FibLSS: A Scalable Label Storage Scheme for Dynamic 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. Dynamic labeling schemes for XML updates have been the focus of significant research activity in recent years. However the label storage schemes underpinning the dynamic labeling schemes have not received as much attention. Label storage schemes specify how labels are physically encoded and stored on disk. The size of the labels and their logical representation directly influence the computational costs of processing the labels and can limit the functionality provided by the dynamic labeling scheme to an XML update service. This has significant practical implications when merging XML repositories such as clinical studies. In this paper, we provide an overview of the existing label storage schemes. We present a novel label storage scheme based on the Fibonacci sequence that can completely avoid relabeling existing nodes under dynamic insertions. Theoretical analysis and experimental results confirm the scalability and performance of the Fibonacci label storage scheme in comparison to existing approaches.

1 Introduction

There has been a noticeable increase in research activity concerning dynamic labeling schemes for XML in recent years. As the volume of XML data increases and the adoption of XML repositories in mainstream industry becomes more widespread, there is a requirement for labeling schemes that can support updates. While read-only XML repositories such as data warehouses have seen significant optimization using views and query adaptation [13], and novel approaches to multi-dimensional modeling [6] faciliate complex rollup and drill-down operations, these efforts do not tackle issues of major changes to the underlying XML documents.

A major obstacle in the provision of an XML update service is the limited functionality provided by existing dynamic labeling schemes. There are a number of desirable properties that characterize a *good* dynamic labeling scheme for XML [16], such as the ability to determine ancestor-descendant, parent-child, and sibling-order relationships between nodes from the labels alone; the generation

* The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2012) under grant agreement no. 304979.

of compact labels under arbitrary dynamic node insertions; and the ability to support the reuse of deleted node labels. In this paper, we address the problem of storing scalable binary encoded bit-string dynamic labeling schemes for XML. 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 size of the databases grow from Gigabytes to Terabytes and beyond, the computational costs of relabeling nodes and rebuilding the corresponding indices becomes prohibitive, not the mention the negative impact on query and updates services while the indices are under reconstruction.

1.1 Motivation

The In-MINDD FP7 project is funded by the European Commission to investigate means to decrease dementia risk and delay the onset of dementia by combining areas of social innovation, multi-factorial modelling and clinical expertise [8]. The project aims to quantify dementia risk and deliver personalised strategies and support to enable individuals to reduce their risk of dementia in later life. One of the main tasks is to integrate longitudinal studies such as the Maastricht Ageing Study (MAAS) [9], construct XML views, and integrate the views for various clinical studies. However, this integration process requires the threading of XML elements from one study into another, requiring many and frequent relabeling of nodes. The benefit of XML views is their highly interoperable qualities but their usage presents the problem of XML updates.

There are only two reasons that cause a dynamic labeling scheme to relabel nodes when updating XML. The first reason is that the node insertion algorithms of the dynamic labeling scheme do not permit arbitrary dynamic node insertions without relabeling. For example, when a new node is inserted into an XML tree, the DeweyID labeling scheme [21] requires the relabeling of all *following-sibling* nodes (and their descendants). The second reason that causes a dynamic labeling scheme to relabel nodes is due to the overflow problem.

The Overflow Problem. The Overflow Problem concerns the label storage scheme used to encode and store the labels on disk or any physical digital medium and affects both fixed-length and variable-length encodings. It should be clear that all fixed length label storage schemes are subject to overflow once all the assigned bits have been consumed by the update process and consequently require the relabeling of all existing nodes. It is not so obvious that variable-length encodings are also subject to the overflow problem. Variable length labels require the size of the label to be stored in addition to the label itself. Thus, if many nodes are inserted into the XML tree, then 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 nodes to be relabeled. This problem has been named the overflow problem in [11].

We hold the position that a modern dynamic labeling scheme for XML should **not** be subject to the overflow problem. All dynamic labeling schemes subject to the overflow problem must relabel existing nodes after a certain number of

updates have been performed. In our previous work [17], we highlighted that there are only two existing dynamic labeling schemes that can completely avoid the need to relabel nodes, namely QED [11] and SCOOTER [17]. All other dynamic labeling schemes for XML must relabel existing nodes after an arbitrary number of node insertions due to either limitations in the node insertions algorithms or limitations in the label storage scheme employed by the dynamic labeling scheme.

1.2 Contribution

In this paper, we provide a comprehensive review of the existing state-of-the-art in label storage schemes employed by XML dynamic labeling schemes. We present a novel label storage scheme that exploits the properties of the Fibonacci sequence to encode and decode node labels of any arbitrary size. The Fibonacci label storage scheme is scalable - it will never require a dynamic labeling scheme for XML to relabel nodes regardless of arbitrary or repeated dynamic node insertions and deletions. The Fibonacci label storage scheme offers comparable storage costs with the best existing approaches and in particular, is well suited for large data volumes. It also offers the best performance in computational processing costs compared to existing approaches. We provide both theoretical analyses and experimental evaluations to validate our approach.

This paper is structured as follows: in §2, we review and analyze the state-of-the-art in label storage schemes for XML, with a particular focus on scalability. In §3, we present the Fibonacci label storage scheme and the properties that underpin it. We present the algorithms for encoding and decoding a node label and provide a detailed explanation of the encoding transformation. In this section, we also provide a theoretical analysis of the growth rate of the Fibonacci encoded labels. In §4, we provide experimental evaluations of our approach in terms of execution time and total label storage costs and analyze the results. Finally in §5, our conclusions are presented.

2 Related Research

A key consideration for all dynamic labeling schemes for XML is how they choose to physically encode and store their labels on disk. All digital data is ultimately stored as binary, but the logical representation of the label on disk directly influences the size of the label on disk and the computational cost to encode/decode from the logical to the physical representation. In this section, we provide an overview of label storage schemes. All existing approaches to the storage of dynamic (variable-length) labels fall under four classifications: length fields, control tokens, separators and prefix-free codes. We employ the same four classifications as those presented in [7].

2.1 Length Fields

The concept underlying length fields is to store the length of the label immediately before the label itself. The naive approach is to assign a fixed-length bit code to indicate the length of the label. In a dynamic environment, after a certain number of node insertions, the label size will grow beyond the capacity indicated by the fixed-length bit code and consequently a larger fixed-length bit code will have to be assigned and all existing labels will have to be relabeled according to the new larger fixed-length bit code. One could initially assign a very large fixed-length bit code to minimize the occurrence of the relabeling process, but that would lead to significant wastage in storage for all relatively small labels. In [7], they present several different variations of variable-length bit codes to indicate the size of the label but the authors acknowledge that all of the variable-length approaches lead to either relabeling of existing nodes or involve significant wastage of storage space.

2.2 Control Tokens

The concept underlying control tokens is similar to length fields, except rather than storing the length of the label, tokens are used instead to indicate or *control* how the subsequent bit sequence is to be interpreted. We now provide a brief overview of UTF-8 [26] which is a multi-byte variable encoding that uses control tokens to indicate the size of a label.

UTF-8 is employed by the DeweyID [21] and Vector labeling schemes [24]. Originally, UTF-8 was designed to represent every character in the UNICODE character set, and to be backwardly compatible with the ASCII character set.

Referring to Table 1, any number between 0 and 127 ($2^7 - 1$) inclusive, may be represented using 1 byte. The first bit sequence in the label is the control token(s). If the first bit is the control token "0", it indicates the label length is 1 byte. If the first bit is the control token "1", then the number of bytes used to represent the label is computed by counting the number of consecutive control token "1" bits until the control token "0" bit is encountered. The first two bits of the second and subsequent bytes always consist of the bit sequence "10" as illustrated in Table 1.

Value	Byte1	Byte2	Byte3	Byte4	Byte5	Byte6
0 - ($2^7 - 1$)	0xxxxxxx					
2^7 - ($2^{11} - 1$)	110xxxxx	10xxxxxx				
2^{11} - ($2^{16} - 1$)	1110xxxx	10xxxxxx	10xxxxxx			
2^{16} - ($2^{21} - 1$)	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx		
2^{21} - ($2^{26} - 1$)	111110xx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	
2^{26} - ($2^{31} - 1$)	1111110x	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx

Table 1: UTF-8 Multi-byte Encoding using Control Tokens

Example 1. To encode the DeweyID label 1.152 in UTF-8, we first determine how many bytes each component requires, convert each component to binary and finally encode using the appropriate number of bytes. 1 is less than 127 ($2^7 - 1$), hence the UTF-8 encoding of 1 is 0 0000001. 152 is between 128 (2^7) and 2047 ($2^{11} - 1$), and 152 in binary is 10011000, hence the UTF-8 encoding of 152 requires two bytes and is 110 00010 10 011000 (the spaces are present for readability only). Finally, the full UTF8 encoding of the label 1.152 is 0 0000001 110 00010 10 011000.

The primary limitation of control tokens relate to the requirement to predefine a fixed-length step governing the growth of the labels under dynamic insertions. The fixed-length step cannot dynamically adjust to the characteristics of the XML document or the type of updates to be performed. Furthermore, from the point of view of scalability, UTF-8 cannot encode a number larger than $2^{31} - 1$.

2.3 Separators

Whereas control tokens are used to interpret and give meaning to the sequence of bits that immediately follow the token, a separator reserves a predefined bit sequence to have a particular meaning. Consequently, regardless of where the predefined bit sequence occurs, it must be interpreted as a separator. The QED [11] and SCOOTER [17] schemes are the only dynamic labeling schemes to date that employ the separator storage scheme to encode their labels. We now describe the label storage scheme employed by the QED labeling scheme and omit the SCOOTER labeling scheme as the label storage scheme adopted is conceptually very similar.

In QED, a quaternary code is defined as consisting of four numbers *0*, *1*, *2*, *3* and each number is stored with two bits, i.e.: *00*, *01*, *10*, *11*. The number *0* (and bit sequence *00*) is reserved as a separator and only *1*, *2*, and *3* are used in the QED code itself. Therefore, any positive integer can be encoded in the base 3 and represented as a quaternary code. For example, the DeweyID label 2.10.8 can be represented in the base 3 as 2.101.22 and can be encoded using quaternary codes and stored on disk as 11 00 100110 00 1111 (the spaces are present for readability only).

The primary advantage of separator storage schemes over control token schemes is that no matter how big the individual components of a label grow, the separator size remains constant. In the case of quaternary code, the separator size will always be 2-bits no matter how large the label grows. A disadvantage suffered by separator storage schemes compared to control token schemes is that control tokens permit a fast byte-by-byte or bit-by-bit comparison operation [7] and consequently facilitate fast query performance when labels have comparable lengths.

2.4 Prefix-free Codes

Prefix-free codes [4] are fixed-length or variable-length numeric codes that are members of a set which have the distinct property that no member in that set is a prefix to any other member in that set. For example, the set m={1,2,3,4} is a prefix set, however the set n={1,2,3,22} is not a prefix set because the member "2" is a prefix of the member "22".

The ORDPATH [18] dynamic labeling scheme uses prefix-free codes as its label storage scheme. The authors present two prefix-free encoding tables; we present their first encoding table in Table 1 and omit the second table as it is conceptually very similar.

Prefix-free Code	Number of bits	Value range
0000001	48	$[-2.8\text{x}10^{14}, -4.3\text{x}10^{9}]$
0000010	32	$[-4.3\text{x}10^{9}, -69977]$
0000011	16	[−69976, −4441]
000010	12	[−4440, −345]
000011	8	[−344, −89]
00010	6	[−88, −25]
00011	4	[−24, −9]
001	3	[−8, −1]
01	3	[0, 7]
100	4	[8, 23]
101	6	[24, 87]
1100	8	[88, 343]
1101	12	[344, 4439]
11100	16	[4440, 69975]
11101	32	$[69976, 4.3\text{x}10^{9}]$
11110	48	$[4.3\text{x}10^{9}, 2.8\text{x}10^{14}]$

Table 2: ORDPATH Variable-length Prefix-free codes and Value range

Example 2. To encode the ORDPATH label 1.152, we must encode each of the components in the label individually. 1 lies in the value range [0, 7] and hence will be represented using three bits (001) and have the prefix code 01. Thus, the full representation of 1 is 01 001. 152 lies in the value range [88, 343] and hence will be represented using 8 bits and have the prefix code 1100. Note, 152 in binary is the 8 digit number 10011000 but ORDPATH uses the binary representation of 64 to represent this number. 64 is obtained by subtracting the start of the value range from the number to be encoded, that is $152 - 88 = 64$. The binary representation of 64 is 01000000 (using 8 bits). Hence the full representation of 152 is 1100 01000000. Finally the full representation of the ORDPATH label 1.152 is 01 001 1100 01000000 (the spaces are present for readability only).

The ORDPATH prefix-free label storage schemes often require less bits to represent a label that the UTF-8 control token scheme - recall the label 1.152

requires 24-bits to be represented in UTF-8 but only 17 bits using ORDPATH prefix-free codes. However, the ORDPATH prefix free codes have higher computational costs in order to decode a label.

2.5 Critique of Label Storage Schemes

In this section, we outlined the four approaches underlying the implementation of all existing label storage schemes for XML to date: length fields, control token, separators, and prefix-free codes. No single approach stands out, each has their own advantages and limitations. Fixed length fields are ideal for static data and variable length fields are ideal for data that is rarely updated. Control token storage schemes *may* facilitate fast byte-by-byte label comparison operations if the properties of the labeling scheme is designed to take advantage of such operations. However the control token storage schemes proposed to date are not compact. The separator storage schemes offer compact label encoding however, the entire label must be decoded bit-by-bit in order to identify each individual component in the label. Lastly, prefix-free codes *may* also permit fast byte-by-byte comparisons but require a pre-computed prefix-free code table to encode and decode labels and a more complex encode/decode function that leads to higher label comparison computational costs.

The key problem we seek to address in this paper is the provision of scalability - that is a label storage scheme that will *never* require the labeling scheme to relabel existing nodes under any arbitrary combination of node insertions and deletions. The SCOOTER and QED labeling schemes are the only labeling schemes to successfully provide this feature by employing the separator label storage scheme in conjunction with node insertion algorithms that do not require the relabeling of existing nodes. Control tokens and prefix-free label storage schemes have been deployed by dynamic labeling schemes that numerically or alphanumerically encoded their labels (DeweyID [21], ORDPATH [18], DLN [2], LSDX [3], Vector [24], DDE [25], however, none of these labeling schemes are scalable as illustrated in [17]). In contrast, length field and separator label storage schemes have been deployed by bit-string dynamic labeling schemes. However, all binary encoded bit-string dynamic labeling schemes (ImprovedBinary [10], CDBS [12], EXEL [14], Enhanced EXEL [15]) are unable to avail of the separator storage scheme (because a bit sequence is reserved as a separator) and are not scalable. Lastly, all length field label storage schemes are subject to relabeling after an arbitrary large number of node insertions. Consequently, there does not exist a label storage scheme that enables binary encoded bit-string dynamic labeling schemes to completely avoid the relabeling of nodes. We address this problem now.

3 Fibonacci Label Storage Scheme

The Fibonacci label storage scheme may be employed by any binary encoded bit-string dynamic labeling scheme and enables the labeling scheme to completely

avoid the relabeling of nodes. The Fibonacci label storage scheme is a hybrid of the control token and length field classifications. Before we describe the label storage scheme, we provide a brief overview of the Fibonacci sequence [22] and the Zeckendorf representation [23].

Definition 1. *Fibonacci Sequence.*
The Fibonacci sequence is given by the recurrence relation $F_n = F_{n-1} + F_{n-2}$ *with* $F_0 = 0$ *and* $F_1 = 1$ *such that* $n \geq 2$.

The first 10 terms of the Fibonacci sequence are: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34. Each term in the sequence is the sum of the previous two terms.

Definition 2. *Zeckendorf Representation.*
For all positive integers n, there exists a positive integer N such that

$$n = \sum_{k=0}^{N} \epsilon_k F_k \quad \text{where } \epsilon_k \text{ is 0 or 1, and } \epsilon_k * \epsilon_{k+1} = 0.$$

This may be more informally stated as: every positive integer n has a unique representation as the sum of one or more distinct non-consecutive Fibonacci numbers. It should be noted that although there are several ways to represent a positive integer n as the sum of Fibonacci numbers, only one representation is the Zeckendorf representation of n. For example, the positive integer 111 may be represented as the sum of Fibonacci numbers in the following way:

1. $111 = 89 + 21 + 1$
2. $111 = 55 + 34 + 13 + 8 + 1$
3. $111 = 89 + 13 + 5 + 3 + 1$

However, only the first expression is the Zeckendorf representation of 111, because the second expression contains two consecutive Fibonacci terms (55 + 34) as does the third expression (5 + 3). We will exploit the property that no two consecutive Fibonacci terms occur in the Zeckendorf representation of a positive integer to construct the Fibonacci label storage scheme.

3.1 Encoding and Decoding the length of the label

We begin with a simple example providing an overview of the encoding process for the Fibonacci label storage scheme before we present our algorithms. Given a binary encoded bit-string label $N_{new} = 110101$,

- We first determine the length of N_{new}. It has 6 bits.
- We then obtain the Zeckendorf representation of the length of the label. The Zeckendorf representation of 6 is 5 + 1.
- We then encode the Zeckendorf representation of the label length as a Fibonacci coded binary string. Specifically, starting from the Fibonacci term F_2 (recall $F_0 = 0$ and $F_1 = 1$), if the Fibonacci term F_{k+1} occurs in the Zeckendorf representation of the label length, then the k^{th} bit in the Fibonacci coded binary string is set to "1". If the Fibonacci term F_{k+1} does *not* occur in the Zeckendorf representation, then the k^{th} bit in the Fibonacci coded binary string is set to "0".

- For example, the first term F_2 (1) occurs in the Zeckendorf representation of 6 and thus, the first bit in the binary string is "1". The second term F_3 (2) does not occur in the Zeckendorf representation of 6 and thus, the binary string is now "10". The third term F_4 (3) does not occur in the Zeckendorf representation of 6 and thus, the binary string is now "100". The fourth term F_5 (5) occurs in the Zeckendorf representation of 6 and thus, the binary string is now "1001". There are no more terms in the Zeckendorf representation of the length of N_{new} (6 bits), therefore stop.
- The Fibonacci coded binary string of the Zeckendorf representation will never contain two consecutive "1" bits, precisely because it is a Fibonacci encoding of an Zeckendorf representation. Also, given that the construction of the Fibonacci coded binary string stops after processing the last term in the Zeckendorf representation, we are certain the last bit in the Fibonacci coded binary string must be "1". We subsequently append an extra "1" bit to the end of the Fibonacci coded binary string to act as a control token or delimiter. Thereafter, we know the only place two consecutive "1" bits can occur in the Fibonacci coded binary string is at the end of the string. Thus the binary string is now "10011".
- The Fibonacci label storage scheme adopts a length field storage approach, which means we encode and store the size of the label immediately before the label itself. The last two bits of the Fibonacci coded binary string will always consist of two consecutive "1" bits and they act as a control token separating the length field of the label from the label itself. To complete our example, the label N_{new} (110101) is encoded and stored using the Fibonacci label storage scheme as 10011 110101 (the space is provided as a visual aid).

It can be seen from above that the Fibonacci label storage scheme is a hybrid of the control token and length field label storage schemes. In [1] and [5], the authors exploit a Fibonacci coding of the Zeckendorf representation of variable-length binary strings for synchronization and error correction during the transmission of codes. However, to the best of our knowledge, Fibonacci coded binary strings have never been proposed as a foundation for a label storage scheme nor have they been proposed to provide scalabilty to dynamic labeling schemes.

Algorithm 1 outlines the label length encoding process. It receives as input a positive integer n representing the label length and outputs a Fibonacci coded binary string of the Zeckendorf representation of n. Algorithm 2 outlines the label length decoding process which is the reverse transformation of algorithm 1.

3.2 Fibonacci Label Storage Scheme Size Analysis

In Table 3, we illustrate the relationship between the growth in label size and the corresponding growth in the quantity of labels that may be encoded. Given a label encoding length n, the quantity of labels that may be encoded with length n is equal to the Fibonacci term F_{n-1}. In [19], the authors prove that the average value of the n^{th} term of a sequence defined by the general recurrence relation G_n

Algorithm 1: `EncodeLabelLength`.

```
   /* Encode n to Fibonacci coded binary string of Zeckendorf representation of n.   */
   input  : n - a positive integer representing the length of a label.
   output: fibStr - a Fibonacci coded binary string of the Zeckendorf representation of n.
 1 begin
 2     F_0 ⟵ 0;
 3     F_1 ⟵ 1;
 4     F_start ⟵ F_0 + F_1;
 5     F_end ⟵ the largest Fibonacci number ≤ n;
 6     fibArray ⟵ the Fibonacci sequence from F_start to F_end inclusive;
 7     fibStr ⟵ "1";
 8     for (i=length(fibArray); i=1; i−−) do
 9         if (n ≥ fibArray[i]) then
10             fibStr ⟵ "1" ⊕ fibStr;
11             n ⟵ n − fibArray[i];
12         else
13             fibStr ⟵ "0" ⊕ fibStr;
14         end
15     end
16     return fibStr;
17 end
```

Algorithm 2: `DecodeLabelLength`.

```
   /* Decode a Fibonacci coded binary string of a Zeckendorf representation to n.   */
   input  : fibStr - a Fibonacci coded binary string of the Zeckendorf representation of n.
   output: n - a positive integer representing the length of a label.
 1 begin
 2     F_0 ⟵ 0;
 3     F_1 ⟵ 1;
 4     F_start ⟵ F_0 + F_1;
 5     fibCount ⟵ length(fibStr);
 6     fibArray ⟵ the first fibCount terms of the Fibonacci sequence from F_start inclusive;
 7     n ⟵ 0 ;
 8     for (i=1; i < length(fibArray); i++) do
 9         if (fibStr[i] == "1") then
10             n ⟵ n + fibArray[i];
11         end
12     end
13     return n;
14 end
```

$= G_{n-1} + - G_{n-2}$ increases exponentially. *Therefore, as the number of labels to be encoded using the Fibonacci label storage scheme increases exponentially, the corresponding growth in the size of the Fibonacci coded binary string is linear.* This demonstrates that when processing a large quantity of labels the Fibonacci label storage scheme scales gracefully.

4 Evaluation

In this section, we evaluate the Fibonacci label storage scheme by comparing it with three other label storage schemes, namely ORDPATH Compressed binary format, UTF-8 and the Separator label storage schemes. All label storage schemes were implemented in Java version 6.38 and all experiments were carried out on a 2.66Ghz Intel(R) Core(TM)2 DUO CPU with 4GB of RAM. The

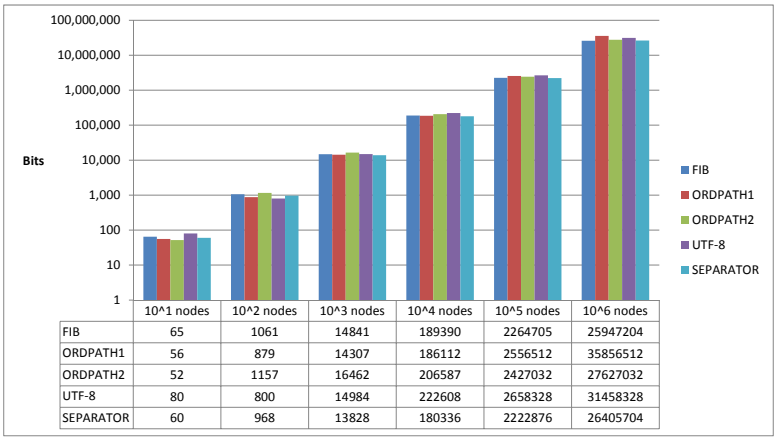
Growth Counter	Label Encoding Length	Num of labels
1	2	1
2	3	1
3	4	2
4	5	3
5	6	5
6	7	8
7	8	13
8	9	21
9	10	34
10	11	55
11	12	89
12	13	144
13	14	233
14	15	377
⋮	⋮	⋮
n	n + 1	F_n

Table 3: Fibonacci Encoded Label Length Growth Rate

experiments were performed 11 times, the time from the first run was discarded and the results of the subsequent 10 experiments averaged. For all experiments, the unit of storage is in *bits* and the unit of time is in milliseconds (ms). The ORDPATH prefix-free code tables and the array of Fibonacci numbers from 1 to N are computed once (in advance), and not each time a label is encoded/decoded, so as to reflect a real-world implementation scenario.

The Fibonacci and Separator label storage schemes were designed to encode bit-string labels, whereas the ORDPATH and UTF-8 label storage schemes were designed to encode integer-based labels. Consequently, to ensure an equitable and fair experimental evaluation, all four label storage schemes encode the positive integers from 1 to 10^n where n has the values from 1 to 6 inclusive. Given that the Fibonacci and Separator label storage schemes expect a bit-string label to encode, the integer is converted from base 10 to base 2 (binary) and the binary string representation of the integer is encoded. In Figure 1, we illustrate the storage costs for all four label storage schemes using labels derived from the integer encodings from 1 to 10^6 (Note: a logarithmic scale is used in the illustration). The ORDPATH compressed binary format provides a choice of two encoding tables to use; we present both encodings to enable a comprehensive evaluation and analysis. In this remainder of this section, "FIB" is used to denote the Fibonacci label storage scheme.

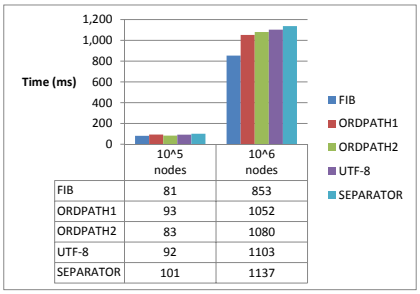
ORDPATH2 provides the most compact storage representation when encoding less than 10 integers. UTF-8 provides the most compact storage representation when encoding 10^2 integers and SEPARATOR when encoding 10^3 through 10^5 inclusive. When encoding 10^6 integers, FIB provides the most compact storage representation. This result is in line with our theoretical analysis in §3.2 which observed that as the number of labels to be encoded using FIB increases exponentially, the corresponding growth rate in the size of the Fibonacci coded



	10^1 nodes	10^2 nodes	10^3 nodes	10^4 nodes	10^5 nodes	10^6 nodes
FIB	65	1061	14841	189390	2264705	25947204
ORDPATH1	56	879	14307	186112	2556512	35856512
ORDPATH2	52	1157	16462	206587	2427032	27627032
UTF-8	80	800	14984	222608	2658328	31458328
SEPARATOR	60	968	13828	180336	2222876	26405704

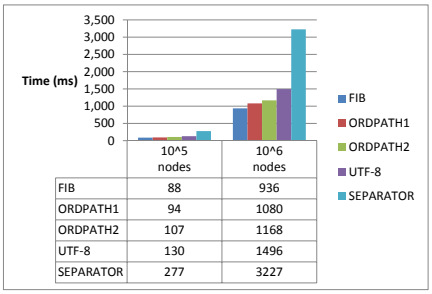
Fig. 1: Storage Costs of Encoding Integers for the Label Storage Schemes.

binary string is linear. Hence, although the performance of FIB is average for small to medium sized labels, FIB provides a highly compact storage representation for large labels. However, unlike ORDPATH and UTF-8, FIB is not subject to the overflow problem and will never require existing labels to be relabeled.



	10^5 nodes	10^6 nodes
FIB	81	853
ORDPATH1	93	1052
ORDPATH2	83	1080
UTF-8	92	1103
SEPARATOR	101	1137

(a) Label Storage Scheme Encoding Times



	10^5 nodes	10^6 nodes
FIB	88	936
ORDPATH1	94	1080
ORDPATH2	107	1168
UTF-8	130	1496
SEPARATOR	277	3227

(b) Label Storage Scheme Decoding Times

Fig. 2: Integer Encoding and Decoding times for Label Storage Schemes.

In Figure 2, we illustrate the computational processing times to encode and decode 10^5 and 10^6 integer labels. The times for 10^4 (or less) integer encodings are not shown because they are single digit results with negligible differences between them. FIB is the fastest label storage scheme at both encoding and decoding. FIB is the fastest because as a length field label storage scheme it only has to encode and decode the length of the label. The actual bit-string label is stored immediately after the Fibonacci coded binary string and can be read and written without having to process each individual bit. All of the other

label storage schemes must process the entire label to generate their encoding. ORDPATH1 has similar encode and decode computational processing costs. ORDPATH2 encodes more quickly than it decodes because the key size that maps to the range of the integer encoding grows more quickly than the encoding table employed by ORDPATH1. UTF-8 decodes approximately 30 percent slower than it encodes because when decoding, it must parse each individual byte in the multi-byte label and strip away the 2-bit control token at the start of each byte. SEPARATOR is the slowest at both encoding and decoding because it must parse every bit in the bit-string label in order to identify each individual component in the label. In summary, FIB is the fastest label storage scheme at both encoding and decoding, it is not subject to the overflow problem and will never required the relabeling of existing node labels.

5 Conclusions

In this paper, we provided a detailed overview of the existing state-of-the-art in label storage schemes for XML dynamic label schemes. We presented a new label storage scheme based on the Fibonacci sequence that may be employed by any binary encoded bit-string dynamic label scheme to completely overcome the overflow problem. Our experimental evaluation demonstrated that the Fibonacci label storage scheme offers storage costs comparable to existing approaches and is particularly well suited for large datasets. The computational processing costs of the Fibonacci label storage are also favorable when compared to existing approaches because the processing requirements are primarily determined by the length of the label and not dependent on the value of the label itself.

Apart from the In-MINDD project, there are many applications that can benefit from this approach. In earlier work [20], we managed repositories with large numbers of sensor values where repeated transformations were required to calibrate data in order to make it usable. However, certain aspects of sensor networks require different labels to what was managed in this current paper. As part of our future work, we are extending the Fibonacci label storage scheme beyond binary encoded bit-string labels to encode both numeric and alphanumeric labels. The goal is to provide an alternative label storage scheme that may be employed by all dynamic labeling schemes, offering a compact storage representation while minimizing processing costs.

References

1. Apostolico, A., Fraenkel, A.S.: Robust Transmission of Unbounded Strings Using Fibonacci Representations. Information Theory, IEEE Transactions on 33(2), 238–245 (1987)
2. Böhme, T., Rahm, E.: Supporting Efficient Streaming and Insertion of XML Data in RDBMS. In: DIWeb. pp. 70–81 (2004)
3. Duong, M., Zhang, Y.: LSDX: A New Labelling Scheme for Dynamically Updating XML Data. In: ADC. pp. 185–193 (2005)

4. Elias, P.: Universal Codeword Sets and Representations of the Integers. Information Theory, IEEE Transactions on 21(2), 194 – 203 (mar 1975)
5. Fraenkel, A.S., Kleinb, S.T.: Robust Universal Complete Codes for Transmission and Compression. Discrete Applied Mathematics 64(1), 31–55 (1996)
6. Gui, H., Roantree, M.: A Data Cube Model for Analysis of High Volumes of Ambient Data. Procedia CS 10, 94–101 (2012)
7. 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)
8. In-MINDD - INnovative, Midlife INtervention for Dementia Deterrence: (2013), online Resource http://www.inmindd.eu/
9. Jolles, J., Houx, P., van Boxtel, M., Ponds, R.: Maastricht Aging Study: Determinants of Cognitive Aging. Neuropsych Publishers (1995)
10. Li, C., Ling, T.W.: An Improved Prefix Labeling Scheme: A Binary String Approach for Dynamic Ordered XML. In: DASFAA. pp. 125–137 (2005)
11. Li, C., Ling, T.W.: QED: A Novel Quaternary Encoding to Completely Avoid Re-labeling in XML Updates. In: CIKM. pp. 501–508 (2005)
12. Li, C., Ling, T.W., Hu, M.: Efficient Processing of Updates in Dynamic XML Data. In: ICDE. p. 13 (2006)
13. Liu, J., Roantree, M., Bellahsene, Z.: A SchemaGuide for Accelerating the View Adaptation Process. In: ER. pp. 160–173 (2010)
14. Min, J.K., Lee, J., Chung, C.W.: An Efficient Encoding and Labeling for Dynamic XML Data. In: DASFAA. pp. 715–726 (2007)
15. Min, J.K., Lee, J., Chung, C.W.: An Efficient XML Encoding and Labeling Method for Query Processing and Updating on Dynamic XML Data. Journal of Systems and Software 82(3), 503–515 (2009)
16. O'Connor, M.F., Roantree, M.: Desirable Properties for XML Update Mechanisms. In: EDBT/ICDT Workshops (2010)
17. O'Connor, M.F., Roantree, M.: SCOOTER: A Compact and Scalable Dynamic Labeling Scheme for XML Updates. In: DEXA (1). pp. 26–40 (2012)
18. 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)
19. Rittaud, B.: On the Average Growth of Random Fibonacci Sequences. Journal of Integer Sequences 10(2), 3 (2007)
20. Roantree, M., Shi, J., Cappellari, P., O'Connor, M.F., Whelan, M., Moyna, N.: Data Transformation and Query Management in Personal Health Sensor Networks. J. Network and Computer Applications 35(4), 1191–1202 (2012)
21. 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)
22. Wolfram||Alpha: Fibonacci Numbers, Wolfram Alpha LLC edn. (December 2012), online Resource http://mathworld.wolfram.com/FibonacciNumber.html
23. WolframAlpha: Zeckendorf Representation, Wolfram Alpha LLC edn. (December 2012), online Resource http://mathworld.wolfram.com/ZeckendorfRepresentation.html
24. Xu, L., Bao, Z., Ling, T.W.: A Dynamic Labeling Scheme Using Vectors. In: DEXA. pp. 130–140 (2007)
25. 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)
26. Yergeau, F.: UTF-8, A Transformation Format of ISO 10646, Request for Comments (RFC) 3629 edn. (November 2003)