Fast Multi-update Operations on Compressed XML Data

Stefan Böttcher, Rita Hartel, and Thomas Jacobs

University of Paderborn, Computer Science, Fürstenallee 11, 33102 Paderborn, Germany
{stb@,rst@,tjacobs@mail.}uni-paderborn.de

Abstract. Grammar-based XML compression reduces the volume of big XML data collections, but fast updates of compressed data may become a bottleneck. An open question still was, given an XPath Query and an update operation, how to efficiently compute the update positions within a grammar representing a compressed XML file. In this paper, we propose an automaton-based solution, which computes these positions, combines them in a so-called Update DAG, supports parallel updates, and uses dynamic programming to avoid an implicit decompression of the grammar. As a result, our solution updates compressed XML even faster than MXQuery and Qizx update uncompressed XML.

Keywords: updating compressed XML data, grammar-based compression.

1 Introduction

Motivation: XML is widely used in business applications and is the de facto standard for information exchange among different enterprise information systems, and XPath is widely used for querying XML data. However, efficient storage, search, and update of big XML data collections have been limited due to their size and verboseness. While compression contributes to efficient storage of big XML data, and many compressed XML formats support query evaluation, fast updates of compressed XML formats involve the challenge to find and to modify only those parts of an XML document that have been selected by an XPath query.

Background: We follow the grammar-based XML compression techniques, and we extend an XML compression technique, called CluX, by fast multi-update operations, i.e. operations that update multiple XML nodes selected by an XPath query without full decompression. Like the majority of the XML compression techniques, we assume that textual content of text nodes and of attribute nodes is compressed and stored separately and focus here on the compression of the structural part of an XML document.

Contributions: Our paper presents a new and efficient approach to simulate multi-update operations on a grammar-compressed XML document. That is, given a grammar G representing an XML document D, and given an update operation O to be performed on all nodes N of D selected by an XPath query Q, we can simulate O's modification of all nodes N on G without prior decompression. To the best of our knowledge, it is the first approach that combines the following properties:

G. Gottlob et al. (Eds.): BNCOD 2013, LNCS 7968, pp. 149–164, 2013.
© Springer-Verlag Berlin Heidelberg 2013

Our approach computes all update positions in G determined by Q in such a way that paths through the grammar to these update positions can be combined to a so-called Update DAG. This Update DAG can be used for updating multiple XML nodes at a time without full decompression of the grammar G. The Update DAG construction combines dynamic programming, a top-down evaluation of Q's main path, and a bottom-up evaluation of Q's filters. As our performance shows, this outperforms competitive query engines like QizX and MXQuery which work on uncompressed documents.

Paper Organization: For simplicity of this presentation, we restrict it to XML documents containing only element nodes. The next section introduces the idea of grammar based XML compression and of executing updates in parallel on such grammars. Based on these fundamentals, we describe our main contribution, the goal of which is to execute an XPath query on a given grammar and to compute the Update DAG that supports parallel updates. The evaluation of the entire approach is then shown in Section 4.

2 Fundamentals and Previous Work

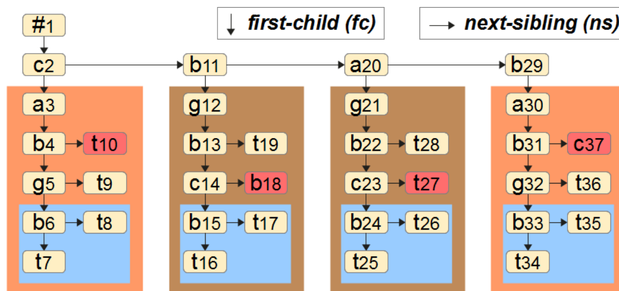

Fig. 1. Document tree of an XML document D with repeated matches of patterns

2.1 Sharing Similar Trees of XML Documents Using Grammars

Fig. 1 shows an example XML document D represented as a binary tree, where e.g. #'s first-child is c, the next-sibling of which is b. To distinguish multiple occurrences of node labels, we have numbered the nodes in pre-order. The simplest grammar-based XML compressors are those compressors that share identical sub-trees, such that the compressed grammar represents the minimal DAG of the XML tree [1]. These Approaches share identical sub-trees T in an XML document D by removing repeated occurrences of T in D, by introducing a grammar rule N→T, and by replacing each T by non-terminal N. Applying this approach to our example document D, the sub-tree b(t,t) is found with four matches, each of which is replaced by non-terminal A0.

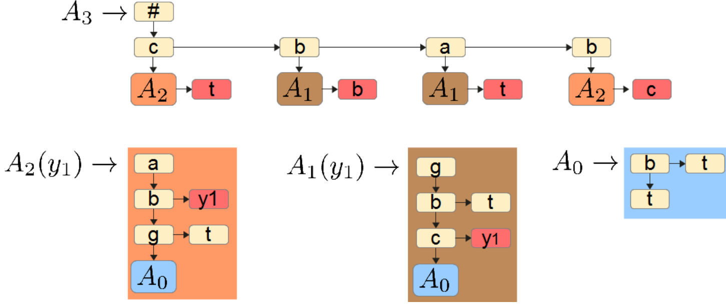

Fig. 2. Document of Fig. 1 with identical and similar sub-trees replaced by rule calls

However, a weakness of this approach is that only identical sub-trees can be compressed. In our example, the sub-trees rooted in nodes a3 and a30 differ only at the highlighted leaf nodes t10 and c37. By using approaches like CluX [2], BPLEX [3], or TreeRePAIR [4], we are able to compress those similar sub-trees by introducing parameterized grammar-rules. These grammar rules consist of the identical parts of the sub-trees and introduce parameters as placeholders for the different parts. Fig. 2 shows one possible resulting grammar, i.e. Grammar 2, represented as a set of trees. The similar sub-trees of Fig. 1 are replaced by non-terminals A1 and A2, which are the left-hand sides of new grammar rules in Grammar 1 containing y1 as a parameter.

```
A3      → #(c(A2(t),b(A1(b),a(A1(t),b(A2(c),ε)))),ε)
A2(y1) → a(b(g(A0,t),y1),ε)
A1(y1) → g(b(c(A0,y1),t),ε)
A0      → b(t,t)
```

Grammar 1: A grammar sharing similar sub-trees by using parameterized rules.

Each non-terminal Ai refers to exactly one grammar rule Ai(y1,y2,…,yn) →rhs(Ai), with rhs(Ai) being the right-hand side of that rule. We call yi a formal parameter (or just parameter). For a non-terminal expression Ai(t1,t2,…,tn) used in a right-hand side of a grammar-rule, we refer to each ti as an actual parameter. The grammars considered here are linear and straight-line. Linearity means that each parameter occurring on the left-hand side of a grammar rule appears exactly once in the right-hand side of that same rule. A grammar is straight-line, if the graph representing the rule calls is acyclic.

2.2 Using Grammar Paths to Identify Nodes

Each path to a selected node in an XML document D corresponds to exactly one grammar path (GP) in the grammar G producing D. Beginning with the start non-terminal of the grammar, this GP contains an alternating sequence of non-terminals Ai and index positions within rhs(Ai) to refer to a symbol, which is a non-terminal Ni

calling the next grammar rule. It ends with the index of the symbol corresponding to the selected terminal in the last grammar rule of the GP.

For example, if we apply the XPath query **Q:=//a//b[./t]** to Grammar 1, one of the selected nodes can be described by GP1:=`[A3,3,A2,4,A0:1]`. Thus, GP1 describes a rule call to rhs(A2) at position 3 in rule A3 and a rule call to rhs(A0) at position 4 in rule A2. Finally, terminal b at position 1 in rhs(A0) is selected. A more formal definition of grammar paths, however omitting rule names, is given in [5].

2.3 Executing an Update-operation for a Given Grammar Path

Now suppose that we want to execute an update operation for a single given GP. As an example consider **GP1:=`[A3,3,A2,4,A0:1]`** and update operation relabelTo(z), which replaces the label b of the selected terminal to z. Clearly, just relabeling the first terminal in rhs(A0) would be wrong, since this terminal represents four nodes in the uncompressed XML document. One possible solution to this problem was presented in [6]. The idea is to first create a copy of each grammar rule occurring in GP1. Let Ai' represent the left-hand side non-terminals of these copied rules. Then, for each sub-sequence (Ai,k,Aj) in GP1, non-terminal Aj at position k in rhs(Ai') is replaced by Aj'. Additionally, for the last sub-sequence (An:k), the update operation (for example relabelTo(z)) is executed on symbol k in rhs(An'). Finally, the start rule is replaced by the copy of the start rule. Applying this strategy to GP1, yields Grammar 2 as a result. Note that the size of this grammar is not optimal and can be further compressed.

```
A3'     → #(c(A2'(t),b(A1(b),a(A1(t),b(A2(c),ε)))),ε)
A2(y1) → a(b(g(A0,t),y1),ε)        A2'(y1)  → a(b(g(A0',t),y1),ε)
A1(y1) → g(b(c(A0,y1),t),ε)
A0      → b(t,t)                          A0'  → z(t,t)
```

Grammar 2: Grammar 2 after applying relabelTo(z) to GP1=[A3,3,A2,4,A0 : 1].

2.4 The Concept of Parallel Updates on Grammars

Given an XPath query, usually a set of multiple GPs is selected. Thus, a desirable goal is to support executing updates on such a set of GPs in parallel and to keep the size of the grammar low. A first step towards a solution of this problem is to construct a prefix tree of the GPs [6]. This tree is constructed by introducing nodes with labels Ai for non-terminals Ai and directed edges with label k for sub-sequences (Ai,k,Aj) in the GPs to be updated. Furthermore, for sub-sequences (An:k), the tree-node created for An saves an entry k. The resulting graph is a tree, as equal prefixes in the grammar paths are combined, and since each grammar path begins in the start-rule. The resulting tree for the set of grammar paths selected by query **Q:=//a//b[./t]** is shown in Fig. 3(a), where edges to numbers represent entries saved in a node, i.e. positions of selected terminals.

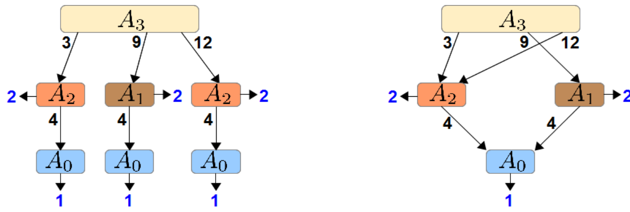

Fig. 3. a) Prefix Tree for query //a//b[./t] on Grammar 1, **b)** Corresponding Update DAG

The updates are now executed by walking top-down through the tree. Intuitively, the grammar rules on each tree-branch are isolated from the original grammar and then updated. That is, for each node of the tree visited, the corresponding grammar rule is copied. Let (Na,Nb) be an edge with label k and let label(Na)=Ai and label(Nb)=Aj respectively. With Ai' and Aj' being the non-terminals of the copied grammar rules, the symbol at position k in the grammar rule of Ai' is replaced by non-terminal Aj'. Finally, for an entry k saved in node Ni, the update is applied to the k-th symbol in rhs(Ai').

Although this approach works correctly, it induces a large overhead, since grammar rules are unnecessarily copied. For example, there are three equal nodes having label A0 in the tree of Fig. 3(a). Thus, copying the corresponding grammar rule once would have sufficed. The same holds for nodes with label A2. Formally, two leaf nodes are equal, if they save the same entries of selected terminals and have the same label, i.e. they correspond to the same non-terminal. Two inner nodes are equal, if they additionally have an identical number of outgoing edges with equal labels pointing to (recursively) equal child nodes. This finally brings us to the concept of parallel updates as introduced in [6]. Instead of looking at each grammar path for its own, we construct the (minimal) grammar path DAG from the prefix tree by combining equal nodes. This way, not only the size of the prefix tree is reduced, but additionally, we avoid unnecessary copying of grammar rules. In the context of executing update operations, we refer to this DAG as the Update DAG. The Update DAG for the given prefix tree of Fig. 3(a) is shown in Fig. 3(b). Executing the update operation relabelTo(z) then results in the more space saving Grammar 3. For a core XPath expression P, our approach supports the update operations P.relabelTo(z), P.deleteNodesAndTheirFirstChildSubtree() P.insertAsFirstChild(tree), and P.insertAsNextSibling(tree) on all selected nodes (More details are given in [6]).

A3' → #(c(A2'(t),b(A1(b),a(A1'(t),b(A2'(c),ε)))),ε)

~~A2(y1) → a(b(g(A0,t),y1),ε)~~ **A2'(y1) → a(z(g(A0',t),y1),ε)**

A1(y1) → g(b(c(A0,y1),t),ε) **A1'(y1) → g(z(c(A0',y1),t),ε)**

A0 → b(t,t) **A0' → z(t,t)**

Grammar 3: Grammar 2 after applying relabelTo(z) based on the Update DAG of Fig.3(b).

3 Construction of the Update DAG

3.1 Assumptions and Problem Definition

Let Q be an XPath query, O be an update operation, and G a straight-line linear grammar representing an XML document D. In the following, we assume that Q is an absolute query corresponding to *Core XPath* [7]. To simplify the explanations, we only consider non-nested relative filters excluding boolean operators. However, note that our software prototype obeys the complete Core XPath specification. Given these assumptions, the aim is to evaluate query Q on grammar G yielding the Update DAG to allow the execution of parallel updates.

3.2 Overview of Our Approach

Our algorithm directly computing the Update DAG consists of three main steps:

a. Given an XPath query Q, we follow the Looking Forward approach of [8], i.e., we rewrite Q in such a way that it consists of forward axes only. Additionally, we extract the filters with their current context nodes from the main path of Q.
b. Given the set of extracted filters, for each filter expression F, we construct a special bottom-up automaton to evaluate F on grammar G. As a result, for each filter expression F, we get the minimal grammar path DAG (called Filter DAG) containing all grammar paths to nodes in the document for which F is fulfilled.
c. As last step, we construct a top-down automaton for the main path of Q following the approach of [9]. To test, whether a filter is fulfilled in a node, we use the Filter DAGs constructed in Step b. The result of this step is the Update DAG.

To avoid an implicit decompression of the grammar in steps b and c, we follow and extend the idea of dynamic programming and hashing as introduced in [5].

3.3 Query Rewriting and Extraction of Filters

As a first step, we rewrite the given XPath query Q, such that it contains forward axes of the set {descendant, descendant-or-self, child, following-sibling, self} only. The example query Q=//a//b[./t] already contains forward axes only. From the rewritten query, we extract the filters from the main path, i.e., for each location step of the form ax::tst[pred] which is not part of another filter predicate itself, we extract tst[pred]. Furthermore, we keep references in the main path pointing to the extracted filters. For Q, this results in the main path **M=/descendant::a/descendant::b→F1** and the filter **F1=b[child::t]**.

3.4 Evaluation of Queries without Filters

Now let us first consider the evaluation of a query without filters. As the example query we use main path M, assuming filter F1 always evaluates to true. To evaluate

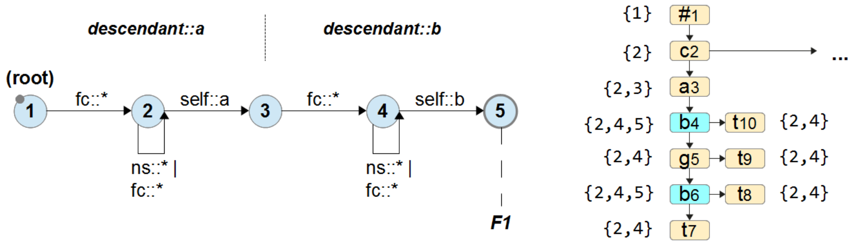

Fig. 4. a) Top-down automaton for main path M, **b)** Evaluation on the document tree of Fig. 1

the query, we extend our automaton-based top-down approach of [9] to work on grammars. It has the advantage that it is rather simple and allows us to use dynamic programming avoiding an implicit decompression of the grammar.

Constructing the Top-Down Automaton: The automaton for the main path of the query is constructed as presented in [9]. That is, each location step ax::tst can be described by an atomic automaton having transitions accepting events of the form binAx::tst, where binAx is a binary XPath axis first-child (fc), next-sibling (ns) or self. The main path then is the concatenation of these automata, as Fig. 4 (a) shows for the example query.

Evaluation on an Uncompressed Document Tree: The evaluation of such an automaton on uncompressed document trees works as described in [9]. The basic idea is to walk top-down in pre-order through the tree and to generate corresponding first-child, next-sibling and self-events. After visiting a first-child node, before continuing to the next-sibling of the parent node, a parent-event is generated, which resets the active automaton states to the states which were active in that parent node before. For example, for a tree b(t,t), the sequence (self::b, fc::*, self::t, parent::*, ns::*, self::t) is generated. Note that self-events are fired, as long as transitions can fire. A detailed description is given in [9]. Fig. 4 (b) sketches the evaluation (represented by sets of active states) of the automaton in Fig. 4 (a) that corresponds to Q's main path.

Evaluation on Grammars: As the evaluation of this top-down automaton so far only worked on uncompressed documents, we extended it to work on grammars and to directly compute the Update DAG. Our idea is to keep the automaton unchanged, but to introduce an additional module which traverses the grammar, generates grammar events, stores and recovers automaton state-sets and forwards some events to the automaton. Table 1 gives an overview of the algorithms involved in this module. Let V be the nodes and E be the edges of the DAG, both initialized with empty sets. The evaluation starts with calling procedure evalRule() for the start-rule of the grammar. For each (recursive) call of evalRule(), a DAG-node D is created, and entries(D) later-on store the positions within the currently visited grammar rule of terminals selected by the query. Each grammar rule is traversed top-down and corresponding events are generated. This works in the same way as for the uncompressed document, but with four new events ***terminal*** (replacing event self::*), ***nonTerminal***, ***actualParameter*** and ***formalParameter*** (c.f. Table 1). E.g., consider t(A(a),y1). For this expression, event-sequence (terminal(t,1), fc::*,

nonTerminal(A,2), actualParameter, terminal(a,3), parent::*, formalParameter) is generated. Events fc::*, ns::* and parent::* are directly forwarded to the automaton. When discovering a terminal L, repeatedly an event self::L is forwarded to the automaton until no more transitions of the automaton can fire (line 17). Whenever the automaton accepts, we know that L is selected by the query, and we add L's position in the grammar rule to entries(D). Second, when a non-terminal Ak is found in the actual rule Ai, we recursively traverse the grammar rule for Ak, unless we can skip the traversal of Ak by dynamic programming as explained in the next sub-section. After processing Ak, we add an edge from

Table 1. Algorithm and events for top-down evaluation of a Grammar

```
(1)   procedure evalRule(non-terminal Nt): (DagNode node, list buffer)
(2)   {  D                = new DagNode;
(3)      label(D)         = Nt;
(4)      entries(D)      = empty set; //positions of selected terminals
(5)      actParamBuffer  = empty list;
(6)      formParamBuffer = empty list;
(7)      traverse and evaluate rhs(Nt) in pre-order and generate the
         events fc::*, ns::*, parent::*, terminal, nonterminal,
         formalParameter, actualParameter;
(8)      if (node-set V of DAG contains a node D' equal to D)  D = D';
(9)      else V = V ∪ D;
(10)     return (D,formParamBuffer);
(11)  }
(12)  event formalParameter
(13)     formParamBuffer.append(automaton.getActiveStates());
(14)  event actualParameter
(15)     automaton.setActiveStates(actParamBuffer.getAndRemoveHead());
(16)  event terminal(label L, int position)
(17)     do (automaton.fire(self::L)) while automaton.changesStates();
(18)     if (automaton.isAccepting()) entries(D).add(position);
(19)  event nonTerminal(label N, int position)
(20)     states = automaton.getActiveStates();
(21)     if (lemmaTable.contains((N,states)))         // skip calling N
(22)        (node,buffer) = lemmaTable.getValueForKey((N,states));
(23)     else                 // cannot skip calling the production of N
(24)     {  (node,buffer) = evalRule(N);
(25)        key           = (N, states);
(26)        value         = (node, buffer);
(27)        lemmaTable.put(key,value);
(28)     }
(29)     edge        = (D,node);
(30)     label(edge) = position;
(31)     E           = E ∪ edge;
(32)     actParamBuffer.prepend(buffer); //copy of buffer now list-head
```

the current DAG-node D to the DAG node returned by evalRule(Ak). Third, when a formal parameter is found, we must store the set of active states of the automaton, since we need these states later when continuing on the corresponding actual parameter of the calling grammar-rule (line 13). Intuitively, we freeze the automaton, until we later continue on the actual parameter of the calling rule. Fourth, when an actual parameter is found, we activate the state-set frozen in the automaton (line 15). We must know, which state-set we have to activate for continuing traversal on the actual parameter. Therefore, when previously processing the non-terminal of that actual parameter, we copy the state-sets to a list actParamBuffer (line 32). After the traversal of a grammar rule, procedure evalRule() checks, whether there is an equal DAG node already in set V, whereas equality is defined as in the section introducing parallel updates. Note, that this test can be done in time O(1) using hashing. As a result, evalRule() finally returns the root-node of the minimal Update DAG.

An example is shown in Fig. 5. While traversing rhs(A2), y1 is discovered and the active state-set of the automaton is stored in list formParamBuffer. After completing the traversal of rhs(A2), this state-set is returned by evalRule(A2) and prepended to actParamBuffer of evalRule(A3). Then, traversal continues at terminal t of rhs(A3), which is an actual parameter. Thus, the head of list actParamBuffer saving the state-set of the previously discovered parameter y1 is removed and activated in the automaton.

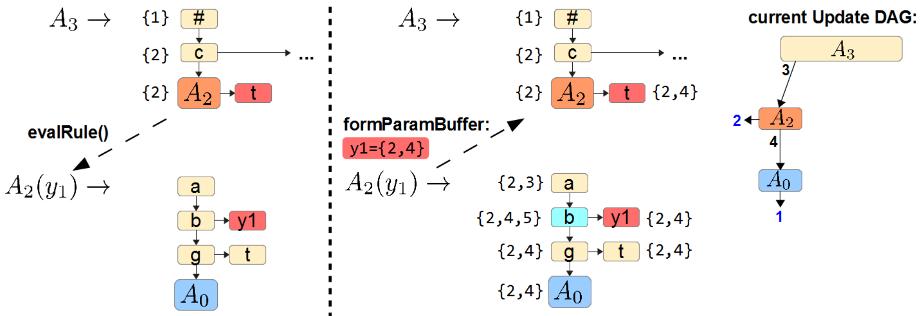

Fig. 5. a), b) Evaluation of the automaton of Fig. 4 (a) on Grammar 1

Optimization Using Dynamic Programming and Hashing: Using this approach, a problem still is that we would unnecessarily evaluate some grammar rules multiple times. To avoid this, we use dynamic programming and hashing for grammar-compressed XML documents as itwas introduced in [5] for bottom-up automaton-based evaluations. For our approach, we extended it to work for a top-down evaluation of the grammar, as well. We introduce a lemma hash-table mapping keys to values. A key is a tuple of a non-terminal label and state-set, whereas a value is a tuple storing the DAG-node which was created for that non-terminal and a list of state-sets holding one state-set for each formal parameter of that grammar rule. The observation is that when a rule was traversed before with the same set of states active in the

automaton, then the subsequent traversal of that rule must produce an equal DAG-node and equal sets of automaton states for the formal parameters. The use of the lemma table is already implemented in event nonTerminal of Table 1. However, note that in the worst case, the lemma table does never permit skipping a grammar rule. In this case, we would still implicitly traverse the whole uncompressed document. A detailed analysis for a bottom-up traversal already was given in [5]. Similar results hold for the top-down traversal, too. However, our evaluations in Section 4 show that on all tested documents and queries we are faster using dynamic programming than without using it, reaching speed-ups up to a factor of 6.7.

3.5 Evaluation of Queries Having Filters

Our example query Q=//a//b[./t] has a filter, and we decomposed Q into the main path M=/descendant::a/descendant::b →F1 and the filter F1=b[child::t]. Therefore, when using the top-down approach of the last section for evaluating M, we somehow need to know for which terminals b on which grammar paths, the filter F1 is fulfilled. In our top-down query evaluation approach of [9] on uncompressed documents, this is done by using a top-down automaton for the filter, as well. An instance of this automaton is created each time a node b is selected by M, and this instance then is evaluated in parallel during top-down evaluation. However, this approach has the disadvantage that there may be several instances of a filter automaton evaluated in parallel, i.e. one for each b-node in this case. Furthermore, as the main path of the query can have more than one filter attached to any location step, the automaton processing that main path needs to store references to filter automata instances in its states. Therefore, we decided to use another approach. Before evaluating M, we evaluate the filters using bottom-up automata resulting in a DAG for each filter. Such a DAG saves all grammar paths to nodes for which the corresponding filter is fulfilled. Thus, for F1, this DAG saves all grammar paths to b-terminals having a child t. When afterwards evaluating the main path of the query, we can use the computed DAGs to decide, for which document node which filter is fulfilled. This approach has the major advantage that the automata for the top-down traversal are kept simple and that we can extend the idea of dynamic programming to consider filters.

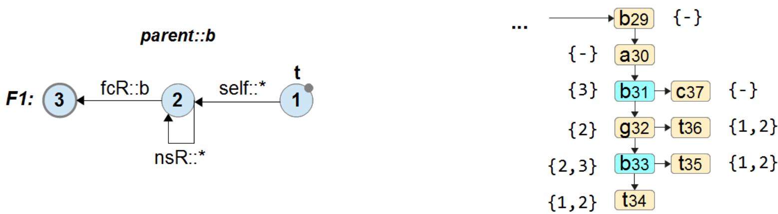

Fig. 6. a) Bottom-up automaton for F1=b[child::t], **b)** Evaluation on document tree of Fig. 1

Construction of Bottom-up Automata: To evaluate the extracted filters, we first construct a special bottom-up automaton for each filter reusing the ideas of [10]. The basic observation is that we can evaluate a location-path bottom-up from right to left, i.e., each expression tst1/ax::tst2 can be represented by the equivalent reverse

expression tst2::axR/tst1, where axR is the reverse XPath axis of ax. For F1=b[child::t], we get t::parent/b. As in the top-down approach, each location step can then be expressed as an automaton using the events fcR::*, nsR::* and self::tst, with fcR being the reverse axis of first-child, nsR the reverse axis of next-sibling, and tst being a node test. Concatenating these automata for each location step then results in an automaton evaluating the whole filter expression. Fig. 6 (a) shows the resulting automaton for filter F1.

Evaluation on an Uncompressed Document Tree: A bottom-up automaton is evaluated on a tree by traversing the tree in a reversed post-order walk. Each time, when continuing evaluation at a leaf-node, a new instance of that automaton is created. When traversing the path from a leaf up to the root of the tree, corresponding events fcR::tst and nsR::tst are generated, with tst being the name of the node reached. For a leaf-node and for each node reached during traversal with label tst, an event self::tst is generated as long as transitions can fire. Note, that a transition with a label of the form axis::* can fire for any event axis::tst. Furthermore, for an event self::tst, the source states of firing transitions stay active for the same reasons as explained in [9]. The start-state of an automaton-instance is activated whenever a document node fulfilling the node name-test attached to the start-state of that automaton is found. As an optimization, when two leaf-to-root paths share a node, the two automata instances are unified to one instance at the first common node by computing the union of the active state sets. This way, sub-paths are traversed only once. Fig. 6 (b) visualizes the evaluation of the automaton for F1 on parts of the document tree of Fig. 1. The filter corresponding to the automaton is fulfilled in a document node n if and only if the automaton accepts in that node. In Fig. 6 (b), this holds for nodes b31 and b33, since they are the only b-nodes having a child with label t.

Evaluation on Grammars: The evaluation of each filter automaton on the grammar follows the idea of the top-down evaluation of the main path of the query. That is, we begin the traversal in the start rule and recursively traverse the rules of non-terminals, we find. The only difference is that grammar rules are processed bottom-up from right to left. This has the advantage that for a non-terminal expression Aj(p1,…,pn), the actual parameters pi are visited before Aj itself. For each actual parameter, the set of active states for every filter automaton is saved. When visiting non-terminal Aj afterwards, the traversal continues in rhs(Aj), which is processed in the same way bottom-up. When visiting a formal parameter yk, the state sets saved for actual parameter pk are activated in the automata. After finishing the traversal of rhs(Aj), processing continues in the calling grammar rule. A sketch of this is shown in Fig. 7 (a). Note that actual parameter t is visited before calling rhs(A2) and that the automaton states are transferred to y1 in rhs(A2). The Filter DAG is constructed in the same way as with the top-down approach. The (minimal) Filter DAG for filter F1 is shown in Fig. 7 (b).

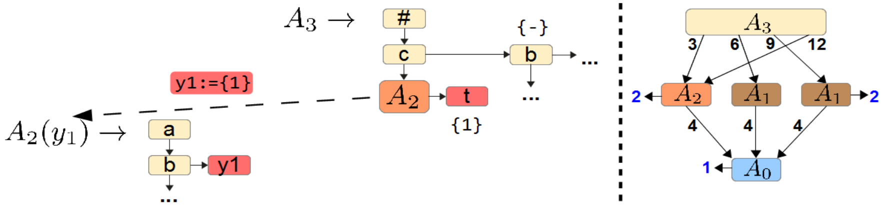

Fig. 7. a) Evaluation of F1 on Grammar 1, **b)** Resulting minimal Filter DAG

Optimization Using Dynamic Programming and Hashing: As for the top-down approach, we reuse the idea for the bottom-up evaluation of filters, as introduced in [5]. Again, we use a lemma hash-table mapping keys to values (one table for each filter). Each entry describes a rule-call from a non-terminal N occurring somewhere in the right-hand side of a production. A key is a tuple consisting of the non-terminal name of N and a list of state-sets. Each state-set at position i in the list describes the set of states which were active in the i-th actual parameter of N. The value is a tuple consisting of the DAG-node generated for traversing rhs(N) and the set of automaton-states, which were active after traversing that rule. The observation is that we can skip an additional traversal of rhs(N), when the automaton produces the same active state-sets for the actual parameters of the new occurrence of N. In this case, traversing the grammar rule again would produce the same DAG-node and the same active state-set.

As an example consider filter F1=b[child::t] evaluated on Grammar 1. Obviously, grammar rule rhs(A0) is traversed four times without dynamic programming. However, non-terminal A0 has no parameters, which means that evaluating the filter automaton on rhs(A0) always produces the same result. Thus, it is sufficient to traverse rhs(A0) only once. Furthermore, consider the grammar rule of A2. It is called two times from rhs(A3) by expressions A2(t) and A2(c). During the bottom-up evaluation of rhs(A3), the actual parameters t and c are visited before their non-terminal A2. Evaluating the automaton on terminals t and c yields state-sets {1,2} and {-} respectively. But this means, when processing rhs(A2) the automaton might produce different results, i.e. accept in different terminals and end-up in different active states after evaluating rhs(A2). This is, because the state sets computed in the actual parameters are used when visiting the formal parameter y1 in rhs(A2). Thus, we must not skip a second traversal through rhs(A2) here.

Note that we evaluate all filters of the query in parallel. Skipping the traversal of a grammar rule therefore is only allowed, if all lemma tables permit skipping. If only some of the lemma tables allow for skipping, we pause the automata of the corresponding filters such that only the automata which need to traverse that rule again are active.

Using the Filter DAGs during Evaluation: Now, having a Filter DAG for each filter, we must extend the top-down approach to use these DAGs. The general idea is to synchronously walk through the Filter DAGs while walking through the grammar. I.e., for the first call of evalRule(), we start in the root-nodes of the Filter DAGs. Then, for each recursive call to evalRule(), we follow the corresponding edges in the

Filter DAGs. This way, the top-down automaton can easily test, whether a filter is fulfilled in a currently visited terminal at position i, by checking whether the currently visited Filter DAG node of the corresponding filter stores an entry i. However, we also have to care about actual rule parameters for the following reason. Suppose, we have expressions N(b) and N(t) in a right-hand side of a grammar rule. Since we have filters and the actual parameters differ, different terminals may be selected in both rule-calls of rhs(N). Thus, we need to know, whether the filters for both calls of rhs(N) evaluate to true at the same positions. Exactly for this situation, we use the Filter DAGs. In a (minimal) Filter DAG, equal nodes have been combined to a single node. But this means, (only) when for both calls of rhs(N), in each Filter DAG, we are in the same Filter DAG node, the filters evaluate to true at the same positions. In this case, we can safely skip a second traversal of rhs(N). Thus, the decision to skip a traversal of a grammar rule also depends on the DAG-nodes currently visited and which were visited at the previous traversal of that rule. Therefore, we extend our lemma table of the top-down approach. A key-tuple additionally saves for each Filter DAG, the node which was active in that Filter DAG.

4 Evaluation

All tests were performed on an Intel Core2 Duo CPU P8800 @ 2.66 GHz. We used Java 1.7 (64 bit) with 2500 MB heap space assigned. As the first test document, we chose XMark (XM) which was generated using a scaling factor of 1.0 [11]. The second document is SwissProt (SP). To make our evaluations independent of the kind of text compression used, we removed all text- and attribute-nodes, resulting in documents of sizes 25.6 MB (XM) and 43.1 MB (SP), respectively. These documents were used as input for the query processors *QizX* [12] and *MXQuery* [13]. Furthermore, we used CluX to compress the documents, yielding grammars of size 1.15 MB (XM) and 1.74 MB (SP), respectively. These grammars were used as input to our algorithm using dynamic programming (*directUD*) and without using dynamic programming (*directUD no*). For a better comparison of the results, all documents were read from a RAM-disc. As running time, we measured the time, the algorithms spent in user mode on the CPU.

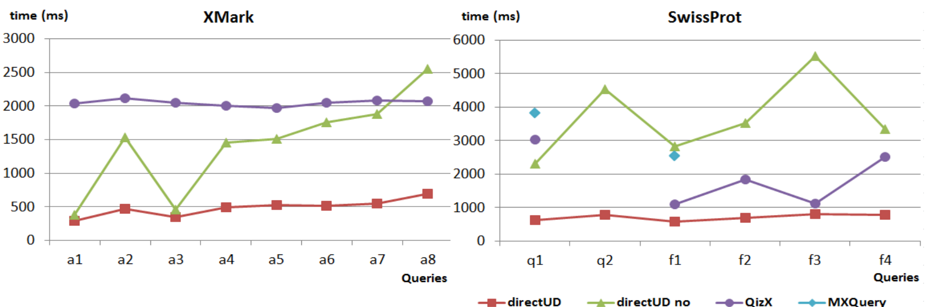

Fig. 8. a) Evaluation results on XMark document, **b)** on SwissProt document

Fig. 8 shows the evaluation results for both documents. The queries a1 to a8 executed on XM correspond to the XPath-A queries of the XPathMark benchmark [14]. For SP, queries q1 and f1 are designed such that they consist of few child axes only, whereas the other ones are more complex, having several following-sibling and descendant axes. Note that MXQuery currently does not support following-sibling axes and therefore was not executed on queries q2, f2 and f3. On the selected nodes by each query, a delete update operation was executed, removing these nodes including their first-child sub-trees. The times measured include both, the evaluation of an XPath query and the execution time of the update operation. Results not shown in the diagram were worse than 3 seconds for XM or worse than 6 seconds for SP, respectively. As both, Fig. 8 (a) and (b), show, our new approach outperforms QizX and MXQuery on each query. For the XMark document, we are about 4.2 times faster than QizX on average. However, when disabling dynamic programming, results get worse, such that QizX was faster than our algorithm for query a8. It has to be noted that query a8 has filters, such that our approach needs two runs through the grammar. Disabling dynamic programming results in implicitly decompressing that grammar twice. In this sense, our results show the benefit of using dynamic programming, being 3 times faster on average on the XMark document, when enabling it. In case of the SwissProt document, we benefit even more from dynamic programming, being up to 6.7 times faster when enabling it. Note that QizX was aborted after 60 seconds running on the rather complex query q2 having a rather high selectivity of 76,573 nodes, whereas our algorithm took less than one second.

5 Related Work

There are several approaches to XML structure compression which can be mainly divided into the categories: encoding-based, schema-based or grammar-based compressors. ***Encoding-based*** compressors (e.g.[15], [16], [17], XMill [18], XPRESS [19], and XGrind [20]) allow for a faster compression speed than the other ones, as only local data has to be considered in the compression as opposed to considering different sub-trees as in grammar-based compressors. ***Schema-based*** compressors (e.g. XCQ [21], Xenia [22], and XSDS [23]) subtract the given schema information from the structural information and only generate and output information not already contained in the schema information. XQzip [24] and the approaches [25] and [1] belong to ***grammar-based*** compression. They compress the data structure of an XML document by combining identical sub-trees. An extension of [1] and [24] is the BPLEX algorithm [3] that not only combines identical sub-trees, but recognizes similar patterns within the XML tree, and therefore allows a higher degree of compression. The approach presented in this paper, which is an extension of [2], follows the same idea. But instead of combining similar structures bottom-up, our approach searches within a given window the most promising pair to be combined while following one of three possible clustering strategies. Furthermore, in contrast to [5] and [26], that perform updates by path isolation only sequentially, our approach allows performing updates in parallel which takes only a fraction of time.

6 Summary and Conclusions

We have presented an approach to directly support updates on grammar-compressed big XML data. Given a grammar G representing an XML document D, and given an XPath query Q selecting nodes N of D and an update operation O to be performed on all these nodes N, our approach simulates this multi-update operation on G without full decompression of G. For this purpose, it computes the set of all grammar paths through G representing the nodes selected by Q, combines these paths into a small Update DAG, and then executes O in parallel on all the paths described by the Update DAG. As an advantage over other algorithms, there is no need to decompress the document and to compress it again afterwards. Additionally, by using the Update DAG, redundant modifications within the compressed grammar can be avoided, which increases the performance and keeps the size of the compressed XML document low. To further speed-up the execution of Q when computing update positions in G, we separate the top-down evaluation of Q's main path from the bottom-up computation of Q's filters, and we use dynamic programming for both, the top-down and the bottom-up computation. As a result, our solution outperforms other update processors like QizX and MXQuery working on uncompressed XML only up to a factor of 37 and more.

References

1. Buneman, P., Grohe, M., Koch, C.: Path Queries on Compressed XML. In: VLDB 2003, Berlin, Germany (2003)
2. Böttcher, S., Hartel, R., Krislin, C.: CluX - Clustering XML Sub-trees. In : ICEIS 2010, Funchal, Madeira, Portugal (2010)
3. Busatto, G., Lohrey, M., Maneth, S.: Efficient memory representation of XML documents. In: Bierman, G., Koch, C. (eds.) DBPL 2005. LNCS, vol. 3774, pp. 199–216. Springer, Heidelberg (2005)
4. Lohrey, M., Maneth, S., Mennicke, R.: Tree Structure Compression with RePair. In: DCC 2011, Snowbird, UT, USA (2011)
5. Fisher, D., Maneth, S.: Structural Selectivity Estimation for XML Documents. In: ICDE 2007, Istanbul, Turkey (2007)
6. Bätz, A., Böttcher, S., Hartel, R.: Updates on grammar-compressed XML data. In: Fernandes, A.A.A., Gray, A.J.G., Belhajjame, K. (eds.) BNCOD 2011. LNCS, vol. 7051, pp. 154–166. Springer, Heidelberg (2011)
7. Gottlob, G., Koch, C., Pichler, R.: Efficient algorithms for processing XPath queries. ACM Trans. Database Syst. 30 (2005)
8. Olteanu, D., Meuss, H., Furche, T., Bry, F.: XPath: Looking forward. In: Chaudhri, A.B., Unland, R., Djeraba, C., Lindner, W. (eds.) EDBT 2002. LNCS, vol. 2490, pp. 109–127. Springer, Heidelberg (2002)
9. Böttcher, S., Steinmetz, R.: Evaluating xPath queries on XML data streams. In: Cooper, R., Kennedy, J. (eds.) BNCOD 2007. LNCS, vol. 4587, pp. 101–113. Springer, Heidelberg (2007)
10. Benter, M., Böttcher, S., Hartel, R.: Mixing bottom-up and top-down xPath query evaluation. In: Eder, J., Bielikova, M., Tjoa, A.M. (eds.) ADBIS 2011. LNCS, vol. 6909, pp. 27–41. Springer, Heidelberg (2011)

11. Schmidt, A., Waas, F., Kersten, M., Carey, M., Manolescu, I., Busse, R.: XMark: A Benchmark for XML Data Management. In : VLDB 2002, Hong Kong, China (2002)
12. Axyana-Software: Qizx, `http://www.axyana.com/qizx`
13. MXQuery, `http://mxquery.org`
14. Franceschet, M.: XPathMark: An XPath Benchmark for the XMark Generated Data. In: Bressan, S., Ceri, S., Hunt, E., Ives, Z.G., Bellahsène, Z., Rys, M., Unland, R. (eds.) XSym 2005. LNCS, vol. 3671, pp. 129–143. Springer, Heidelberg (2005)
15. Zhang, N., Kacholia, V., Özsu, M.: A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML. In: ICDE 2004, Boston, MA, USA (2004)
16. Cheney, J.: Compressing XML with Multiplexed Hierarchical PPM Models. In: DCC 2001, Snowbird, Utah, USA (2001)
17. Girardot, M., Sundaresan, N.: Millau: an encoding format for efficient representation and exchange of XML over the Web. Computer Networks 33 (2000)
18. Liefke, H., Suciu, D.: XMILL: An Efficient Compressor for XML Data. In: SIGMOD 2000, Dallas, Texas, USA (2000)
19. Min, J.-K., Park, M.-J., Chung, C.-W.: XPRESS: A Queriable Compression for XML Data. In: SIGMOD 2003, San Diego, California, USA (2003)
20. Tolani, P., Haritsa, J.: XGRIND: A Query-Friendly XML Compressor. In: ICDE 2002, San Jose, CA (2002)
21. Ng, W., Lam, W., Wood, P., Levene, M.: XCQ: A queriable XML compression system. Knowl. Inf. Syst. (2006)
22. Werner, C., Buschmann, C., Brandt, Y., Fischer, S.: Compressing SOAP Messages by using Pushdown Automata. In: ICWS 2006, Chicago, Illinois, USA (2006)
23. Böttcher, S., Hartel, R., Messinger, C.: XML Stream Data Reduction by Shared KST Signatures. In: HICSS-42 2009, Waikoloa, Big Island, HI, USA (2009)
24. Cheng, J., Ng, W.: XQzip: Querying compressed XML using structural indexing. In: Bertino, E., Christodoulakis, S., Plexousakis, D., Christophides, V., Koubarakis, M., Böhm, K. (eds.) EDBT 2004. LNCS, vol. 2992, pp. 219–236. Springer, Heidelberg (2004)
25. Adiego, J., Navarro, G., Fuente, P.: Lempel-Ziv Compression of Structured Text. In: DCC 2004, Snowbird, UT, USA (2004)
26. Fisher, D., Maneth, S.: Selectivity Estimation. Patent WO 2007/134407 A1 (May 2007)