View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by DCU Online Research Access Service

SCOOTER: A Compact and Scalable Dynamic Labeling Scheme for XML Updates.

Martin F. O'Connor and Mark Roantree

Interoperable Systems Group, School of Computing,
Dublin City University, Dublin 9, Ireland
{moconnor,mark}@computing.dcu.ie

Abstract. Although dynamic labeling schemes for XML have been the focus of recent research activity, there are significant challenges still to be overcome. In particular, though there are labeling schemes that ensure a compact label representation when creating an XML document, when the document is subject to repeated and arbitrary deletions and insertions, the labels grow rapidly and consequently have a significant impact on query and update performance. We review the outstanding issues to-date and in this paper we propose SCOOTER - a new dynamic labeling scheme for XML. The new labeling scheme can completely avoid relabeling existing labels. In particular, SCOOTER can handle frequently skewed insertions gracefully. Theoretical analysis and experimental results confirm the scalability, compact representation, efficient growth rate and performance of SCOOTER in comparison to existing dynamic labeling schemes.

1 Introduction

At present, most modern databases providers support the storage and querying of XML documents. They also support the updating of XML data at the document level, but provide limited and inefficient support for the more fine-grained (node-based) updates within XML documents. The XML technology stack models an XML document as a tree and the functionality provided by a tree labeling scheme is key in the provision of an efficient and effective update solution. In particular, throughout the lifecycle of an XML document there may be an arbitrary number of node insertions and deletions. In our previous work, we proposed a labeling scheme that fully supported the reuse of deleted node labels under arbitrary insertions. In this paper, we focus on the more pressing problem that affects almost all dynamic labeling schemes to-date, the linear growth rate of the node label size under arbitrary insertions, whether they are single insertions, bulk insertions, or frequently skewed insertions.

1.1 Motivation

The length of a node label is an important criterion in the quality of any dynamic labeling scheme [2] and the larger the label size, the more significant is

the negative impact on query and update performance [6]. In the day-to-day management of databases and document repositories, it is a common experience that more information is inserted over time than deleted. Indeed in the context of XML documents and repositories, a common insertion operation is to append new nodes into an existing document (e.g.: heart rate readings in a sensor databases, transaction logging in financial databases) or to perform bulk insertions of nodes at a particular point in a document. These insertion operations are classified as frequently skewed insertions [20]. Over time, such insertions can quickly lead to large label sizes and consequently impact negatively on query and update performance. Furthermore, large label sizes lead to larger storage costs and more expensive IO costs. Finally, large node labels require higher computational processing costs in order to determine the structural relationships (ancestor-descendant, parent child and sibling-order) between node labels. Our objectives are to minimize the update cost of inserting new nodes and while minimizing the label growth rate under any arbitrary combination of node insertions and deletions.

1.2 Contribution

In this paper, we propose a new dynamic labeling scheme for XML called SCOOTER. The name encapsulates the core properties - Scalable, Compact, Ordered, Orthogonal, Trinary Encoded Reusable dynamic labeling scheme. The principle design goal underpinning SCOOTER is to avoid and bypass the congestion and bottleneck caused by large labels when performing fine-grained node-based updates of XML documents. SCOOTER is scalable insofar as it can support an arbitrary number of node label insertions and deletions while completely avoiding the need to relabel nodes. SCOOTER provides compact label sizes by constraining the label size growth rate under various insertions scenarios. Order is maintained between nodes at all times by way of lexicographical comparison. SCOOTER is orthogonal to the encoding technique employed to determine structural relationships between node labels. Specifically, SCOOTER can be deployed using a prefix-based encoding, containment-based encoding or a prime number based encoding. SCOOTER employs the quaternary bit-encoding presented in [9] and uses the ternary base to encode node label values. SCOOTER supports the reuse of shorter deleted node labels when available.

This paper is structured as follows: in §2, we review the state-of-the-art in dynamic labeling schemes for XML, with a particular focus on scalability and compactness. In §3, we present our new dynamic labeling scheme SCOOTER and the properties that underpin it. We describe how node labels are initially assigned and we analyse the growth rate of the label size. In §4, we present our insertion algorithms and our novel compact adaptive growth mechanism which ensures that label sizes grow gracefully and remain compact even in the presence of frequently skewed node insertions. We provide experimental validation for our approach in terms of execution time and total label storage costs by comparing SCOOTER to three dynamic label schemes that provide similar functionality and present the results in §5. Finally in §6, our conclusions are presented.

2 Related Research

There are several surveys that provide an overview and analysis of the principle dynamic labeling schemes for XML proposed to date [15], [4], [16], [12]. In reviewing the state-of-the-art in dynamic labeling schemes for XML, we will consider each labeling scheme in its ability to support the following two core properties: scalability and compactness. By scalable, we mean the labeling scheme can support an arbitrary number of node insertions and deletions while completely avoiding the need to relabel nodes. As the volume of data increases and the size of databases grow from Gigabytes to Terabytes and beyond, the computational cost of relabeling node labels and rebuilding the corresponding indices becomes prohibitive. By compact, we mean the labeling scheme can ensure the label size will have a highly constrained growth rate both at initial document creation and after subsequent and repeated node insertions and deletions. Indeed almost all dynamic labeling schemes to-date [14], [9], [2], [3], [8], [7], [13], [1] are compact only when assigning labels at document creation, but have a linear growth rate for subsequent node label insertions which quickly lead to large labels.

To the best of our knowledge, there is only one published dynamic labeling scheme for XML that is scalable, and offers compact labels at the document initialisation stage, namely the QED labeling scheme [9] (subsequently renamed CDQS in [10]). The Quaternary encoding technique presented in [9] overcomes the limitations present in all other binary encoding approaches. A comprehensive review of the various binary encoding approaches for node labels and their corresponding advantages and limitations is provided in [10]. We now briefly summarise their findings and supplement them with our own analysis.

All node labels must be stored as binary numbers at implementation, which are in turn stored as either fixed length or variable length. It should be clear that all fixed length labels are subject to overflow (and are hence, not scalable) once all the assigned bits have been consumed by the update process and consequently require the relabeling of all existing labels. The problem may be temporarily alleviated by reserving labels for future insertions, as in [11] but this leads to increased storage costs and only delays the relabeling process. There are three types of variable encoding: variable length encoding, multi-byte encoding and bit-string encoding. The V-CDBS labeling scheme [10] employs a variable length encoding. Variable length encodings require the size of the label to be stored in addition to the label itself. Thus, when many nodes are inserted into the XML tree, at some point the original fixed length of bits assigned to store the size of the label will be too small and overflow, requiring all existing labels to be relabeled. This problem has been called the overflow problem in [9]. The Ordpath [14], QRS [1], LSDX [3] and DeweyID [17] labeling schemes all suffer from the overflow problem. In addition, Ordpath is not compact because it only uses odd numbers when assigning labels during document initialisation.

UTF-8 [21] is an example of a multi-byte variable encoding. However UTF-8 encoding is not as compact as the Quaternary encoding. Furthermore, UTF-8 can only encode up to 2^{31} labels and thus, cannot scale beyond 2^{32}. The vector order labeling scheme [19] stores labels using a UTF-8 encoding. Cohen et al.

[2] and EBSL [13] use bitstrings to represent node labels. Although they do not suffer from the overflow problem they have a linear growth rate in label size when assigning labels during document initialisation and also when generating labels during frequently skewed insertions - consequently they are not compact. The Prime number labeling scheme [18] also avoids node relabeling by using simultaneous congruence (SC) values to determine node order. However, order-sensitive updates are only possible by recalculating the SC values based on the new ordering of nodes and in [9], they determined the recalculation costs to be highly prohibitive.

Lastly, we consider the problem of frequently skewed insertions. To the best of our knowledge, there are only two published dynamic labeling schemes that directly address this problem. The first is the vector order labeling scheme [19] which as mentioned previously uses a UTF-8 encoding for node labels which does not scale. We will detail in our experiments section that for a relatively small number of frequently skewed insertions that are repeated a number of times, the vector order labels can quickly grow beyond the valid range permitted by UTF-8 encoding. The second labeling scheme is Dynamic Dewey Encoding (DDE) [20] which has the stated goal of supporting both static and dynamic XML documents. However, although the authors indicate their desire to avoid a dynamic binary encoding representation for their labels due to the overhead of encoding and decoding labels, they do not state the binary representation they employ to store or represent labels and thus, we are not in a position to evaluate their work.

In summary, there does not currently exist a dynamic labeling scheme for XML that is scalable, can completely avoid relabeling node labels and is compact at both document initialisation and during arbitrary node insertions and deletions, including frequently skewed insertions.

3 The SCOOTER Labeling Scheme

In this section, we introduce our new dynamic labeling scheme for XML called SCOOTER, describe how labels are initially assigned at document creation and then, highlight the unique characteristics of our labeling scheme.

SCOOTER adopts the quaternary encoding presented in [9] so we now briefly introduce quaternary codes. A Quaternary code consists of four numbers, “0”, “1”, “2”, “3”, and each number is stored with two bits, i.e.: “00”, “01”, “10”, “11”. A SCOOTER code is a quaternary code such that the number “0”, is reserved as a separator and only “1”, “2”, “3” are used in the SCOOTER code itself. The SCOOTER labeling scheme inherits many of the properties of the QED labeling scheme, such as being orthogonal to the structural encoding technique employed to represent node order and to determine relationships between nodes. Specifically, node order is based on lexicographical order rather than numerical order and SCOOTER may be deployed as a prefix-based or containment-based labeling scheme. The containment based labeling schemes exploit the properties of tree traversal to maintain document order and to determine the various

```
     input  : k - the kth node to be labelled.
     input  : childCount
     input  : base - the base to encode in.
     output: label - the label of node k.
 1   digits = ⌈log_base (childCount + 1)⌉
 2   divisor = base^digits
 3   quotient = k
 4   label = null
 5   while (i=1; i < digits; i++) do
 6   |   divisor ⟵ divisor / base
 7   |   code ⟵ ⌊ quotient / divisor ⌋ + 1
 8   |   label ⟵ label ⊕ code
 9   |   remainder ⟵ quotient % divisor
10   |   if remainder == 0 then
11   |   |   return label
12   |   else
13   |   |   quotient ⟵ remainder
14   |   end
15   end
16   return label ⟵ label ⊕ (quotient + 1)
```

(a) Function AssignNodeK.

Decimal	SCOOTER 2 digits	SCOOTER 3 digits	QED
1	12	112	112
2	13	113	12
3	2	12	122
4	22	122	123
5	23	123	13
6	3	13	132
7	32	132	2
8	33	133	212
9		2	22
10		212	222
11		213	223
12		22	23
13		222	232
14		223	3
15		23	312
16		232	32
17		233	322
18		3	323
19		312	33
20		313	332
Total Size		104	100

(b) SCOOTER & QED labels

Fig. 1

structural relationships between nodes. Two SCOOTER codes are generated to represent the start and end intervals for each node in a containment-based scheme. In a prefix-based labeling scheme, the label of a node in the XML tree consists of the parent's label concatenated with a delimiter (separator) and a positional identifier of the node itself. The positional identifier indicates the position of the node relative to its siblings. In the prefix approach, the SCOOTER code represents the positional identifier of a node, also referred to as the selflabel.

3.1 Assigning Labels

A SCOOTER code must end in a "2" or a "3" in order to maintain lexicographical order in the presence of dynamic insertions due to reasons outlined in [10]. For the purpose of presenting our algorithms, we shall assume a prefix-based labeling scheme in this paper. The QED labeling scheme adopts a recursive divide-and-conquer algorithm to assign initial labels at document creation [9]. The SCOOTER AssignInitialLabels algorithm presents a novel approach described in algorithm 1. The SCOOTER codes presented in Figure 1(b) are examples generated using algorithm 1. The algorithm takes as input a parent node P (the root node is labeled '2'), and *childCount* - the number of children of P. When *childCount* is expressed as an positive integer of the base three, Line 1 computes the minimum number of digits required to represent *childCount* in the

Algorithm 1: Assign Initial Labels.

```
   input  : P - a parent node
   input  : childCount - the number of child nodes of P
   output: a unique SCOOTER selflabel for each child node.
1  maxLabelSize = ⌈log_3 (childCount + 1)⌉
2  selfLabel[1] = null
   /* Compute the SCOOTER selflabel of the first child.                    */
3  while (i=1; i < maxLabelSize; i++) do
4  |   selfLabel[1] ⟵ selfLabel[1] ⊕ 1 ;             // ⊕ means concatenation.
5  end
6  selfLabel[1] ⟵ selfLabel[1] ⊕ 2
   /* Now compute the SCOOTER selflabels for all remaining children.       */
7  while (i=2; i <= childCount; i++) do
8  |   selfLabel[i] ⟵ Increment (selfLabel[i - 1], maxLabelSize)
9  end
```

base three. The minimum number of digits required will determine the maximum label size generated by our AssignInitialLabels algorithm.

There are a small number of rules used to determine the assignment of labels in order to ensure a compact label size and to maintain lexicographical order between labels. The first 4 of these rules concern the first label while the remaining 2 rules, determine remaining labels.

- The first label must terminate with "2".
- It must have no other "2" other than the final digit.
- It can never have the digit "3".
- It will always be the maximum allowable length.
- The second and remaining labels can never end in "1".
- The second and remaining labels can be of any allowable length.

As previously stated, the first label will always end with a '2' symbol but as it must be of maximum length, it is preceded by a sequence of '1' symbols. The sequence of '1' symbols may be zero (empty) if the minimum number of digits required to represent *childCount* in the base three is one digit. All subsequent child labels after the first label are generated by incrementing the child label immediately preceding it. Figure 1(b) shows the sequence of labels for both 2- and 3-digit initially assigned labels. The maximum number of labels that may be assigned with D digits is 3^D - 1 labels.

The Increment algorithm (algorithm 2) takes as input a node label and the maxLabelSize and returns a new node label that is the immediate lexicographical increment of the input node. The Increment algorithm will never receive a label longer than maxLabelSize. Furthermore, the Increment algorithm will never receive a label with a length of maxLabelSize **and** consisting of all '3' symbols by virtue of line 1 in algorithm 1. Lastly, although the Increment algorithm will never receive a node label from the AssignInitialLabels algorithm that ends with a '1' symbol, we may pass substrings of labels that end with a '1' symbol to the Increment algorithm when handling dynamic node label insertions and deletions (discussed in the next section).

Algorithm 2: Increment

```
   input  : N_left - a node label; maxLabelSize - maximum number of symbols allowed in label.
   output: N_new - a new self_label such that N_left ≺ N_new
 1 N_temp ⟵ N_left
 2 if Length(N_temp) == maxLabelSize then
 3 |   if Last symbol in N_temp is '1' then
 4 |   |   N_new ⟵ N_temp with last symbol changed to '2'
 5 |   else if (Last symbol in N_temp is '2' then
 6 |   |   N_new ⟵ N_temp with last symbol changed to '3'
 7 |   else if (Last symbol in N_temp is '3' then
 8 |   |   while last symbol of N_temp is '3' do
 9 |   |   |   N_temp ⟵ N_temp with last symbol removed.
10 |   |   end
11 |   |   if Last symbol in N_temp is '1' then
12 |   |   |   N_new ⟵ N_temp with last symbol changed to '2'
13 |   |   else if (Last symbol in N_temp is '2' then
14 |   |   |   N_new ⟵ N_temp with last symbol changed to '3'
15 |   |   end
16 |   end
17 else if Length(N_temp) < maxLabelSize then
18 |   while (i = Length(N_temp) + 1; i < maxLabelSize; i++) do
19 |   |   N_temp ⟵ N_temp ⊕ 1
20 |   end
21 |   N_new ⟵ N_temp ⊕ 2
22 end
23 return N_new
```

SCOOTER's AssignInitialLabels algorithm has three distinct properties which make it quite different from the QED labeling scheme.

1. Firstly, each SCOOTER label can be determined solely based on the label of the node to the immediate left (and immediate right but we omit the Decrement algorithm due to lack of space). This is a key property which we will exploit to enable and maintain compact node labels in the presence of an arbitrary number of node insertions and deletions. This property also facilitates the reuse of deleted node labels. In contrast, the QED encoding algorithm employs the mathematical `round` function which introduces an approximation function into the QED assign initial labels process. In order words, the label value of node n is not and cannot be determined solely from the label values of node n+1 or node n-1. The QED encoding algorithm can guarantee lexicographical order but cannot guarantee the accurate calculation of the size of a node label n or indeed the label n itself, based solely on the node labels immediately adjacent to node n. It follows that when node n is deleted, the QED labeling scheme cannot guarantee the accurate calculation of the deleted node label n (and its size), and consequently cannot guarantee that the deleted node label n will be reused.
2. One significant limitation arising from the sequential determination of initially assigned node labels is that to generate a label for node n, we must first generate all n-1 node labels. This limitation can be a significant bottleneck when processing very large XML files. Hence, the second distinct property: the SCOOTER initially assigned labels may also be computed independent of each other as illustrated in function `AssignNodeK` in Figure 1(a). Specif-

ically, given n child nodes to be labeled, the function `AssignNodeK` can determine an arbitrary k^{th} child node label without having to compute any other child node label. When parsing very large XML documents, the ability to compute node labels independent of one another opens up the possibility of parallel processing in a multi-threaded and multi-core environment and may offer sizeable gains in computation time.

3. The third distinct property: the SCOOTER label encoding mechanism is independent of the underlying numeric base, as illustrated by function `AssignNodeK` in Figure 1(a). Our SCOOTER dynamic labeling scheme and compact adaptive growth mechanism may be applied and implemented using any numeric base greater than or equal to two. In mathematical numeral systems, the base or radix is the number of unique symbols that a positional numeral system uses to represent numbers. For example, the decimal system uses the *base 10*, because it uses the 10 symbols from 0 through 9. The highest symbol usually has the value of one less than the base of that numeric system. In [5], the authors demonstrate the most economical radix for a numbering system is e, the base of the natural logarithms, with a value of approximately 2.718. Economy is measured as the product of the radix and the number of digits needed to express a range of given values. Consequently the economy is also a measure of how compact is the numerical representation of a given radix. In [5], the authors also demonstrate that the integer 3, being the closest integer to e, is almost always the most economical integer radix or base. For this reason, in this paper, we have chosen to use the numeric *base 3* and consequently quaternary codes to represent SCOOTER labels.

4 Compact Adaptive Growth Mechanism

In this section, we present our novel compact adaptive growth mechanism and related node label insertion algorithms which ensure a highly constrained label growth rate irrespective of the quantity of arbitrary and repeated node label insertions and deletions. We will begin with a simple example to provide an overview of the conceptual approach followed by a more detailed analysis of the underlying properties.

Consider an XML tree consisting of a root node R and two child nodes with selflabels '2' and '3'. We insert a sequence of 100 nodes to the right of the rightmost child node. Table 1 in Figure 2(a) illustrates the first 18 insertions. The first two ($3^1 - 1$) node labels generated consist of a *prefix* string '3' and a *postfix* string that mirrors the labels normally generated for a maxLabelSize of 1 digit (i.e.: '2' and '3'). For the next 8 ($3^2 - 1$) insertions, from the third to the tenth insertion inclusive, the newly generated labels consist of a *prefix* string '33' and a *postfix* string that mirrors the labels generated for a maxLabelSize of 2 digits (e.g.: '12', '13', '2' and so on). For the next 26 ($3^3 - 1$) insertions, from the 11th to the 36th insertion inclusive, labels consist of a *prefix* string '3333' and a *postfix* string that mirrors the labels generated for a maxLabelSize of 3

Insert after rightmost node	SCOOTER Label
	2
	3
1	32
2	33
3	3312
4	3313
5	332
6	3322
7	3323
8	333
9	3332
10	3333
11	3333112
12	3333113
13	333312
14	3333122
15	3333123
16	333313
17	3333132
18	3333133

(a) Table 1.

Range Start	Range End	Node Start	Node End	Prefix Length	Postfix Length	Maximum SCOOTER SelfLabel Length	SCOOTER SelfLabel Total bits
3^0	3^1-1	1	2	1	1	2	4
3^1	3^2-1	3	10	2	2	4	8
3^2	3^3-1	11	36	4	3	7	14
3^3	3^4-1	37	116	7	4	11	22
3^4	3^5-1	117	358	11	5	16	32
3^5	3^6-1	359	1,086	16	6	22	44
3^6	3^7-1	1,087	3,272	22	7	29	58
3^7	3^8-1	3,273	9,832	29	8	37	74
3^8	3^9-1	9,833	29,514	37	9	46	92
3^9	$3^{10}-1$	29,515	88,562	46	10	56	112
3^{10}	$3^{11}-1$	88,563	265,708	56	11	67	134
3^{11}	$3^{12}-1$	265,709	797,148	67	12	79	158
3^{12}	$3^{13}-1$	797,149	2,391,470	79	13	92	184
3^{13}	$3^{14}-1$	2,391,471	7,174,438	92	14	106	212
3^{14}	$3^{15}-1$	7,174,439	21,523,344	106	15	121	242
3^{15}	$3^{16}-1$	21,523,345	64,570,064	121	16	137	274
3^{16}	$3^{17}-1$	64,570,065	193,710,226	137	17	154	308
3^{17}	$3^{18}-1$	193,710,227	581,130,714	154	18	172	344
3^{18}	$3^{19}-1$	581,130,715	1,743,392,180	172	19	191	382
3^{19}	$3^{20}-1$	1,743,392,181	5,230,176,580	191	20	211	422

(b) Table 2. Label Insertion - Compact Adaptive Growth Rate

Fig. 2

digits (e.g.: '112', '113', '12' and so on). This process is repeated as many times as required.

We now provide an analysis of the underlying properties. Conceptually, we consider a label as comprising of two components: a prefix and a postfix. We define the smallest permissible prefix length to be 1 symbol. When the prefix has length 1, we define the maximum postfix length permissible to be 1 also (please refer to Table 2 in Figure 2(b)). The maximum label length will always equal the sum of the prefix length and the maximum postfix length.

- When inserting a new rightmost node label, we extend the length of the prefix *if and only if* the current rightmost label consists of all '3' symbols *and* the length of the current rightmost node label equals the sum of the current prefix length and maximum postfix length. For example, given the current rightmost node label '33' with a prefix length of 1 and a maximum postfix length of 1; in order to insert a new node after node '33', we must extend the prefix and postfix lengths.
- The rule governing the adaptive growth rate of the prefix and postfix lengths is simple: the new prefix length is assigned the value of the previous maximum label length; the new maximum postfix length is assigned the value of the previous maximum postfix length plus 1. This rule is codified in lines 6–13 in algorithm 3.

Algorithm 3: Insert New Node After Rightmost Node.

```
    input  : left self_label N_left, N_left is not empty.
    output: New self_label N_new such that N_left ≺ N_new
1   if first symbol in N_left is '1' then
2   |  N_new ⟵ '2'
3   else if first symbol in N_left is '2' then
4   |  N_new ⟵ '3'
5   else if first symbol in N_left is '3' then
6   |  numConsecThrees ⟵ the number of consecutive '3' symbols at start of N_left
7   |  prefixLength ⟵ postfixLength ⟵ 1
8   |  labelLength ⟵ prefixLength + postfixLength
    |  /* Compute the prefixLength and postfixLength based on numConsecThrees.          */
9   |  while labelLength <= numConsecThrees do
10  |  |  prefixLength ⟵ prefixLength + postfixLength
11  |  |  postfixLength ⟵ postfixLength + 1
12  |  |  labelLength ⟵ prefixLength + postfixLength
13  |  end
14  |  postfix ⟵ substring(N_left, 1 + prefixLength, length(N_left))
15  |  if postfix is not empty then
    |  |  /* An arbitrary number of nodes may have been deleted, thus the label may be
    |  |     longer than the postfixLength. If it is longer, trim it.                  */
16  |  |  postfix ⟵ substring(postfix, 1, postfixLength)
17  |  |  if last symbol in postfix is '1' then
18  |  |  |  postfix ⟵ postfix with last symbol changed to '2'
19  |  |  else
20  |  |  |  postfix ⟵ Increment (postfix, postfixLength)
21  |  |  end
22  |  else if postfix is empty then
23  |  |  while i=1; i < postfixLength; i++ do
24  |  |  |  postfix ⟵ postfix ⊕ 1
25  |  |  end
26  |  |  postfix ⟵ postfix ⊕ 2
27  |  end
28  |  prefix = null
29  |  while i=1; i <= prefixLength; i++ do
30  |  |  prefix ⟵ prefix ⊕ 3
31  |  end
32  |  N_new ⟵ prefix ⊕ postfix
33  end
34  return N_new
```

All bit-string dynamic labeling schemes (including QED) have a label growth rate of one bit per node insertion. Therefore, after one thousand insertions and one million insertions, the largest selflabel sizes are 1,000 and 1,000,000 bits respectively. In contrast, after one thousand insertions and one million insertions, the largest SCOOTER selflabels are 44 bits and 184 bits respectively. Thus, SCOOTER labels may be several orders of magnitude smaller than the labels of all existing bit-string labeling schemes when processing frequently skewed insertions. Furthermore, in contrast to all existing dynamic labeling schemes, SCOOTER generates compact labels without requiring advance knowledge of the number of nodes to be inserted. The compact adaptive growth mechanism is made possible by virtue of the deterministic property of our AssignInitialLabels algorithm. The compact adaptive growth mechanism may also be applied when inserting new nodes before the leftmost node, however in this case we count the number of consecutive '1' symbols to determine the length of the prefix.

Algorithm 4: InsertBetweenTwoNodes_LessThan.

```
input  : left self_label N_left; right self_label N_right; both labels not empty.
output: New self_label N_new such that N_left ≺ N_new ≺ N_right.
1  if length(N_left) < length(N_right) then
2      if N_left is a prefix of N_right then
3          if symbol at N_right[length(N_left)+1] is a '3' then
4              N_new ⟵ N_left ⊕ 2
5          else if symbol at N_right[length(N_left)+1] is a '2' then
6              N_new ⟵ N_left ⊕ 12
7          else
8              N_temp ⟵ N_left
9              Let P ⟵ length(N_left) + 1
10             while symbol at position P in N_right is '1' do
11                 N_temp ⟵ N_temp ⊕ 1
12                 P ⟵ P + 1
13             end
14             if symbol at N_right[P] is a '3' then
15                 N_new ⟵ N_temp ⊕ 2
16             else
17                 N_new ⟵ N_temp ⊕ 12
18             end
19         end
20     else if N_left is not a prefix of N_right then
21         Let P ⟵ first position of difference between N_left and N_right
22         if P == 1 then
23             N_new ⟵ Increment (first symbol in N_left, 1)
24         else if P > 1 then
25             N_temp ⟵ substring(N_left, 1, P - 1)
26             N_new ⟵ N_temp ⊕ Increment (symbol at position P in N_left, 1)
27         end
28     end
29 end
30 return N_new
```

4.1 Insertion between Two Consecutive Non-Empty Node Labels

The most difficult insertion scenario is between two non-empty consecutive node labels. Between any two consecutive nodes, there may have been an arbitrary number of node deletions. The ability to determine whether deletions have occurred must be determined from the information encoded in the label alone. In addition, there are 4 distinct insertion scenarios permitted when inserting a new node between two consecutive node labels:

1. The left label is a prefix string of the right label;
2. The left label is shorter than the right label but not a prefix of the right label;
3. The left label is the same length as the right label; or
4. The left label is longer than the right label

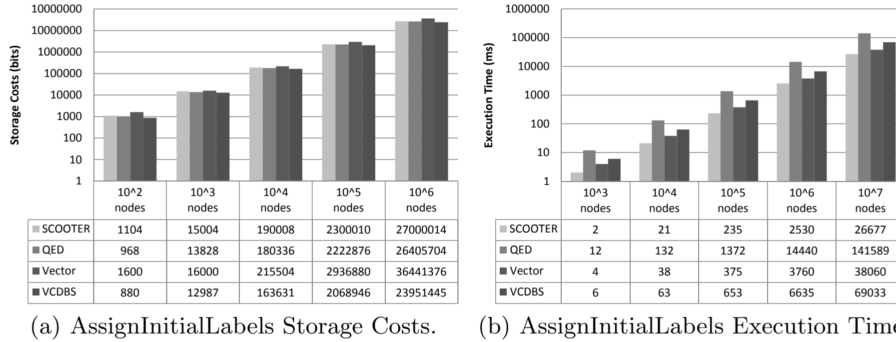
The SCOOTER labeling scheme provides the same highly constrained adaptive growth rate when processing node label insertions in all four scenarios. In the remainder of this section, we analyse the four algorithms and highlight some observations.

In algorithm 4, the InsertBetweenTwoNodes_LessThan algorithm processes the first two insertion scenarios. The new label returned will always be shorter

Algorithm 5: InsertBetweenTwoNodes_GreaterThan.

```
   input  : left self_label N_left; right self_label N_right; both labels not empty.
   output: New self_label N_new such that N_left ≺ N_new ≺ N_right.
1  if length(N_left) > length(N_right) then
2  |  Let P ⟵ first position of difference between N_left and N_right
3  |  if P < length(N_right) then
   |  |  /* If the position of difference is not the last symbol of N_right          */
4  |  |  N_temp ⟵ substring(N_left, 1, P)
5  |  |  N_new ⟵ InsertBetweenTwoNodes_LessThan (N_temp, N_right)
6  |  else if P == length(N_right) then
7  |  |  N_temp ⟵ substring(N_left, 1, P - 1)
8  |  |  if (symbol at position P in N_left is '1') and (symbol at position P in N_right is
   |  |  '3' ) then
9  |  |  |  N_new ⟵ N_temp ⊕ 2
10 |  |  else
11 |  |  |  Affix ⟵ substring(N_left, 1, P)
12 |  |  |  N_temp ⟵ substring(N_left, P + 1, length(N_left))
13 |  |  |  numConsecThrees ⟵ the number of consecutive '3' symbols at start of N_temp
14 |  |  |  if numConsecThrees == 0 then
15 |  |  |  |  postfix ⟵ Increment (first symbol of N_temp, 1)
16 |  |  |  |  N_new ⟵ Affix ⊕ postfix
17 |  |  |  else if numConsecThrees > 0 then
   |  |  |  |  /* Replace the PLACEHOLDER with lines 7 through 31 inclusive from
   |  |  |  |     Algorithm 3, substituting all references to N_left with N_temp.   */
18 |  |  |  |  PLACEHOLDER
19 |  |  |  |  N_new ⟵ Affix ⊕ prefix ⊕ postfix
20 |  |  |  end
21 |  |  end
22 |  end
23 end
24 return N_new
```

than both input labels *if and only if* a shorter deleted node label is available for reuse. By available, we mean a shorter unique and valid SCOOTER code that is lexicographically ordered between the left and right node labels. If no shorter label is available (such as when the left label is a prefix of the right label), the algorithm will still return the smallest valid label lexicographically ordered between the two input labels. When the two labels are the same size, if the position of difference between the two input labels is the last symbol in both labels, then both input labels must be lexicographical neighbours with no deleted node label available between them. Consequently a new label is generated by concatenating a '2' symbol to the end of the left input label. Otherwise, the left input label is trimmed and algorithm InsertBetweenTwoNodes_LessThan is invoked which will reuse a shorter deleted node label lexicographically ordered between the two input labels. Finally, in algorithm 5, the InsertBetweenTwoNodes_GreaterThan algorithm processes the fourth insertion scenario: the left input label is longer than the right input label. Algorithm 5 will reclaim and reuse the shortest deleted node label, if one exists. Otherwise, it will ensure that all newly generated labels will be assigned according to our compact adaptive growth rate insertion mechanism.



	10^2 nodes	10^3 nodes	10^4 nodes	10^5 nodes	10^6 nodes
SCOOTER	1104	15004	190008	2300010	27000014
QED	968	13828	180336	2222876	26405704
Vector	1600	16000	215504	2936880	36441376
VCDBS	880	12987	163631	2068946	23951445

(a) AssignInitialLabels Storage Costs.

	10^3 nodes	10^4 nodes	10^5 nodes	10^6 nodes	10^7 nodes
SCOOTER	2	21	235	2530	26677
QED	12	132	1372	14440	141589
Vector	4	38	375	3760	38060
VCDBS	6	63	653	6635	69033

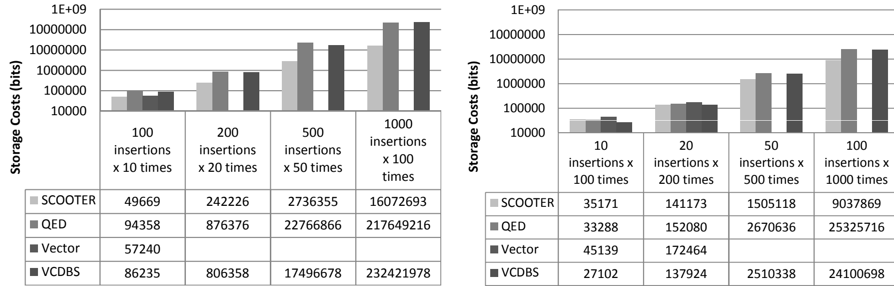
(b) AssignInitialLabels Execution Time.

Fig. 3

5 Experiments

In this section, we evaluate and compare our SCOOTER labeling scheme with three other dynamic labeling schemes, namely QED [9], Vector [19] and V-CDBS [10]. The three labeling schemes were chosen because they each offer those properties we have encapsulated in SCOOTER - scalability, compactness and the ability to process frequently skewed insertions in an efficient manner. QED is the only dynamic labeling scheme that offers a compact labeling encoding at document initialisation, while overcoming the overflow problem and completely avoiding node relabeling. The SCOOTER labeling scheme inherits these properties by virtue of the quaternary encoding. The Vector labeling scheme is the only dynamic labeling scheme that has as one of its design goals, the ability to process frequently skewed insertions efficiently. SCOOTER has also been designed with this specific property in mind. Lastly, V-CDBS is the most compact dynamic labeling scheme presented to-date, as illustrated in [10]. Although V-CDBS is subject to the overflow problem and thus, cannot avoid relabeling nodes, we want to compare SCOOTER against the most compact dynamic encoding available. All the schemes were implemented in Python and all experiments were carried out on a 2.66Ghz Intel(R) Core(TM)2 Duo CPU and 4GB of RAM. The experiments were performed 10 times and the results averaged.

In Figure 3(a), we illustrate the total label storage cost of 10^2 through 10^6 initially assigned node labels. As expected V-CDBS is the most compact, QED in second place and SCOOTER has marginally larger storage costs than QED. However, SCOOTER has the most efficient processing time, illustrated in Figure 3(b), when generating initially assigned labels, due to the efficient Increment algorithm. Although SCOOTER theoretical scales efficiently under frequently skewed insertions - a result that was validated by experimental analysis - it was necessary to evaluate SCOOTER under multiple frequently skewed insertions and in this process, revealed some interesting results. Figure 4(a) illustrates large skewed node label insertions performed at a randomly chosen position and repeated a small number of times. Figure 4(b) illustrates small skewed node



	100 insertions x 10 times	200 insertions x 20 times	500 insertions x 50 times	1000 insertions x 100 times
SCOOTER	49669	242226	2736355	16072693
QED	94358	876376	22766866	217649216
Vector	57240			
VCDBS	86235	806358	17496678	232421978

	10 insertions x 100 times	20 insertions x 200 times	50 insertions x 500 times	100 insertions x 1000 times
SCOOTER	35171	141173	1505118	9037869
QED	33288	152080	2670636	25325716
Vector	45139	172464		
VCDBS	27102	137924	2510338	24100698

(a) Large and Infrequent Skewed Random Insertions.

(b) Small and Frequent Skewed Random Insertions.

Fig. 4

label insertions performed at a randomly chosen position and repeated many times. SCOOTER initially performs best in the former case, because our adaptive growth mechanism is designed to give greater savings as the quantity of insertions increase. As the quantity or frequency of insertions scale, SCOOTER offers significant storages benefits in all cases. The vector labeling scheme is absent from some of the results because the label sizes grew beyond the storage capacity permitted by UTF-8 encoding.

6 Conclusions

Updates for XML databases and caches provide ongoing problems for both academic and industrial researchers. One of the primary issues is the labeling scheme that provides uniqueness and retrievability of nodes. In this, there are two major issues for researchers: the length of the label as it may negatively impact performance; and an efficient insertion mechanism to manage updates. In this paper, we introduced the SCOOTER labeling scheme and algorithms for assigning node labels and inserting new labels. We developed new algorithms for assigning labels and a novel highly compact adaptive insertion mechanism that compare favourably to all existing approaches. Our evaluations confirmed these findings through a series of experiments that examined both a very large number of label assignments and similarly large insertion operations.

Although SCOOTER offers compact labels at document initialisation, we are investigating the possibility of an improved AssignInitialLabels algorithm that generates deterministic labels as compact as the labels initially assigned by V-CDBS. Secondly, SCOOTER generates and maintains compact labels under frequently skewed insertions, such as appending a large number of node labels before or after any arbitrary node. However, when a large number of node labels are inserted at a fixed point, the label size grows rapidly. We are investigating a modification to our compact adaptive growth mechanism, such that label sizes will always have a highly constrained growth rate under any insertion scenario.

Lastly, we are adapting SCOOTER to work in binary (and not use the quaternary encoding) and we are investigating a new binary encoding to overcome the overflow problem that the quaternary encoding set out to address.

References

1. Amagasa, T., Yoshikawa, M., Uemura, S.: QRS: A Robust Numbering Scheme for XML Documents. In: ICDE. pp. 705–707 (2003)
2. Cohen, E., Kaplan, H., Milo, T.: Labeling Dynamic XML trees. In: PODS. pp. 271–281. ACM, New York, NY, USA (2002)
3. Duong, M., Zhang, Y.: LSDX: A New Labelling Scheme for Dynamically Updating XML Data. In: ADC. pp. 185–193 (2005)
4. Härder, T., Haustein, M.P., Mathis, C., Wagner, M.: Node Labeling Schemes for Dynamic XML Documents Reconsidered. Data Knowl. Eng. 60(1), 126–149 (2007)
5. Hayes, B.: Third Base. American Scientist 89(6), 490–494 (2001)
6. Kay, M.: Ten Reasons Why Saxon XQuery is Fast. IEEE Data Eng. Bull. 31(4), 65–74 (2008)
7. Kobayashi, K., Liang, W., Kobayashi, D., Watanabe, A., Yokota, H.: VLEI code: An Efficient Labeling Method for Handling XML Documents in an RDB. In: ICDE. pp. 386–387 (2005)
8. Li, C., Ling, T.W.: An Improved Prefix Labeling Scheme: A Binary String Approach for Dynamic Ordered XML. In: DASFAA. pp. 125–137 (2005)
9. Li, C., Ling, T.W.: QED: A Novel Quaternary Encoding to Completely Avoid Re-labeling in XML Updates. In: CIKM. pp. 501–508 (2005)
10. Li, C., Ling, T.W., Hu, M.: Efficient Updates in Dynamic XML Data: from Binary String to Quaternary String. VLDB Journal 17(3), 573–601 (2008)
11. Li, Q., Moon, B.: Indexing and Querying XML Data for Regular Path Expressions. In: VLDB. pp. 361–370 (2001)
12. O'Connor, M.F., Roantree, M.: Desirable Properties for XML Update Mechanisms. In: EDBT/ICDT Workshops (2010)
13. O'Connor, M.F., Roantree, M.: EBSL: Supporting Deleted Node Label Reuse in XML. In: XSym. pp. 73–87 (2010)
14. O'Neil, P.E., O'Neil, E.J., Pal, S., Cseri, I., Schaller, G., Westbury, N.: ORDPATHs: Insert-Friendly XML Node Labels. In: SIGMOD Conference. pp. 903–908 (2004)
15. Sans, V., Laurent, D.: Prefix based Numbering Schemes for XML: Techniques, Applications and Performances. PVLDB 1(2), 1564–1573 (2008)
16. Su-Cheng, H., Chien-Sing, L.: Node Labeling Schemes in XML Query Optimization: A Survey and Trends. IETE Technical Review 26, 88–100 (2009)
17. Tatarinov, I., Viglas, S., Beyer, K.S., Shanmugasundaram, J., Shekita, E.J., Zhang, C.: Storing and Querying Ordered XML using a Relational Database System. In: SIGMOD Conference. pp. 204–215 (2002)
18. Wu, X., Lee, M.L., Hsu, W.: A Prime Number Labeling Scheme for Dynamic Ordered XML Trees. In: ICDE. pp. 66–78 (2004)
19. Xu, L., Bao, Z., Ling, T.W.: A Dynamic Labeling Scheme Using Vectors. In: DEXA. pp. 130–140 (2007)
20. Xu, L., Ling, T.W., Wu, H., Bao, Z.: DDE: From Dewey to a Fully Dynamic XML Labeling Scheme. In: SIGMOD Conference. pp. 719–730 (2009)
21. Yergeau, F.: UTF-8, A Transformation Format of ISO 10646, Request for Comments (RFC) 3629 edn. (November 2003)