In particular, recall that if nodes in

a B-tree become too full as the result of an insertion they are split, and if they contain fewer than two non-empty cells as a result of a deletion they are either merged with a sibling node or non-empty cells are transferred from a sibling node. The node splits and merges may propagate all the way to the root. Due to the similarity to classical B-tree maintenance we omit the details but illustrate how to handle the first variant of insertion; deletion and the second variant of insertion are similar. Assume that an element y with label a is inserted after element x in the current string. If there is some empty cell in the leaf node n of $\mathcal{T}$ containing the set T_x corresponding to x we insert the relation $T_y = \delta_a$ in the cell following that for x and we revise the appropriate T_s relations in ancestor nodes. For example, if a new node n_8 is inserted in the left string of Figure 3 after n_7, we insert T_{s_8} in the node $\langle T_{s_7}, T_{s_9}\rangle$, as shown in the right side of Figure 3, and we revise T_{s_c}, which becomes $T_{s_7} \circ T_{s_8} \circ T_{s_9}$.

If the leaf node n for x has no non-empty cells, then we split n into two nodes n' and n'' containing two relations each. We delete from the parent the relation T_s, where s is the subsequence that corresponds to the node n, and we attempt to insert in the parent relations $T_{s'}$ and $T_{s''}$, which correspond to n' and n''. If the parent already has three relations, the deletion of T_s and the insertion of $T_{s'}$ and $T_{s''}$ will require splitting the parent into two nodes. As is the case for B-trees, this process may propagate all the way to the root and may end up creating a new root. For example, the insertion of a node n_4 following n_3 leads to splitting the node $\langle T_{s_3}, T_{s_5}, T_{s_6}\rangle$ into $\langle T_{s_3}, T_{s_4}\rangle$ and $\langle T_{s_5}, T_{s_6}\rangle$. The relation T_{s_b} is deleted and two new relations $T_{s_{b'}}$ and $T_{s_{b''}}$ are inserted into $\langle T_{s_a}, T_{s_b}, T_{s_c}\rangle$, which leads to a new split and a new root. The result tree is shown in the right side of Figure 3. In the worst case, when an insertion in a leaf node results in splits propagating all the way to the root, we need to recompute $2\log n$ new relations (one at the leaf level, one at the new root, and $2(\log n - 1)$ at the internal nodes). Hence, the worst case complexity is $O(|Q|^2 \log |Q| \log n)$. Deletion proceeds similarly and may lead to node merging or root deletion, with the same complexity. As in the case of B-trees, the maintenance algorithm guarantees that $\mathcal{T}$ always has depth $O(\log n)$ for strings of length n. Altogether, maintenance of $\mathcal{T}$ after m updates takes time $O(m|Q|^2 \log |Q| \log n)$.

1-unambiguous regular expressions. As discussed earlier, XML Schemas require regular expressions used in type definitions to be 1-unambiguous. If r is a 1-unambiguous regular expression, the corresponding DFA is of size linear in r. In this case, the relations T_s used in the above auxiliary structure have size $O(|Q|)$ rather than $O(|Q|^2)$. This brings down the size of the auxiliary structure to $O(|Q|n)$ and the complexity of maintenance and validation to $O(m|Q| \log |Q| \log n)$.

4. INCREMENTAL DTD AND XML SCHEMA VALIDATION

We begin this section by presenting an extension to DTDs and XML Schemas of our incremental validation algorithm for strings. Next, we study in depth a special class of regular languages, called local, that arises frequently in practice and that can be validated very efficiently. DTDs and XML schemas whose rules involve only local regular languages benefit from the efficient validation algorithm we present.

4.1 From strings to DTDs and XML Schemas

The incremental validation of DTDs and XML Schemas extends the divide-and-conquer algorithm for incremental validation of strings described in Section 3. The following discussion excludes XML Schema validation for internal node renamings, which is handled using the techniques we use for specialized DTD validation (see Section 5).

Let d be a DTD, $T = \langle t, \lambda \rangle$ a labeled tree satisfying d, and consider first updates consisting of a sequence of m label modifications yielding a new tree $T' = \langle t', \lambda' \rangle$. To check that T' satisfies d, we must verify that for each node v in t' with children $v_1 \dots v_n$ for which at least one label was modified, the sequence of labels $\lambda'(v_1) \dots \lambda'(v_n)$ belongs to $r_{\lambda'(v)}$. If the label of v has not been modified, i.e. $\lambda(v) = \lambda'(v)$, then validation can be done using the divide-and-conquer algorithm described in Section 3 for strings. However, if the label of v has been modified, so that $\lambda(v) \neq \lambda'(v)$, the sequence $\lambda'(v_1) \dots \lambda'(v_n)$ has to be validated with respect to the new regular language $r_{\lambda'(v)}$ rather than $r_{\lambda(v)}$. Thus, it would seem that, in this case, validation has to start again from scratch. To avoid this, we preemptively maintain information about the validity of each string of siblings with respect to *all* regular languages r_a for $a \in \Sigma$. To this end we maintain some additional auxiliary information. Specifically, for each sequence s of siblings in the tree, we compute the transitions relations T_s of the divide-and-conquer algorithm described in Section 3, for each NFA N_a corresponding to r_a, and $a \in \Sigma$. We denote the sets T_s for a particular $a \in \Sigma$ by T_s^a.[6] Since the auxiliary structure for each fixed NFA and string of length n has size $O(|Q|^2 n)$ (where Q is the set of states of the NFA), the size of the new auxiliary structure is at most $O(|\Sigma||d|^2|T|)$, where $|T|$ is the size of T and $|d| = max\{|r_a| \mid a \to r_a \in d\}$. The maintenance of the auxiliary structure is done in the same way as in the string case, at a cost of $O(m|\Sigma||d|^2 \log |d| \log |T|)$ for a sequence of m modifications. Finally, the updated tree T' is valid wrt d if for each node v with label a in T' such that either v or one of its children has been updated, $\langle q_0, f \rangle$ is in the relation T_s^a where s is the list of children of v, q_0 is the start state of N_a, and f is one of its final states. Each such test takes $O(|d|^2 \log |d|)$ and the number of tests is m in the worst case. This yields a total validation time of $O(m|\Sigma||d|^2 \log |d| \log |T|)$.

For the efficient incremental XML Schema validation of renamings of leaf nodes we maintain for each node of the tree its type; recall that the type of a node can be inferred from its label and the type of its parent. For each list of siblings, whose parent has type a^t, we maintain a transition relation tree for the NFA of the regular expression r_{a^t} that describes the content of nodes with type a^t. However, XML Schema validation for internal node renamings cannot be handled in the same way as DTD validation. The reason is that the renaming of a node v may change the types of all descendants of v. Indeed, it is easy to see that incremental validation of XML Schemas with respect to internal node renamings is at least as hard as validation of strings. We handle this case in the same way we handle validation of specialized DTDs (see Section 5).

Insertions and deletions are handled by a straightforward extension of the B-

[6] Examples 6.1 and 6.2 of Section 6 illustrate the need and the remedy for a particular example.

tree approach outlined in Section 3, for both the DTD and XML Schema cases. In the case of XML Schemas we also compute and store the type of the inserted node. Insertion of m subtrees can be implemented by a sequence of insertions of the individual nodes of the subtrees. The data complexity of this implementation is $O(M \log |T|)$, where M is the total number of nodes of the inserted subtrees. A more efficient implementation first inserts the roots of the subtrees in $O(m \log |T|)$. Then it validates from scratch each subtree. In the case of DTDs the subtree rooted at node v must conform to a DTD that has the same rules but its root is $\lambda(v)$. In the case of XML Schemas the subtree rooted at node v must conform to an XML Schema that has the same types, rules and mapping from types to symbols but its root type is the type of v, which can be inferred from the parent of v and the label of v. The complexity of this implementation is $O(M + m \log |T|)$.

4.2 Local DTDs

In the remainder of this section we focus on a restricted class of DTDs that arises commonly in practice, and for which incremental validation can be done very efficiently. Specifically, these are DTDs using regular expressions for which validity of a string can be decided after an update by examining only substrings of bounded length around the position of the update. This is ensured by a property of the regular expressions called *locality*, which we define shortly. Locality turns out to be a very appealing property, since it immediately yields an incremental validation algorithm that, for all practical purposes, has constant data complexity. In addition, locality is a very common property in practice. We analyzed a set of 60 DTDs collected from OASIS and described in [Choi 2002]. The DTDs contained 2141 complex regular expressions. Only 21 regular expressions in 10 DTDs were not local. We further analyzed these 21 expressions, by examining their content and contacting the authors. We determined that only 8 non-local regular expressions in 3 DTDs describe potentially large sequences of elements. The above experimental results are influenced by the fact that OASIS DTDs primarily describe message exchange information formats and relatively small files. As XML databases mature we expect that large sequences in XML documents will become more common and more non-local regular expressions will be in need of efficient incremental validation. Nevertheless, the results are indicative of how common locality is in practice.

4.3 Local regular languages

Before defining local regular languages, we illustrate the intuition with the following example.

EXAMPLE 4.1. Consider the language defined by the following regular expression, taken from the WellLogML DTD [Well] $CurveData = (data|(piValue, data))+$. Its minimal DFA is shown in Figure 4 (a). It has four states: state 1 is a starting state, state 2 is the only accept state (indicated by shading), and state 0 is a reject sink, i.e. a state from which no accepting state is reachable (note that a minimum DFA has at most one such state). Observe that the DFA has the following special property: when run on a valid string, the current state is uniquely determined by the most recent symbol. Indeed, all transitions from states other than the reject sink lead to state 2 for symbol "data", and to state 3 for symbol "piValue". Thus,

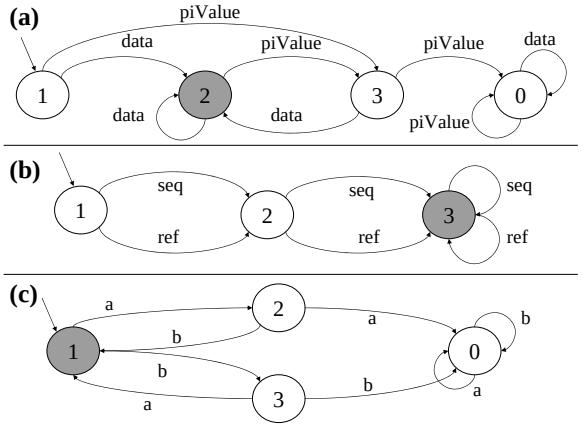


Fig. 4. Example DFAs: (a) and (b) define local languages; (c) does not.

when running on a valid string, the DFA is in state 2 if the last processed symbol was "data", and in state 3 if the last symbol was "piValue". ♢

In the minimum DFA for the regular expression in Example 4.1, the state is determined by the most recently read symbol (unless it is the sink state). As might be expected, some regular expressions require more than one symbol to determine the current state, as illustrated next.

EXAMPLE 4.2. The following regular expression is taken from DDML.DTD, available at
http://www.w3.org/TR/NOTE-ddml: $choice = (seq|ref)(seq|ref)+$. This expression specifies that a "choice" element should have two or more children. This expression defines a local language. The minimal DFA is shown in Figure 4 (b). From any given state, after a single symbol, the DFA may be in state 2 or 3. However, any two letter word always brings the DFA to state 3. ♢

We will call *k-local* a regular expression for which the current non-sink state is determined by the previous k symbols. Before providing the formal definition, we introduce the following notation. Given a DFA $M = \langle \Sigma, Q, q_0, F, \delta \rangle$, we denote by δ^* the natural extension of the transition function δ to $Q \times \Sigma^*$, defined by $\delta^*(q, \epsilon) = q$ and $\delta^*(q, sa) = \delta(\delta^*(q, s), a)$. The set of *potentially accepting* states of M, denoted Q^A, consists of the states in Q other than the reject sink state. We denote the reject sink state, if any, by q^{rej}.

DEFINITION 4.1. (**Local regular languages and DTDs**) A DFA is *k-local* if for every string s of length at least k, there are no distinct $p, q \in Q^A$, such that for some $p', q' \in Q$, $\delta^*(p', s) = p$, and $\delta^*(q', s) = q$. A regular language is *k-local* iff the minimum DFA accepting it is k-local. A regular language is *local* iff it is k-local for some k. The minimum k for which a regular language is k-local is called its *degree of locality*. A regular expression is (k-)local iff the language it defines is (k-)local. A DTD is (k-)local iff all regular expressions it uses are (k-)local. ♢

Let r be a k-local regular language. Note that, given a string w of length $\geq k$ and a potentially accepting state p, either $\delta^*(p, w) = q^{rej}$ or $\delta^*(p, w) = q$ where

q is a potentially accepting state independent of p. We call a string w *rejecting* iff $\delta^*(p, w) = q^{rej}$ for every $p \in Q$. For every string w of length $\geq k$ that is not rejecting, we denote by $\delta^*(-, w)$ the unique potentially accepting state q such that $\delta^*(p, w) = q$ for some potentially accepting p.

Note that the locality of a language is defined as a property of the minimum DFA accepting the language. The following useful fact shows that locality of *any* DFA for the language is sufficient.[7]

PROPOSITION 4.1. *If a regular language is accepted by some k-local DFA, then it is k-local.*

PROOF. Suppose a regular language r is accepted by some k-local DFA D, and let M be the minimum DFA accepting r. We show that M is also k-local. Recall that the DFA minimization algorithm produces M from D by merging equivalent states. Specifically, states p and q are equivalent if for all strings $s \in \Sigma^*$, $\delta^*_D(p, s)$ is accepting iff $\delta^*_D(q, s)$ is accepting, where δ_D is the transition function of D. The algorithm builds equivalence classes of states and replaces each class with a single state of the minimum automaton M. Thus, the minimization algorithm constructs a total mapping μ from the states of D onto the states of M that preserves transitions.

Consider a string s such that $|s| > k$. Let p, q, r, t be states in Q^A_M such that $\delta^*_M(p, s) = r$ and $\delta^*_M(q, s) = t$. There exist states p', q', r', t' in Q^A_D such that $\mu(p') = p, \mu(q') = q, \mu(r') = r, \mu(t') = t$. Furthermore, $\delta_D(p', s) = r'$ and $\delta_D(q', s) = t'$. Since D is k-local, it follows that $r' = t'$, so $r = \mu(r') = \mu(t') = t$. Thus, M is k-local. □

Notice that Proposition 4.1 does *not* imply that all DFAs accepting a k-local language must be k-local, or even local. For example, the language a^* is 0-local, and is accepted by a DFA with two accepting states and two a transitions from each state to the other. Clearly, this DFA is not local.

It is critical to our approach to determine, given a regular language, whether it is local, and if so to compute its degree of locality. We will show that both questions can be answered in time $O(|M|^4)$ where M is the minimum DFA for the language and the size of M, denoted by $|M|$, is $|Q| + |\Sigma| + |\delta|$. Furthermore, if M is local, then the degree of locality is bounded by $|Q|^2$ where Q is the set of states of M.

THEOREM 4.1. *Given a regular language r and its minimum DFA $M = \langle \Sigma, Q, q_0, F, \delta \rangle$, it can be decided in $O(|M|^4)$ time whether r is local. Furthermore, if r is local, its degree of locality is at most $|Q|^2$ and can be computed in $O(|M|^4)$.* ♢

PROOF. Consider the directed labeled graph G whose vertices are pairs $\langle p, q \rangle$ where p, q are in Q^A, $p \neq q$, and there is an edge labeled $a \in \Sigma$ from $\langle p, q \rangle$ to $\langle p', q' \rangle$ iff $\delta(p, a) = p'$ and $\delta(q, a) = q'$. Note that the size of G is at most $|M|^2$. From the definition of locality, it follows that M is local iff G does not contain infinite paths. In other words, M is local iff G is acyclic.

Acyclicity of G can be tested in $O(|G|^2)$ by attempting to perform a topological sort of the vertices of the graph: first compute the in-degree of each node. Next,

[7] In the cases where a language/DTD is non-local it may be worthwhile to consider modifying the DTD to a non-equivalent local one.

start with the nodes with in-degree zero, then remove their outgoing edges and decrease the in-degree count of the target nodes, and repeat. G is acyclic iff eventually all nodes have in-degree zero.

Now suppose G is acyclic, so M is local. From the definition it immediately follows that the degree of locality of M is the longest path in G, which is bounded by $|Q|^2$. This can be computed while the previous topological sort algorithm is performed, by associating a counter with each node n that stores the length of the maximum path from some initial node of in-degree zero to n at the point when n is reached. When the algorithm ends, the maximum value of the counters provides the length of the maximum path in G. Since G itself is quadratic in M, this yields an algorithm of $O(|M|^4)$.

Note that the analysis yielding $O(|G|^2)$ complexity of the above algorithm is extremely conservative, since it assumes that the complexity of finding a node v, that is the target of an edge, and its associated counter is $O(|G|)$. Assuming a data structure that allows locating a node in $O(1)$, which is a reasonable assumption in practice, the complexity is linear with respect to G. □

4.4 Incremental validation of local DTDs

We next show how the locality property of regular languages can be used to obtain a very efficient incremental validation algorithm, of constant time data complexity. Given a DTD using only local regular expressions, if updates only affect leaf nodes without changing the labels of internal nodes, we can use a constant-time incremental validation algorithm without any additional data structure. However, if updates may also cause a renaming of a parent node the sequence of its children needs to be validated against the regular expression for the new label. As in the general incremental validation algorithm for DTDs, we therefore need to maintain some auxiliary information allowing to validate each string of siblings with respect to any of the regular expressions of the DTD, whenever the need arises. In the case of DTDs using only local regular expressions, we attach to each internal node a counter for each regular expression. Each counter's size should be at least $\lceil \log_2 n \rceil$, where n is the maximum size of a sibling list. Notice that 4-byte integers can accommodate sibling lists of up to 2^{32} elements and, hence, in our implementation we just use 4-byte integers for counters.

Our algorithm relies on the following observation.

LEMMA 4.1. *Let r be a k-local regular expression. A string $s \in \Sigma^*$ of length $\geq k$ is valid with respect to r iff the following conditions hold:*

(1) there is no rejecting substring w of s of length $k+1$.
(2) $\delta^*(q_0, w) \neq q^{rej}$, where w is the prefix of s of length k;
(3) $\delta^*(-, w) \in F$, where w is the suffix of s of length k.

♢

PROOF. Clearly, conditions (1-3) are necessary for validity. We show they are also sufficient. Suppose $s = a_1 \dots a_n$ is a string of length $\geq k$ that satisfies conditions (1)-(3). We show that

$$(\dagger) \quad \delta^*(q_0, a_1 \dots a_i) \neq q^{rej} \text{ for all } i, k \leq i \leq n.$$

Clearly, (†) proves the statement, since in conjunction with condition (3) and k-locality it implies that $\delta^*(q_0, a_1 \dots a_n) = \delta^*(-, a_{n-k+1} \dots a_n) \in F$.

We prove (†) by induction. The basis (for $i = k$) is true by condition (2). For the induction step, suppose that $i \geq k$ and $\delta^*(q_0, a_1 \dots a_i) \neq q^{rej}$. Consider $a_1 \dots a_{i+1}$. By the induction hypothesis and k-locality, $\delta^*(q_0, a_1 \dots a_{i+1}) = \delta(\delta^*(-, a_{i-k+1} \dots a_i), a_{i+1})$. By condition (1), $a_{i-k+1} \dots a_{i+1}$ is not rejecting (since it has length $k+1$). Therefore, there exist potentially accepting states p, p' such that $\delta^*(p, a_{i-k+1} \dots a_{i+1}) = p'$. By k-locality,

$$\delta^*(p, a_{i-k+1} \dots a_{i+1}) = \delta(\delta^*(-, a_{i-k+1} \dots a_i), a_{i+1}).$$

It follows that

$$\delta^*(q_0, a_1 \dots a_{i+1}) = \delta(\delta^*(-, a_{i-k+1} \dots a_i), a_{i+1}) = p' \neq q^{rej}.$$

This completes the induction and the proof of (†). □

Lemma 4.1 suggests the following validation algorithm for the sequence of siblings in an XML document. For each regular expression r of the DTD, determine its degree k of locality. As long as the string remains of length at most k, we validate the string with respect to r from scratch if needed. When the string exceeds length k (if ever), we check conditions (1)-(3) of the lemma. Conditions (2) and (3) are easy to check from scratch in constant time whenever required. Condition (1) is checked incrementally by maintaining some auxiliary information consisting in the count of the number of rejecting substrings of length $k+1$ in the current string. To do so, we first initialize the count in a pre-processing step that takes linear time in the size of the string (we do so as soon as the length of the string exceeds k). A single update occuring at position i affects the substrings of length $k+1$ containing i, whose number is constant. For each such affected substring we determine whether or not it becomes (or ceases to be) rejecting, and update the count accordingly. Whenever validation with respect to r is needed, we check that the count of rejecting substrings is zero. For multiple updates, we maintain the counters one update at a time. Thus, the required auxiliary structure consists, for each internal node in the XML tree, of one counter for each regular expression in the DTD. For a tree T with i internal nodes, the size of the auxiliary structure is $O(i \log(|T|/i))$, neglecting the fact that in practice lists will be smaller than 2^{32}, which can be accomodated by a typical 4-byte counter. Maintaining incrementally the auxiliary structure takes $O(m \log |T|)$ time with respect to the string for m updates. Although its theoretical worst-case complexity is no better than the general DTD validation algorithm, the incremental validation algorithm has a very efficient implementation, which practically provides constant time validation. First, the auxiliary structure consists of counters. Assuming that no list of siblings is longer than 2^{32} elements, then each counter can be a usual 32-bit integer. The counters are stored together with the data tree (internal nodes), hence simplifying their access, and their maintenance consists of simply incrementing or decrementing them, which in practice takes constant time.

Notice that, if updates do not involve the renaming of internal nodes, then we do not need to maintain counters. Instead, the number of rejecting substrings is zero before the transaction starts and we check that it remains zero after the transaction has ended. This requires that the updates generate no rejecting substring, the prefix

of the sequence was not rejecting when starting at q_0, and the suffix of the sequence leads to an accepting state.

Furthermore, we do not need to maintain counters if every pair (r, r') of regular languages used by the DTD is *incompatible*, in the sense that it is impossible to turn a string that belongs to $L(r)$ into a string that belongs to $L(r')$, without a transaction performing a number of updates that is in the order of the size of the resulting string - hence, making revalidation of r' from scratch equally attractive to incremental validation. Indeed, whenever two regular expressions r and r' do not contain any common symbol a within the scope of a * they are incompatible. Notice that the reverse is not always true.

Even if we cannot fully eliminate counters, we may still be able to reduce the number of counters we have to maintain for each sibling list. Consider a sibling list whose parent is labeled a and the list belongs to $L(r_a)$. Consider also a label b and the corresponding regular expression r_b, which is incompatible with r_a. We do not need to maintain a counter for r_b since a renaming of a to b needs to be accompanied by a large number of updates on the sibling list and it is not beneficial to incrementally validate such a number of updates.

5. INCREMENTAL VALIDATION OF SPECIALIZED DTDS

In this section we discuss the incremental validation of specialized DTDs. We begin with a simple attempt yielding a validation algorithm of time complexity $O(|T| \log |T|)$. We then explore a more sophisticated approach based on encoding the XML document as a binary tree and validating it using a bottom-up tree automaton. This improves the time complexity to $O(\log^2 |T|)$.

5.1 A first attempt

Specialized DTDs add another degree of complexity to the update validation problem. Intuitively, they abstract the ability of RELAX NG and XML Schemas to associate different types to each element label. Consider a specialized DTD $\tau = \langle \Sigma, \Sigma^t, d, \mu \rangle$. Recall that a tree T over Σ satisfies τ iff there exists some tree T' over Σ^t, satisfying d, such that $\mu(T') = T$. Essentially, T' associates a type in Σ^t to each node in T so that the DTD d over Σ^t is satisfied. The existence of such a type assignment, and therefore the validity of T, can be tested in a bottom-up manner as follows. For each leaf v of T, let $types(v) = \{\alpha \mid \mu(\alpha) = \lambda(v) \text{ and } \epsilon \in r_\alpha\}$. Thus, $types(v)$ consists of all types in Σ^t that may be assigned to the label of v and allow it to be a leaf.

Then apply the following procedure recursively: for each internal node v of T with children $v_1 \dots v_n$ for which $types(v_i)$ has already been computed, let $types(v)$ consist of the types $\alpha \in \mu^{-1}(\lambda(v))$ for which $types(v_1) \dots types(v_n) \cap r_\alpha \neq \emptyset$, where $\alpha \to r_\alpha \in d$. In other words, $types(v)$ consists of all types allowed for the label of v for which there is at least one choice of allowed types for its children that is compatible with d. Clearly, $T \in sat(\tau)$ iff $types(root(T)) \neq \emptyset$. This procedure closely corresponds to the evaluation on T of a bottom-up unranked tree automaton corresponding to τ.

Consider now a tree $T \in sat(\tau)$. We first consider label modifications. We maintain the following auxiliary structure:

—for each node v in T, maintain the set of allowed types $types(v)$. This has size $O(|T||\Sigma^t|)$;

—for each sequence of siblings $v_1 \dots v_n$ in T and $\alpha \in \Sigma^t$, maintain the sets

$$T_s^\alpha = \{\langle p, q\rangle \mid q \in \delta_\alpha(p, \beta_i \dots \beta_j),$$

$$\beta_k \in types(v_k), i \le k \le j\}$$

where s is a subsequence $v_i \dots v_j$ used in $\mathcal{T}$, formulated by the usual divide-and-conquer strategy. This has size $O(|\Sigma^t||d|^2|T|)$.

We describe how to maintain the auxiliary structure when a single label is modified. For m modifications, we apply this for each modification. Validity is checked after the auxiliary structure has been updated for all modifications.

Suppose the node whose label is modified is v, the old label is a, and the new label is b. We need to update the sets $types(w)$ for all nodes w on the path from root to v in T, as well as the sets T_s^α for the sequences s of siblings where such nodes occur. This is done in a bottom-up fashion as follows. First, if v is a leaf, then $types(v) = \{\beta \mid \mu(\beta) = b, \epsilon \in r_\beta\}$. If v has children $v_1 \dots v_n$ then $types(v)$ contains all types $\beta \in \mu^{-1}(b)$ such that $\langle q_0, f\rangle \in T_s^\beta$ where q_0 is the start state of N_β, f is one of its accepting states, and T_s^β is the root of the auxiliary structure corresponding to β and the children of v. Note that this step takes $O(|\Sigma^t||d|^2 \log |d|)$. Next, suppose that w is a node in T whose sequence of children $w_1 \dots w_n$ contains one node w_k for which $types(w_k)$ has been updated. First, the sets T_s^α need to be updated for the $\log n$ affected subsequences s as in the divide-and-conquer string validation algorithm. This takes time $O(|\Sigma^t||d|^2 \log |d| \log n)$. Next, $types(w)$ is updated as in the base case to contain the types $\beta \in \mu^{-1}(\lambda(w))$ for which $\langle q_0, f\rangle \in T_s^\beta$ where q_0 is the start state of N_β, f is one of its accepting states and s is the sequence $w_1 \dots w_n$. This takes time $O(|\Sigma^t||d|^2 \log |d|)$. Thus, the maintenance time for this step is $O(|\Sigma^t||d|^2 \log |d| \log n)$, and this has to be repeated at most $depth(T)$ times. This yields a total maintenance time of $O(|\Sigma^t||d|^2 \log |d| depth(T) \log |T|)$for a single label modification. For m modifications, the maintenance time is $O(m|\Sigma^t||d|^2 \log |d| depth(T) \log |T|)$. Finally the updated tree is valid iff $root(d) \in types(root(T))$. Hence, the total validation time is also $O(m|\Sigma^t||d|^2 \log |d| depth(T) \log |T|)$.

Node insertions and deletions can be handled by adapting the B-tree approach used for strings. The resulting complexity is the same as for label renamings.

Note that for fixed specialized DTD and update sequence, the validation algorithm outlined above takes time $O(depth(T) \log |T|)$. Thus, the algorithm works well for shallow trees. However, in the worst case $depth(T)$ could equal $|T|$, in which case the complexity is $O(|T| \log |T|)$. This is not satisfactory. We will see in the next section how to use a more subtle strategy that reduces the overall maintenance and validation cost to $O(\log^2 |T|)$.

5.2 Incremental Validation via Binary Trees Encodings

In this section we develop a refinement of the incremental validation technique for specialized DTDs described in the previous section. This results in maintenance and validation algorithms of complexity $O(\log^2 |T|)$ for fixed DTD and update sequence,

instead of the previous $O(|T| \log |T|)$. Intuitively, the algorithm of Section 4 is based on a divide-and-conquer strategy to split the work of validating sequences of siblings in the tree. However, for trees of small width and large depth, this strategy is defeated. The refinement presented in this section extends the divide-and-conquer strategy to validation of the overall tree, by splitting the work simultaneously with respect to the horizontal and vertical components. To this end, it is useful to adopt a representation of unranked trees as complete binary trees and reduce the problem of validating specialized DTDs on unranked trees to that of acceptance of the binary tree encodings by a corresponding bottom-up tree automaton. The advantage of this approach is that it unifies the horizontal and vertical components of validation and facilitates a natural formulation of the new divide-and-conquer strategy.

Binary tree encoding of unranked trees. We next describe the encoding of unranked trees as binary trees. We use one of the standard encodings in the literature (e.g, see [Neven 2002]). To each unranked labeled ordered tree $T = \langle t, \lambda \rangle$ over alphabet Σ we associate a binary tree $enc(T)$ over alphabet $\Sigma_\# = \Sigma \cup \{\#\}$, where $\# \notin \Sigma$. The input of enc is a (possibly empty) sequence of unranked trees over Σ, and the output is a complete binary tree over $\Sigma_\#$. The mapping enc is defined recursively as follows (where $\bar{T}_0$ and $\bar{T}$ are sequences of trees, possibly ϵ, and n_0 is a single node):

- $enc(\epsilon) = \#$
- $enc(n_0(\bar{T}_0)\ \bar{T}) = n_0(enc(\bar{T}_0), enc(\bar{T}))$

For example, a tree T and its encoding $enc(T)$ are shown in Figure 5 (neglect for now the boxes and bold letters).

We would like to reduce the validation of unranked trees T wrt a specialized DTD τ to the question of whether $enc(T)$ is accepted by a bottom-up non-deterministic tree automaton. To this end, we show the following result (a variant of known results on equivalences of specialized DTDs and unranked tree automata, and of unranked tree automata and automata on binary trees, see [Neven 2002]):

LEMMA 5.1. *For each specialized DTD $\tau = \langle \Sigma, \Sigma^t, d, \mu \rangle$ there exists a BNTA A_τ over $\Sigma_\#$ whose number of states is $O(|\Sigma^t||d|)$, such that $\mathcal{T}(A_\tau) = \{enc(T) \mid T \in sat(\tau)\}$.* ♢

PROOF. For each $\alpha \in \Sigma^t$, let $N_\alpha = \langle \Sigma^t, Q_\alpha, q_0^\alpha, F_\alpha, \delta_\alpha \rangle$ be a standard NFA that accepts the language $r'_\alpha = \{w^r \mid w \in r_\alpha\}$ where w^r is the reverse of w. Distinct N_α have disjoint sets of states. Let $\mathcal{Q}_d = \bigcup_{\alpha \in \Sigma^t} Q_\alpha$. Let A_τ be the BNTA $\langle \Sigma_\#, Q, Q_0, q_f, \delta \rangle$ where $Q = \{q_f\} \cup \mathcal{Q}_d$, $Q_0 = \{q_0^\alpha | \alpha \in \Sigma^t\}$, q_f is the accept state ($q_f \notin \mathcal{Q}_d$), and δ is defined as follows (δ is empty whenever not specified):

—If $a \in \Sigma, \alpha \in \Sigma^t, \alpha \neq root(d), \mu(\alpha) = a, \beta \in \Sigma^t, q^\beta \in Q_\beta$ and $q_f^\alpha \in F_\alpha$ then

$$\delta(a, q_f^\alpha, q^\beta) = \delta_\beta(\alpha, q^\beta)$$

—If $\rho = root(d), r = \mu(\rho), q_f^\rho \in F_\rho, \beta \in \Sigma^t$ and $q^\beta \in Q_\beta$ then

$$\delta(r, q_f^\rho, q^\beta) = \delta_\beta(\rho, q^\beta) \cup \{q_f\}$$

It is easily seen that $\mathcal{T}(A_\tau) = \{enc(T) \mid T \in sat(\tau)\}$. □

Our approach is based on reducing the validation of unranked trees with respect to specialized DTDs to the validation of their binary encodings with respect to the corresponding BNTA, say $A = \langle \Sigma, Q, Q_0, q_f, \delta \rangle$. As before, the problem really amounts to efficiently updating the auxiliary structure associated with the input. In our case, the auxiliary structure will include (among other information to be specified shortly) the binary encoding $enc(T)$ of the input T, and will provide, for each node v in $enc(T)$, the set $types(v)$ consisting of the possible states of A at node v after consuming the subtree rooted at v. Once the auxiliary structure is updated, validity amounts to checking that $types(root(enc(T)))$ contains the accept state of A, where T is the updated tree. The strategy for updating the types associated with nodes applies the divide-and conquer strategy for string validation to certain paths in the tree, chosen to appropriately divide the work. More precisely, we will select, in every subtree T_0 of a given tree $enc(T)$, a particular path from the root to a leaf. We call this path the *principal line* of T_0, denoted by $line(T_0)$, and defined as follows:

—$root(T_0)$ belongs to $line(T_0)$;
—let v be an internal node of T_0 that belongs to $line(T_0)$, and suppose v has children v_1, v_2. If $|tree(v_1)| \geq |tree(v_2)|$, then v_1 belongs to $line(T_0)$; otherwise, v_2 belongs to $line(T_0)$.

Validation of $enc(T)$ can be done by associating to each maximal principal line[8] an NFA that validates that particular line. We make this more precise next.

From BNTA to NFA on principal lines. Consider the principal line $v_1 \dots v_n$ of a binary tree encoding $enc(T)$ where v_1 is the root and v_n is a leaf. By the definition of binary encodings, each non-leaf node v_i has one child v_i' that does *not* belong to the principal line $v_1 \dots v_n$, for $1 \leq i < n$. Consider the sets $types(v_i')$. Note that if these sets are given, we can validate $enc(T)$ by an NFA N that works on the string $v_1 \dots v_n$. For technical reasons, the constructed NFA recognizes the reverse word $v_n \dots v_1$. Essentially, the NFA guesses a sequence of state assignments to $v_n \dots v_1$ that is compatible with the transition function of A, given the sets of states $types(v_i')$.

The above intuition is captured as follows: We define new labels for the nodes v_i, which include both $\lambda(v_i)$ and the set $types(v_i')$. More precisely, let $\Sigma' = \{\#\} \cup (\mathcal{P}(Q) \times \Sigma) \cup (\Sigma \times \mathcal{P}(Q))$ and λ' be the labeling function defined as follows:

- $\lambda'(v_i) = \langle \lambda(v_i), types(v_i') \rangle$, if v_i' is the right child of v_i, $1 \leq i < n$,
- $\lambda'(v_i) = \langle types(v_i'), \lambda(v_i) \rangle$, if v_i' is the left child of v_i, $1 \leq i < n$.
- $\lambda'(v_n) = \lambda(v_n) = \#$.

The NFA N we construct will accept the string $\lambda'(v_n) \dots \lambda'(v_1)$ iff $A = \langle \Sigma_\#, Q, Q_0, q_f, \delta \rangle$ accepts $enc(T)$. At any rate, it will compute the type derived by the sequence. More precisely, let $N = \langle \Sigma', Q, q_0, F', \delta' \rangle$, where Σ' is as described above, $F' = \{q_f\}$, and δ' is defined by the following (and is empty everywhere else)

- $\delta'(\#, q_0) = Q_0$;

[8] A principal line is maximal if it is not included in another principal line.

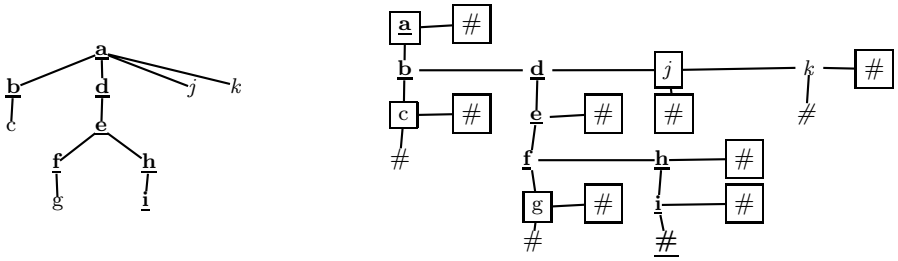


Fig. 5. A tree T (top) and its encoding $enc(T)$

- $\delta'(\langle a, S\rangle, q) = \bigcup_{q' \in S} \delta(a, q, q')$ for $a \in \Sigma$
- $\delta'(\langle S, a\rangle, q) = \bigcup_{q' \in S} \delta(a, q', q)$ for $a \in \Sigma$

Intuitively, the NFA simulates A by allowing only state transitions compatible with the transition function of A and the sets of states associated to siblings. It is easy to verify that N works as desired.

Note that the number of states of N is $O(|Q|)$. Recall that $|Q|$ is itself $O(|\Sigma^t||d|)$ where $\tau = \langle \Sigma, \Sigma^t, d, \mu\rangle$ is the specialized DTD to which the BNTA A corresponds. The size of its alphabet Σ' is $O(|\Sigma|2^{|Q|})$ which is $O(|\Sigma|2^{|\Sigma^t||d|})$. Hence, each symbol in Σ' can be represented in space $O(|\Sigma^t||d| + \log|\Sigma|)$. Notice however that our auxiliary structure never represents the alphabet or the transition mapping of N explicitly.

The auxiliary structure. The auxiliary structure used for incremental validation includes (i) the binary tree $enc(T)$, (ii) the set of maximal principal lines, as explained below and (iii) for each maximal principal line in $enc(T)$, the auxiliary transition relation tree for the NFA corresponding to that line.

Note that the principal lines can be specified concisely by annotating each node in $enc(T)$ with 0 or 1 by a labeling μ as follows: $\mu(root(enc(T))) = 0$, and for every pair of siblings v_1, v_2, $\mu(v_1) = 1$ and $\mu(v_2) = 0$ if $|tree(v_1)| \geq |tree(v_2)|$; otherwise, $\mu(v_1) = 0$ and $\mu(v_2) = 1$. Clearly, the principal line of a subtree T_0 is the unique path from $root(T_0)$ to a leaf where all non-root nodes are labeled 1. Note that the principal line of T_0 is maximal iff $\mu(root(T_0)) = 0$.

For example, consider the unranked tree represented in Figure 5 (top), and its binary encoding in the same figure (bottom). In the binary encoding in the figure, the nodes w for which $\mu(w) = 0$ are those inside a box. Note that this identifies all maximal principal lines. The bold and underlined nodes participate in the principal line of $enc(T)$. The nodes of one of the secondary principal lines (line j, k) are in italics.

Part (iii) of the auxiliary structure provides the transition relation trees for the NFAs associated with the maximal principal lines. The size of each transition relation tree for an NFA N is $O(|enc(T)||Q|^2)$ where Q is the number of states of N.

In summary, consider an input tree T and a specialized DTD $\tau = \langle \Sigma, \Sigma^t, d, \mu\rangle$. In view of our construction of the BNTA A from τ (Lemma 5.1), of the NFA N

from A (above), and of the tree of transition relations for each NFA N (Section 3) it follows that the size of the auxiliary structure associated with T and τ is $O((|\Sigma^t|^2|d|^2|T|)$.

Validation and maintenance for label renamings. Let us consider first the validation and maintenance of updates consisting of label renamings. Note that label renamings in T translate straightforwardly to label renamings in $enc(T)$. To validate a sequence of label renamings, it is sufficient to show how the auxiliary structure is maintained for a single renaming. For a sequence of renamings this is iterated one update at a time and validity is checked at the end using the updated auxiliary structure. So, suppose the label of some node v in $enc(T)$ is modified from a to b. Suppose first that v belongs to the maximal principal line $l = v_1 \dots v_n$ of $enc(T)$, say $v = v_k$. In the string $\lambda'(v_1) \dots \lambda'(v_n)$ the label renaming corresponds to modifying the label of v_k from a to b if $k = n$ and from $\langle a, types(v'_k)\rangle$ to $\langle b, types(v'_k)\rangle$ if $k < n$ and v'_k is the right child of v_k (left is analogous). Then the transition relation tree associated to l is updated as in the string case in time $O(|Q|^2 \log |Q| \log |l|)$, that is $O(|\Sigma^t|^2|d|^2 \log(|\Sigma^t||d|) \log |l|)$. Since $|l|$ is $O(|enc(T)|)$ and $|enc(T)|$ is $O(|T|)$, the update takes time $O(|\Sigma^t|^2|d|^2 \log(|\Sigma^t||d|) \log |T|)$.

Now suppose that v does not belong to the principal line l of $enc(T)$. Then there is some $k > 0$ such that v belongs to $tree(v'_k)$ where v'_k is the child of some v_k belonging to l. Note that the update to the label of v may cause a change in the value of $types(v'_k)$. In order to update l, we now have to first compute the new value for $types(v'_k)$, then apply the update procedure for the corresponding modification in the label $\langle \lambda(v_k), types(v'_k)\rangle$ of v_k. If v belongs to the principal line l' of $tree(v'_k)$ then the transition relation tree associated with the NFA for l' can be updated as before in time $O(|\Sigma^t|^2|d|^2 \log(|\Sigma^t||d|) \log |T|)$. This provides, in particular, the new value for $types(v'_k)$. Continuing inductively, it is clear that renaming the label of a node v affects precisely the maximal principal lines encountered in the path from root to v. Let M be the number of such maximal principal lines. Clearly, M is precisely the number of nodes w along the path from root to v for which $\mu(w) = 0$. We next provide a bound on this number, using the notion of *line diameter* of a tree.

DEFINITION 5.1. (**Line diameter**) The *line diameter* of $enc(T)$ is the maximum number of distinct maximal principal lines crossed by any path from root to leaf in $enc(T)$. Equivalently, the line diameter of $enc(T)$ is the maximum number of nodes w for which $\mu(w) = 0$, occurring along a path from root to leaf in $enc(T)$, where μ is defined as above. ♢

For example, the line diameter of $enc(T)$ in Figure 5 is 3. We can show the following:

LEMMA 5.2. *The line diameter of* $enc(T)$ *is no larger than* $1 + \log |enc(T)|$. ♢

PROOF. Consider a path from root to leaf in $enc(T)$ and let $w_1 \dots w_M$ be the sequence of nodes w along the path for which $\mu(w) = 0$. Note that, by the definition of μ, w_1 is the root of $enc(T)$, and each node w_i other than w_M has two children w'_i and w''_i where w'_i is on the path from w_i to w_{i+1} and $|tree(w'_i)| \geq |tree(w''_i)|$. It easily follows that $|enc(T)| \geq 2^{M-1}$ so $M \leq 1 + \log |enc(T)|$. □

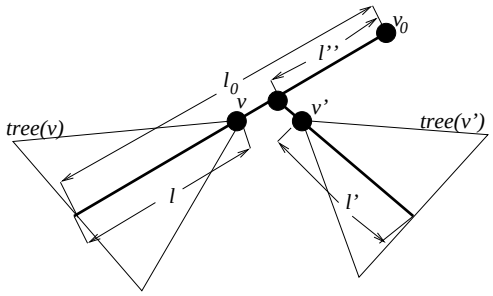


Fig. 6. Scenario of Line Rearrangement

From the bound on the line diameter of $enc(T)$, it follows that a label renaming can cause at most $O(\log |enc(T)|)$ updates to distinct transition relation trees of maximal principal lines in $enc(T)$. Since each update takes time $O(|\Sigma^t|^2|d|^2 \log(|\Sigma^t||d|) \log |T|)$, the entire auxiliary structure can be updated in time $O(|\Sigma^t|^2|d|^2 \log(|\Sigma^t||d|) \log^2 |T|)$ (and in $O(m|\Sigma^t|^2|d|^2 \log(|\Sigma^t||d|) \log^2 |T|)$ for a sequence of m updates).

Insertions and deletions. We next describe how to extend the maintenance and validation algorithm described above to updates that include insertions and deletions.

For a maximal principal line l in $enc(T)$, we denote by N_l the NFA corresponding to l and by $\mathcal{T}_l$ the transition relation tree corresponding to l and N_l.

Note that each insertion or deletion of a leaf node in T translates into up to four node insertions and deletions into $enc(T)$ (for example, deleting a node in T may require deleting in $enc(T)$, besides the node itself, up to two leaves labeled #, and may require inserting another such leaf). This constant factor blow-up in the number of updates does not affect our analysis.

Insertions and deletions are handled by an extension of the technique used to maintain the transition relation trees for maximal principal lines in the case of label renamings. Insertions and deletions that do not cause a change in the set of maximal principal lines existing prior to the update are handled straightforwardly. More precisely, let us call an insertion or deletion *line preserving* if the restriction of μ to the nodes of $enc(T)$ that are not affected by the update is the same before and after the update. Note that an insertion may be line preserving but nonetheless introduce a new singleton maximal principal line consisting of the new node. Also observe that line-preserving updates affect precisely the maximal principal lines intersected by the path from the root of $enc(T)$ to the newly inserted node or to the parent of the deleted node. The transition relation trees for these maximal principal lines are updated as in the case of label renamings, at the same cost. If a new singleton maximal line l consisting of an inserted node needs to be added, computing its auxiliary transition relation tree takes additional time $O(|Q|^2)$ where Q is the set of states of the NFA N_l. This is dominated by the rest of the cost.

Handling inserts and deletes that are not line preserving requires more care. In this case, the set of maximal principal lines in $enc(T)$ changes as the result of updates. To illustrate the problem, consider the situation depicted in Figure 6. The maximal principal line $l_0 = line(tree(v_0))$ contains a node v, which has a

sibling v'. Initially, $|tree(v)| \geq |tree(v')|$. However, a deletion in $tree(v)$ or an insertion in $tree(v')$ may make $tree(v')$ larger than $tree(v)$. In this case a new line structure is needed, where the line $l = line(tree(v))$ becomes a maximal principal line and the new principal line $line(tree(v_0))$ is the concatenation of l'' and $l' = line(tree(v'))$. This requires updating the auxiliary structure in two steps: First we compute the transition relation tree $\mathcal{T}_l$ for the new maximal principal line l obtained by truncating l_0. Then we compute the transition relation tree $\mathcal{T}_{l''l'}$ for the new maximal principal line obtained by concatenating l'' and l'.

Fortunately, the new transition relation trees can be computed efficiently from the old ones. Specifically, $\mathcal{T}_l$ is obtained by truncating $\mathcal{T}_{l_0}$, and $\mathcal{T}_{l''l'}$ is obtained by merging a subtree of $\mathcal{T}_{l_0}$ corresponding to l'' with the tree $\mathcal{T}_{l'}$. This is done by adapting usual B-tree techniques. We next provide more details.

Given the balanced tree $\mathcal{T}_{l_0}$ of the line l_0, we compute the balanced tree $\mathcal{T}_l$ by traversing bottom-up the path in $\mathcal{T}_{l_0}$ from the leaf that contains v to the root. Note that the path has maximum length $\lceil \log |l_0| \rceil$. At each cell n along the path we delete the relations T_{s_1} where $s_1 \cap l = \emptyset$ and we recompute the relation T_s, where the segment s contains v. Recall that each cell in $\mathcal{T}_{l_0}$ has between two and three relations, so it is not possible for any cell to become empty after these deletions. In addition, if the deletions have left only the relation T_s at cell n then we do the following, assuming n is not the root (the case where n is root is simple):

—if the right sibling n' of n has two relations we delete n and we transfer T_s (and the corresponding child node) to n'. In the parent of n and n' we delete the relation that corresponds to n and we continue our processing at the parent of n and n'.

—if the right sibling n' of n has three relations we move its leftmost relation (and the corresponding child node) to the cell n, so that n also has two relations. We recompute the entry of n' at the parent of n and n'.

—if n has no right sibling the we delete n and we copy T_s at the parent cell. Notice that this case reduces the depth of the balanced tree.

In all cases we continue recursively with the parent cell. The complexity of this procedure is $O(|Q|^2 \log |Q| \log |l_0|)$ where Q is the set of states of N_{l_0}, since the size of the traversed path is at most $\lceil \log |l_0| \rceil$ and in each step we recompute at most two relations.

Next, consider the computation of the balanced tree $\mathcal{T}_{l''l'}$ of the new main line $l''\ l'$. First, we compute a balanced tree $\mathcal{T}_{l''}$ for the segment l'' in $O(|Q|^2 \log |Q| \log |l''|)$. Then we merge $\mathcal{T}_{l''}$ and $\mathcal{T}_{l'}$ as follows. Assume that the depth of $\mathcal{T}_{l''}$ is equal or less to the depth of $\mathcal{T}_{l'}$ - the other case is symmetrical. Locate a node n_2 on the leftmost path of $\mathcal{T}_{l'}$ such that the depth of the tree rooted at n_2 is $depth(\mathcal{T}_{l''})$. Then insert each segment (and corresponding child node) of the root of $\mathcal{T}_{l''}$ into n_2. The insertions are handled as usual: if there is not enough space in n_2 then n_2 will be split, and so on. It is easy to see that the merge takes $O(|Q|^2 \log |Q| \log |l'|)$ since we have to recompute one or two relations at each level on the path from n_2 to the root of $\mathcal{T}_{l'}$. Overall, the rearrangement of these lines requires $O(|Q|^2 \log |Q| (\log |l_0| + \log |l'|))$, which is $O(|Q|^2 \log |Q| \log |enc(T)|)$. Also, note that a single insertion or deletion may cause at most $O(\log |enc(T)|)$

line rearrangements (one for each maximal principal line intersected by the path from root to the affected node). Thus, all line rearrangements can be done in time $O(|Q|^2 \log |Q| \log^2 |enc(T)|)$. In terms of the original specialized DTD and input tree T, this is $O((|\Sigma^t|^2|d|^2 \log(|\Sigma^t||d|) \log^2 |T|)$.

Once the line rearrangements have been computed, additional updates to the transition relation trees of maximal principal lines may have to be computed, as in the case of label renamings. This takes again time $O((|\Sigma^t|^2|d|^2 \log(|\Sigma^t||d|) \log^2 |T|)$.

In summary, the size of the auxiliary structure used for incremental validation is $O((|\Sigma^t|^2|d|^2|T|)$. Maintaining the auxiliary structure and validating the updated tree following a sequence of m updates (label renamings, insertions, or deletions) is done in time $O(m|\Sigma^t|^2|d|^2 \log(|\Sigma^t||d|) \log^2 |T|)$.

Similarly to the DTD and XML Schema validation cases, insertion of m subtrees containing M nodes can be implemented in $O(M \log^2 |T|)$ by a sequence of insertions of the individual nodes of subtrees. A more efficient implementation that encodes each subtree and validates its type in "batch" mode is $O(M + m \log^2 |T|)$.

6. IMPLEMENTATION

We implemented incremental and local validation algorithms on the XCacheDB XML database [Balmin and Papakonstantinou] which is built on top of a commercial relational database engine. The XCacheDB gives an administrator control over the decomposition of XML data into relations of the underlying RDBMS. For the purposes of this study we decomposed the XML data into normalized relational schemas. The following description applies exclusively to the normalized approach, which is similar to the "hybrid inlining" of [Shanmugasundaram et al. 1999]. [Balmin and Papakonstantinou] provides a description of alternative approaches.

The XCacheDB creates a table for every *repeatable* element in the DTD. We say that an element is repeatable if it can occur an unlimited number of times in a sibling list. A table contains zero or one data attributes, one system-generated node ID attribute, and at least one of `prnt`, `rsib`, and `tid`. A data attribute is created for every element with string content. We encode the tags of elements in the elements' IDs to efficiently identify the table that needs to be accessed given an element's ID. To preserve the parent-child relationship, each table includes a parent reference `prnt`, with the exception of the table that stores the root element of the DTD. We extended XCacheDB to preserve the document order of XML elements, by adding the `rsib` attribute to every relation in the schema. This attribute stores the ID of the element's right sibling. Since the ID encodes the tag, the value of `rsib` also determines the table in which the sibling is stored.

Every element whose parent requires validation[9] also stores a `tid` attribute. The `tid` is a foreign key into the transtion relation storage table, which we will describe shortly, and stands for the pointers from elements to transition relation tree leafs.

We illustrate the XCacheDB data storage and auxiliary storage used for incremental validation with an example, which will be the running example of this section.

[9]We do not validate trivial regular expressions, with minimal DFAs containing a single state.

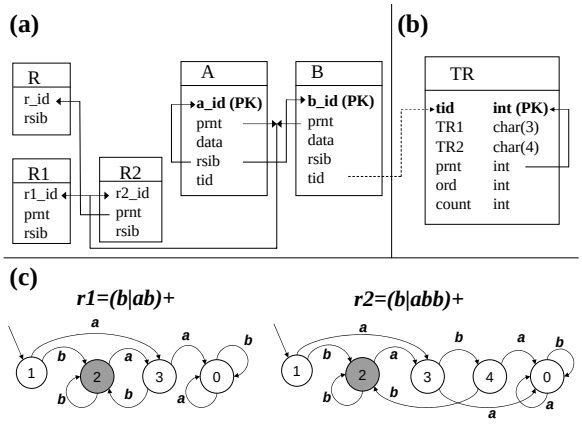


Fig. 7. Relational schema of XCacheDB and TR storage.

EXAMPLE 6.1. Consider the following DTD:
$r = (r1|r2)*$
$r1 = (b|ab)+$
$r2 = (b|abb)+$
where a and b are elements with string content. The DTD is based on the Well-LogML DTD [Well], which contains the expression
$CurveData = (data|(piValue, data))+$. To illustrate the validation of the renaming of intermediate nodes, we added the $r2$ element to the DTD. The minimal DFAs of the expressions $r1$ and $r2$ are shown in Figure 7 (c). Figure 7 (a) shows the relational schema created by the XCacheDB. The tables `R`, `R1`, `R2`, `A` and `B` hold the data of the XML document. The string content of elements a and b is stored in the `data` attributes of tables `A` and `B` respectively. Tables `A` and `B` include parent references to $r1$ or $r2$ elements, since both $r1$ and $r2$ can have a and b as their children. For example, `A.prnt` references either `R1.r1_id` or `R2.r2_id`. Recall that the element IDs encode the tags of elements. In this example, the last two bits of the element ID determine whether the element is a, b, $r1$, or $r2$. Thus by looking at the value of the parent attribute of an element we immediately know whether it references $r1$ or $r2$. In Figure 8 we explicitly show the tags as part of the elements' ID for clarity of exposition. For example, $3b$ is the ID of the b_3 element, and $1r1$ is the ID of $r1_1$. ♢

We implemented the three basic update operations, insert, delete and rename, as well as three validation algorithms, incremental (Section 4), local (Section 4.4), and a naive, which involves reading the whole sibling list and re-validating it from scratch.

Incremental Validation. The transition relations (TR) trees required by the incremental update validation algorithm are stored in the `TR` relation, as illustrated in Figure 7 (b). We store m TR nodes per tuple, where m is the number of regular expressions that the string has to be validated against. The transition relations that correspond to the same sequence of nodes with respect to different regular expressions are always accessed simultaneously, and it is advantageous to

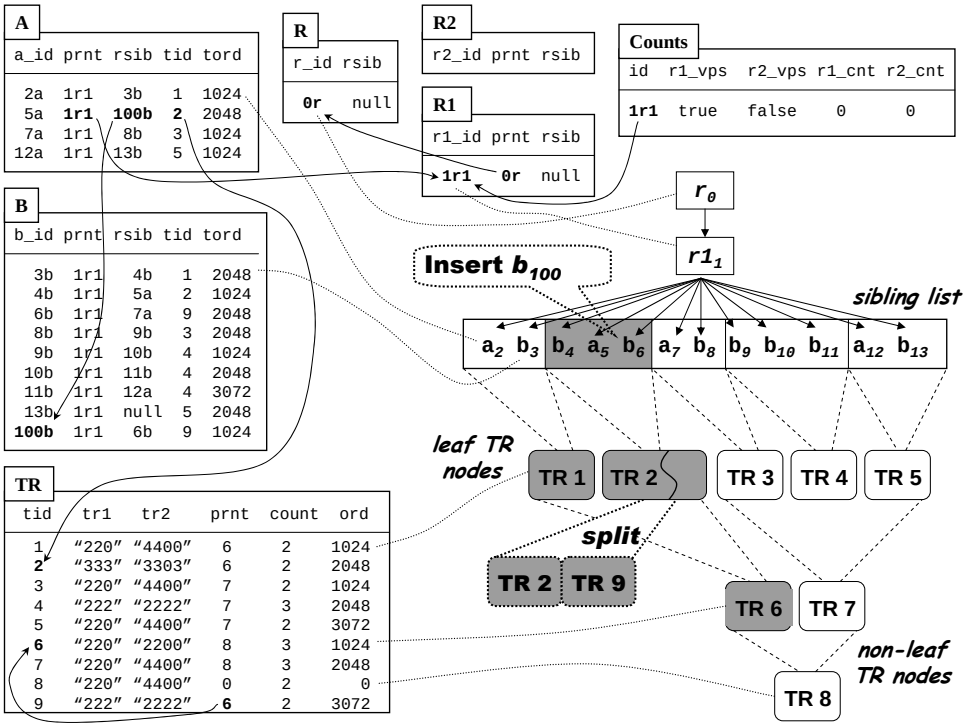


Fig. 8. The state of the storage after a new b element is inserted.

store them together. For instance, the DTD of Example 6.1 contains two regular expressions $r1$ and $r2$ that we need to be able to validate. Thus, the TR table for this DTD will have two columns `tr1` and `tr2` as shown in Figure 8. We do not need to validate trivial regular expressions, such as r, with minimal DFAs containing a single state.

Each transition relation is stored as a string of length n, where n is the number of states in the minimal DFA of the regular expression that is being validated. We assume that the minimal DFA has at most 256 states and, hence, a byte is enough to represent a state. The reject sink is by convention the state 0, and its transitions are not stored in the database. For example, the minimal DFAs for $r1$ and $r2$ are shown in Figure 7 (c). These DFAs have three and four potentially accepting states respectively. Hence, `tr1` is a string of size 3, and `tr2` has length 4.

Each tuple of the `TR` relation has a unique `tid` attribute, and a `prnt` attribute that references the parent TR node. The attribute `count` stores the number of children TR nodes, and `ord` is used for ordering sibling TR nodes, which map to the same parent. When assigning the `ord` numbers, we leave enough space to accommodate many updates without renumbering. Notice, that renumbering does not entail much overhead since the `ord` numbers have to be unique only among the children of the same TR node.

We create an index on the pair (`prnt`,`ord`), to facilitate efficient validation and TR splits. These operations require access to a list of sibling TR nodes.

EXAMPLE 6.2. Consider the XML fragment of Figure 8 that is valid with respect to the DTD of Example 6.1. Note that the `TR` table maintains transition relations that validate the list of siblings both against $r1$'s content definition (those relations are stored in `tr1`) and transition relations that validate against $r2$'s content definition (those relations are stored in `r2`) despite the fact the parent of the list of siblings is not $r2$. The reason is that, as we explained in Section 4, we need to be able to validate a renaming of $r1$ to $r2$ without validating the sequence of siblings from scratch. Assume that the TR tree is constructed for the list of a's and b's as shown in Figure 8. Assume that the maximum TR tree node size is 3 (nodes cannot have more than 3 children).

The first tuple of the TR relation in Figure 8, corresponds to the TR1 node, which covers the substring a_2b_3. If we run this string on the $r1$ DFA, the automaton will terminate in states 2, 2, and 0 if it was initialized at states 1, 2, and 3 respectively. Thus, the transition relation `TR.tr1` for this substring is encoded by "220".

Now consider an insertion of a new b element with $id = 100$ between the fourth and fifth child of $r1$. Figure 8 shows the state of the database after the insertion. Shading indicates elements and transition relations that were accessed to validate the insertion. Notice that a new TR node ("TR 9") has to be created, since "TR 2" cannot have 4 children elements, due to the maximum TR node size assumption. ♢

Local Validation. The table `Counts` stores counters used for validation of complex node renames by the local validation algorithm described in Section 4.4. For each complex element, i.e., for each internal node of the data tree, we store a tuple that contains the element's ID and a pair of `vps` and `cnt` attributes for each regular expression that needs to be validated. The `vps` attribute is a boolean "valid prefix/suffix" flag which indicates whether conditions (2) and (3) of Lemma 4.1 are satisfied for the element's children list. The `cnt` attribute stores the number of rejecting sequences of length $k + 1$ in the list. The condition (1) of Lemma 4.1 is satisfied if this number is 0.

EXAMPLE 6.3. Figure 8 shows a `Counts` tuple that corresponds to $r1_1$. The list of the element's children is valid with respect to expression $r1$. Indeed, `r1_vps`=*true* and `r1_cnt`=*0*. However, this element cannot be renamed to $r2$, since the same list is not valid with respect to expression $r2$. Even though the sequence does not contain any rejecting substring (`r2_cnt`=*0*), the last two elements leave the minimal DFA of $r2$ in state 4, which is not accepting (see Figure 7 (c)). This breaks condition (3) of the lemma, hence `r2_vps`=*false*. ♢

To facilitate efficient access to the element's left sibling, which is required by the local validation algorithm, we create indices on `A.rsib` and `B.rsib`.

7. APPLICABILITY AND PERFORMANCE EXPERIMENTS

Local validation was applicable on the majority of 60 real-world DTDs found at OASIS and described at [Choi 2002]. In particular, there were only 21 non-local regular expressions in 10 DTDs; the total number of regular expressions was 2141. By investigating the documentation of the DTDs and contacting their authors we determined that 8 of those expressions describe potentially very large lists of data.

In addition, the minimal k, which affects the performance of the local validation algorithm was always less than 4. In total, 2070 expressions were 1-local, 23 expressions were 2-local but not 1-local, and 27 expressions were 3-local but not 2-local. 39 DTDs contained only 1-local expressions, 8 DTDs contained only 2-local expressions (recall, the set of 2-local expressions includes the set of 1-local expressions) and 3 DTDs contained only 3-local expressions; recall, 10 DTDs contained non-local expressions.

Next we experimentally compare the performance of three update validation algorithms: local validation, described in Section 4.2, general incremental validation described in Section 4, and a naive algorithm that involves reading whole sibling lists and re-validating them from scratch. We simulated a number of update scenarios and analyzed the running time and space overhead of the algorithms under various parameters. We have considered a case where the database objects are perfectly clustered and a case where a fraction of the database objects is not placed at the optimal clusters, which is a typical assumption for a database that is being updated.

7.1 Experiment Setup

All the experiments were performed on a 1.2 GHz Pentium system with 512 MB of memory and 5400 rpm hard drive. The XCacheDB server and the RDBMS were installed on the same system. To offset the relatively small dataset size, the RDBMS was configured to use only 16 MB for the buffer cache.

We used a synthetic XML dataset containing about 150000 elements. The dataset conformed to the DTD of Example 6.1, which is similar to the WellLogML DTD [Well]. The a and b elements have text content, which does not alter the validation algorithms, but its size affects the performance of the implementation since it increases the size of the dataset. Notice that this DTD is local, as the *r1* expression is 1-local, and *r2* is 2-local. Since the DTD is local, we use the same dataset to compare all three algorithms: local, incremental, and naive.

Scenarios and Parameters. We controlled the following parameters of the dataset, and of the Transition Relation (TR) storage.

(1) Sibling list size: Each dataset had 150000 leaf elements, evenly distributed between 10, 100, 1000, or 10000 top-level ($r1$ or $r2$) elements. Thus, the average length of a sibling list that had to be validated was 15000, 1500, 150, and 15 respectively. This is the number of elements that has to be accessed by a naive algorithm to validate each update.

(2) Size of the `data` attributes of relations `A` and `B`. These attributes store text content for the leaf a and b elements. We tried sizes 25, 100, and 400 bytes, which translated to 8MB, 20MB and 65MB relational database sizes respectively.

(3) Transition Relation (TR) Tree Leaf Node Size: This size refers to the maximum number of XML elements that can point to the same transition relation tree node. For every update the incremental algorithm has to read the data elements belonging to the same leaf TR node. For this parameter we tested values of: 4, 16, 64, and 256.

(4) TR Tree Non-Leaf Node Size: This size refers to the maximum number of children an internal TR tree node is allowed to have. For this parameter we

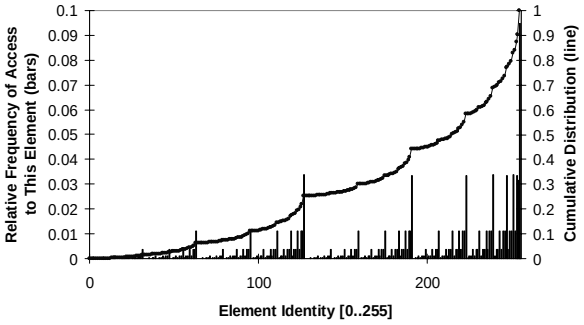


Fig. 9. Relative frequency and cumulative distribution of NURand() function, used in TPC-C.

tested values of: 4, 16, 64, 256 and 1024.

We tested the performance of element renames and three insert scenarios: uniformly distributed insertions, insertions distributed according to the TPC-C benchmark, and append-only access.

Unless otherwise specified, the experiments were conducted with the transition relation tree nodes being 75% full. For example, when the TR leaf node size was 4 and the non-leaf node size was 64, a leaf TR node would be created for every 3 XML elements and a non-leaf TR node would be created for every 48 sibling TR nodes.

In the random insertion scenario, 10000 b elements were inserted at uniformly distributed random points in the document. All insertions were done as a part of a single transaction. The incremental algorithm had to maintain the data structure for each insertion. Notice that all our transactions are valid to avoid an (orthogonal) issue of transaction roll-backs. Since the DTD is local, incremental validation requires reading only a single TR tree node, unless the insertion happens at distance less than k from the end of the sub-list of elements that point to the same transition relation tree leaf node.

In the "average case" insertion scenario, which follows TPC-C, the same 10000 b elements were inserted at points picked by the NURand() non-uniform random function. The cumulative distribution of this function, for the case when one element is picked out of a list of 256 elements, is shown in Figure 9 (taken from [Levine 1997]). The same random function is used to construct update transactions in the TPC-C benchmark [TPC-C].

In the "append-only" scenario the dataset was constructed in the same way. However, the 10000 b elements were all appended to one of the r lists. Notice, that this scenario caused the maximum number of node splits, and, at the same time, took maximum advantage of database caching.

We performed the experiments on both clustered and unclustered datasets. In the first case, each list of siblings was stored consecutively in the tables A and B, hence leading to perfect clustering of the data. In the second case, 15% of

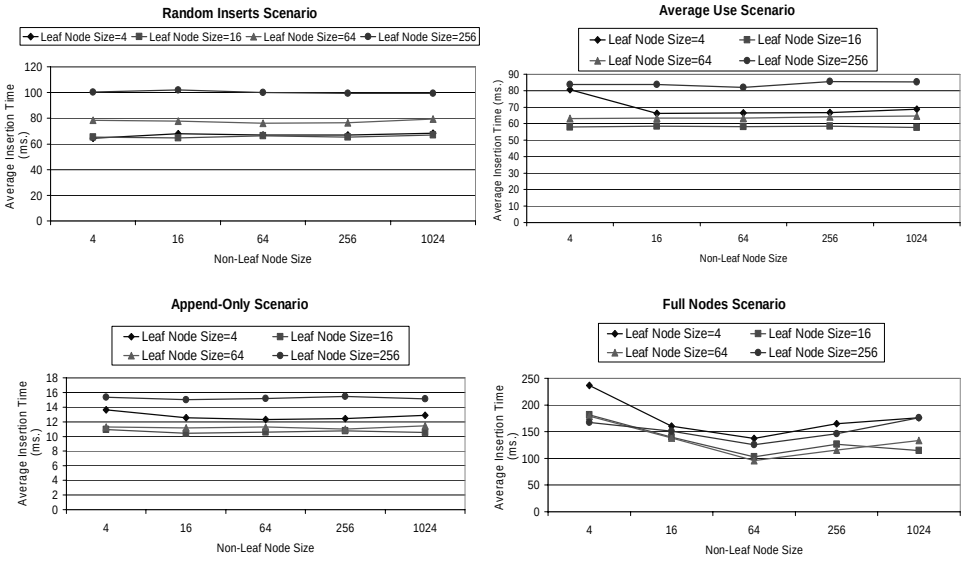


Fig. 10. Effects of the node size parameters under different update scenarios.

randomly picked records were shuffled around by deleting and reinserting them back into the table. The second case models a databases that has been updated up to 15%, without having been reorganized for clustering purposes yet. Without the perfect clustering the naive approach was at even greater disadvantage, as it requires accessing full sibling lists for each update.

Optimizing the TR parameters. First, we experimented with the TR node size parameters to maximize the performance of the incremental algorithm. Figure 10 shows the average insertion time, which includes actual insertion and incremental validation time, for the uniform, average and append scenarios described above. The last graph corresponds to the "full nodes" scenario, which is a random insertion scenario modified so that all nodes were initially created 100% full. Thus any insertion will trigger one or more node splits. This is essentially a worst-case scenario for the incremental validation algorithm, as it requires the highest number of node splits and the database server cannot take full advantage of caching, due to the random locations of the updates.

All four graphs of Figure 10 exhibit the tradeoff between large number of node splits needed for smaller node size and large sibling lists that need to be read to validate for larger node size. In all four scenarios, leaf node size of 16 performed better than very small and very large values.

The TR leaf node size affects the performance more than the size of the internal TR nodes, since leaf TR tree nodes need to be accessed for every insertion, while internal ones are accessed much less frequently[10]. The sibling list size and data

[10]Since our DTD is locally updateable, non-leaf TRs are accessed only when the leaf node is split, or when the updated element happens to be within k of the end of sibling sub-list that maps to a particular TR leaf node.

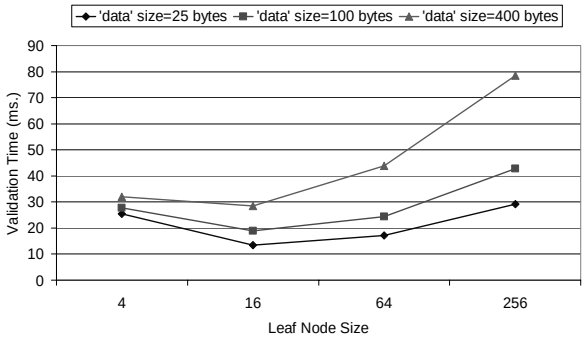


Fig. 11. Effects of the "data" attribute size.

attribute size for these experiments were fixed at 15000 and 100 respectively and the dataset was perfectly clustered. We don't include the graphs for different sibling list sizes and unclustered data, since they are very similar with the ones of Figure 10.

The effects of the size of the `data` attribute on the "average-case" scenario are shown in Figure 11. The non-leaf node size does not significantly affect this experiment and was fixed at 64. We report only validation time, as the insertion time, naturally, increases with the `data` size and creates the same offset for all methods.

Notice that in all three cases the leaf size 16 performed the best. However, with 25 byte `data` attributes, leaf size 64 performed better than leaf size 4, while with 400 byte `data` attributes the opposite can be observed. The reason is that with larger column size fewer tuples fit on a database page. Thus there is higher chance that an update validation will require access to multiple pages. This increases the negative effects of larger leaf node sizes.

In the rest of the experiments reported in this section we fix leaf and non-leaf node sizes to be 16 and 64 respectively, since these values consistently provide good performance.

Comparing the algorithms. Figure 12 shows that the local and incremental update validation algorithms are virtually insensitive to the sibling list size, while the naive algorithm scales almost linearly. Notice that the graphs are in logarithmic scale. The `data` attribute size was fixed at 100.

The local validation algorithm is a winner even when the updated element has as few as 15 siblings. This is remarkable considering how much more efficient the local validation implementation could have been if we had lower level access to the data. With small sibling list size, performance of the incremental validation is comparable to that of the naive algorithm. As the sibling list size grows, the incremental validation outperforms the naive by more than an order of magnitude (sibling list of length 15000).

Naturally, all three algorithms perform worse on non-clustered data, as they cannot take full advantage of caching and prefetching done by the database server. However, the naive algorithm's performance is impacted more since it has to access

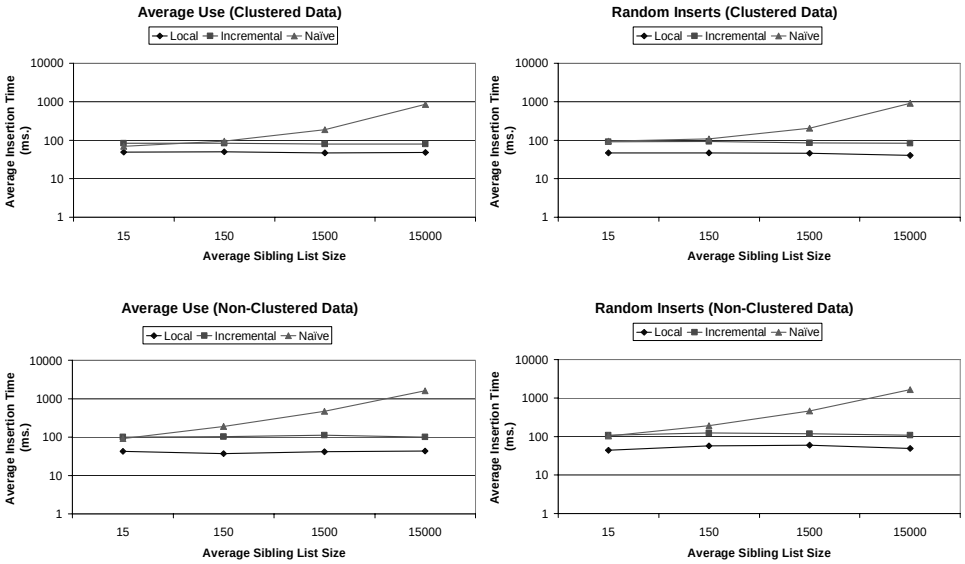


Fig. 12. Performance of the three validation algorithm under different update scenarios on clustered and non-clustered datasets.

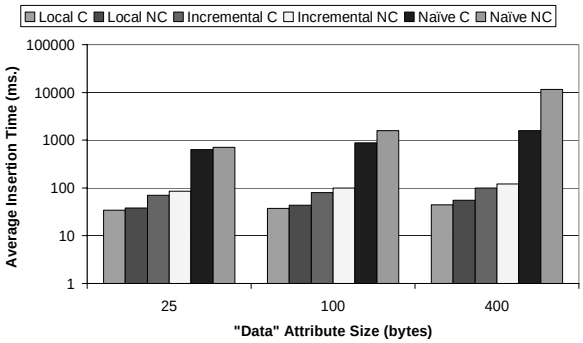


Fig. 13. Effects of the `data` attribute size on the three validation algorithms, on clustered and non-clustered datasets, in "average-use" case.

more data per update.

Figure 13 shows effects of the `data` attribute size on all three validation algorithms in the "average-use" case. The sibling list size was fixed at 15000. Once again, the graph is in logarithmic scale. With all three algorithms the validation for larger `data` size takes longer as it requires reading more data. Notice that with small `data` size, clustering almost does not impact the performance of the naive algorithm. In this case the entire database fits in the buffer space of the RDBMS. However, when the `data` attribute size is large and the database is 4 times bigger than the

node size	*fan-out*	*time (sec)*	*size (KB)*
4	15000	194	1875
16	15000	40	341
64	15000	12.5	80
256	15000	6.6	20
1024	15000	5.2	5.3
4	1500	198	1886
16	1500	59	344
64	1500	26	82
256	1500	19.8	23
1024	1500	19	7.5
4	150	282	1912
16	150	128	395
64	150	103	126
256	150	92	25
1024	150	94	25
4	15	1068	2027
16	15	1051	685
64	15	881	244
256	15	881	244
1024	15	872	244

Table I. Overhead of Transition Relations

buffer space, the naive algorithm slows down by almost an order of magnitude when running on non-clustered data. In this case the data has to be read from disk, and without the clustering, the algorithm cannot take full advantage of the prefetching.

Construction Overhead of Transition Relations. Table I shows time and space required to initially construct transition relations, with various node sizes, for datasets with average sibling list sizes 1500, 150, and 15. Notice that larger TR node sizes do not incur larger overhead. Local validation algorithm also has space overhead as it stores an integer counter for each complex element and each regular expression that the element is validated against. In our experiments this overhead ranged from 80 bytes (10 internal nodes) to 80 KB (10000 internal nodes).

Summary. We compared naive ("from scratch") update validation with the described in Section 6 implementation of general incremental validation and local validation, in the context of an XML database built on a relational database. The following key results and guidelines emerged. First, local validation should always be used, when applicable, since it outperforms naive and general incremental validation in all scenarios. Fortunately, our investigation in real-life DTDs posted at OASIS showed that local DTDs are very common. Second, the tuning of the leaf node size parameters of the auxiliary structure needed for general incremental validation is relatively easy, since values in the 16-64 range consistently provided good results. General incremental validation marginally outperforms naive validation for sibling list sizes around 150 and significantly outperforms naive validation for sibling list sizes in the thousands and beyond. The relative comparison results were not significantly sensitive to how the data were clustered and the pattern of the sequence of updates.

8. CONCLUSIONS AND FUTURE WORK

The incremental validation algorithms we exhibited are significant improvements over brute-force validation from scratch. However, several issues on update validation need further investigation:

Lower bounds. To understand how close our algorithms are from optimal, it would be of interest to exhibit lower bounds on incremental maintenance of strings, DTDs, and specialized DTDs. There are known results that yield lower bounds for validation from scratch: acceptance of a tree by a tree automaton is complete for uniform NC^1 under DLOGTIME reductions [Lohrey 2001]. However, this does not seem to yield any non-trivial lower bound on the incremental validation problem. We are not aware of any work providing such lower bounds applicable to our framework.

Optimizing over multiple updates. For a sequence of m updates, our incremental validation algorithm modifies the auxiliary structure one update at a time, then checks validity of the final updated tree. Clearly, it is sometimes more efficient to consider groups of updates at a time. For example, this may avoid performing unnecessary intermediate line rearrangements in the incremental algorithm for specialized DTDs. Also, if the number of updates is large compared to the size of the resulting tree, it may be more efficient to re-validate from scratch.

More complex updates on trees. We only considered here elementary updates affecting one node at a time. Some scenarios, such as XML editors, require more complex updates arising from manipulation of entire subtrees (deletion, insertion, cut-and-paste, etc). Our approach can still be applied by reducing each of these updates to a sequence of elementary updates. However, in this case it may be more efficient to consider updates of coarser granularity.

Update Languages. It is expected that XQuery will soon be augmented with an update language [Sur et al. 2004; Tatarinov et al. 2001]. Systems supporting complex update languages can naturally use our work: first compute the set of updates of particular nodes and then apply the incremental validation techniques described in this paper. However, this approach may miss the extra optimization opportunities presented by the fact that the set of updates has been developed by a single update statement. Realizing those opportunities requires analysis of the update statement.

ACKNOWLEDGMENTS

We would like to thank Byron Choi for providing us a collection of DTDs and the source code of the DTD Inquisitor tool used in [Choi 2002]. We are also grateful to Jayavel Shanmugasundaram for useful discussions on the problem. Finally, we are grateful to the ACM TODS reviewers for their constructive comments.

REFERENCES

Balmin, A. and Papakonstantinou, Y. Storing and querying XML data using denormalized relational databases. *VLDB Journal*. To appear. Also available at `http://www.db.ucsd.edu/people/andrey`.

Barbosa, D., Mendelzon, A., Libkin, L., Mignet, L., and Arenas, M. 2004. Efficient incremental validation of xml documents. In *Proceedings of the 20th International Conference on Data Engineering (ICDE'04)*. IEEE Computer Society, Los Alamitos, CA.

Beeri, C. and Milo, T. 1999. Schemas for integration and translation of structured and semistructured data. In *Proceedings of the 7th Int'l. Conf. on Database Theory*. Lecture Notes in Computer Science, vol. 1540. Springer, New York.

Bruggemann-Klein, A., Murata, M., and Wood, D. 2001. Regular tree and regular hedge languages over non-ranked alphabets. Tech. Rep. HKUST-TCSC-2001-05, Hong Kong Univ. of Science and Technology Computer Science Center. Available at http://www.cs.ust.hk/tcsc/RR/2001-05.ps.gz.

Bruggemann-Klein, A. and Wood, D. 1998. One-unambiguous regular languages. *Information and Computation 142,* 2, 182–206.

Choi, B. 2002. What are real dtds like? In *Proceedings of the Fifth International Workshop on the Web and Databases, WebDB*. Informal proceedings, 43–48.

Cluet, S., Delobel, C., Simeon, J., and Smaga, K. 1998. Your mediators need data conversion! In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*. ACM, New York, 177–188.

Cormen, T., Leiserson, C., and Rivest, R. 1990. *Introduction to Algorithms*. MIT Press, Cambridge, MA.

Dong, G. and Su, J. 1995. Space-bounded foies. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 22-25, 1995, San Jose, California*. ACM, New York, 139–150.

Garcia-Molina, H., Ullman, J., and Widom, J. 2001. *Database Systems: The Complete Book*. Prentice Hall, Upper Saddle River, NJ.

Ghezzi, C. and Mandrioli, D. 1980. Augmenting parsers to support incrementality. *Journal of the ACM 27,* 3, 564–579.

Hesse, B. and Immerman, N. 2002. Complete problems for dynamic complexity classes. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS 2002)*. IEEE Computer Society, Los Alamitos, CA.

Ipedo. Ipedo XML Database: Technical overview. Available at `http://www.ipedo.com/downloads/products_ixd_technical_overview.pdf`.

Jalili, F. and Gallier, J. 1982. Building friendly parsers. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*. ACM, New York.

Larcheveque, J. 1995. Optimal incremental parsing. *ACM Transactions on Programming Languages and Systems 17,* 1, 1–15.

Levine, C. 1997. Standard benchmarks for database systems. Presented at Sigmod 97. Available at `http://www.tpc.org/information/sessions/sigmod/indexc.htm`.

Li, W. 1995. A simple and efficient incremental LL(1) parsing. In *Proceedings of the 22nd Seminar on Current Trends in Theory and Practice of Informatics(SOFSEM '95)*. Lecture Notes in Computer Science, vol. 1012. Springer, New York.

Linden, G. 1993. Incremental updates in structured documents. Licentiate Thesis, Report C-1993-19, Department of Computer Science, University of Helsinki.

Lohrey, M. 2001. On the parallel complexity of tree automata. In *Proceedings of the 12th International Conference on Rewriting Techniques and Applications, (RTA 2001)*. Lecture Notes in Computer Science, vol. 2051. Springer, New York.

Miltersen, P., Subramanian, S., Vitter, J., and Tamassia, R. 1994. Complexity models for incremental computation. *TCS 130,* 1, 203–236.

Murching, A., Prasant, Y., and Srikant, Y. 1990. Incremental recursive descent parsing. *Computer Languages 15,* 4, 193–204.

Neven, F. 2002. Automata, logic and XML. In *Computer Science Logic, 16th International Workshop (CSL 2002)*. Lecture Notes in Computer Science, vol. 2471. Springer, New York. Available at `http://alpha.luc.ac.be/ lucg5503/publs.html`.

Papakonstantinou, Y. and Vianu, V. 2000. DTD inference for views of XML data. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2000), Dallas, Texas, USA*. ACM, New York, 35–46.

Patnaik, S. and Immerman, N. 1997. Dyn-FO: A parallel, dynamic complexity class. *JCSS 55,* 2, 199–209.

Petrone, L. 1995. Reusing batch parsers as incremental parsers. In *Foundations of Software Technology and Theoretical Computer Science*. Lecture Notes in Computer Science, vol. 1026. Springer, New York.

RELAX NG. http://www.relaxng.org.

Segoufin, L. 2002. Personal communication.

Shanmugasundaram, J., Tufte, K., Zhang, C., He, G., DeWitt, D. J., and Naughton, J. F. 1999. Relational databases for querying XML documents: Limitations and opportunities. In *Proceedings of 25th International Conference on Very Large Data Bases (VLDB'99), Edinburgh, Scotland, UK*. Morgan Kaufmann, San Francisco, CA, 302–314.

Sur, G., Hammer, J., and Simeon, J. 2004. UpdateX - an XQuery-based language for processing updates in XML. In *International Workshop on Programming Language Technologies for XML (PLAN-X 2004)*. Informal Proceedings.

Tatarinov, I., Ives, Z., Halevy, A., and Weld", D. 2001. Updating xml. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York.

TPC-C. Benchmark. Available at `http://www.tpc.org/tpcc/`.

Vollmer, H. 1999. *Introduction to Circuit Complexity*. Springer Verlag, New York.

W3C. 1998. The extensible markup language (XML). W3C Recomendation available at http://www.w3c.org/XML.

W3C. 2001. XML schema definition. W3C Recomendation available at http://www.w3c.org/XML/Schema.

Wagner, T. and Graham, S. 1998. Efficient and flexible incremental parsing. *ACM Transactions on Programming Languages and Systems 20,* 2, 980–1013.

Well. WellLogML DTD. Available at `http://www.posc.org/ebiz/WellLogML/`.

XML Edt. XML editor products. Available at `http://www.perfectxml.com/soft.asp?cat=6`.

XMLmind. XMLmind XML Editor. Available at `http://www.xmlmind.com/xmleditor/`.

XMLSpy. xmlspy document editor. Available at `http://www.xmlspy.com/products_doc.html`.

Received Month Year; revised Month Year; accepted Month Year