A Partitioning Scheme for Big Dynamic Trees

Atsushi Sudoh, Tatsuo Tsuji(✉), and Ken Higuchi

Information Science Department, Faculty of Engineering, University of Fukui,
Bunkyo 3-9-1, Fukui City, 910-8507, Japan
{sudou,tsuji,higuchi}@pear.fuis.u-fukui.ac.jp

Abstract. In this paper, we propose a scheme for partitioning dynamically growing big trees and its implementation. The scheme is based on the history-pattern encoding scheme for dynamic multidimensional datasets. Our scheme of handling big dynamic trees is relying on the history-pattern encoding, by which large scale datasets can be treated efficiently. In order to partition these dynamic trees efficiently, the encoding scheme will be improved and adapted to the partitioning. In our partitioning scheme of a tree T, the path from the T's root node to the root node of a partitioned subtree, is treated as an index for selecting the subtree. The path is split into a shared path and the local path in the subtree and each path is encoded by using the history-pattern encoding. The partitioning scheme contributes to the reduction of the storage cost and the improvement of the retrieval cost. In this paper, after our tree encoding scheme designed for the partitioning is described, some problems caused in the encoding are addressed and their countermeasure is presented. Finally, an implemented prototype system is described and evaluated.

1 Introduction

Tree graph is an extremely useful graph structure and is employed as a basic data structure in wide application domains such as sorting/searching, information retrieval, data mining, XML document processing, and so on. In these application domains, tree graphs are used for data/knowledge representation or control structure. It has been widely recognized that efficient data analysis and processing are enabled due to their simple and powerful operational capabilities.

Along with the increase of the application domains such as XMLDB or Web mining, in which dynamic growth of graph size or change of graph structures are caused frequently, highly efficient processing capability for such kind of dynamicity is strongly desired. Many researches have been conducted concerning on the schemes for clustering and partitioning a tree graph. The aim of these researches often includes improvement of query processing (e.g., [4]) or performance improvement obtained by parallel processing (e.g., [5]).

In this paper, we will propose an encoding scheme for large scale dynamic tree graphs with which re-encoding is not necessary even if a tree structure dynamically grows. This encoding scheme is based on the history-pattern encoding scheme [6, 7] designed for dynamically increasing multidimensional datasets.

Z. Bao et al. (Eds.): DASFAA 2017 Workshops, LNCS 10179, pp. 18–34, 2017.
DOI: 10.1007/978-3-319-55705-2_2

Implementation of dynamic trees based on the history-pattern encoding suffers from its own problem that the storage cost might be degraded when the insertion of new nodes occur at the level lower than the current highest level. In order to avoid this deficiency, a tree is partitioned into a set of subtrees. In our partitioning scheme, the path from the root node of a tree to a node n works as an index for selecting the partitioned subtree whose root node is n. This enables the reduction of the storage cost and the improvement of the retrieval cost. We present an history-pattern encoding scheme adapted to this partitioned tree. Note that by placing each partitioned subtree to separate machines on the network, our partitioning scheme can be well adapted to the distributed environment.

In the rest of this paper, after the related work is explained, the history-pattern encoding is outlined. Next the implementation of tree graphs is described. Then the countermeasure against the above problem will be presented. Subsequently, we propose a scheme for partitioning dynamically growing trees along with the improvement and adaptation of the history-pattern encoding to this partitioning scheme. Finally, an implemented prototype system is described and evaluated.

2 Related Work

As an important work related to the history-pattern encoding scheme, notion of *array database* [10, 11] for scientific applications can be listed. It provides a data model handling arrays as its main data structure. Scientific phenomena captured by arrays can be treated easily in this data model. Array databases can be well utilized in the various array oriented application domains other than scientific application. In addition to these array oriented applications, the history-pattern encoding is also based on array data structure, it can efficiently encode and handle a dynamic tree by embedding it to an dynamically extendible array. Therefore our approach can also cover tree-oriented application domains.

In order to handle dynamically growing trees, it is important to provide an efficient encoding scheme for each tree node n that can well capture and reflect the position in the tree using the path from the root node to node n. For example, [1, 2, 8] provide a encoding scheme for dynamic XML tree preserving the *document order* even if a new node is inserted in the tree.

However, these schemes have a problem in handling big XML trees. In [1, 2], the label length become so large and the storage cost is high [8] is based on the encoding scheme [9] similar to the history-pattern encoding, but the encoded result should be within single machine word; e.g., 64 bits. This makes it impossible to encode a big tree. In contrast the history-pattern encoding scheme can handle very big data with only small metadata for each tuple. In fact only with one byte metadata, a tuple whose encoded size is up to 256 bits can be represented and treated. At the same time, a dynamically increasing tuple dataset up to 2^{256} tuples can be handled. Therefore very big trees in their height and breadth can be implemented and handled.

3 History-Pattern Encoding

In this section, we describe the *history-pattern* encoding scheme for dynamic multidimensional datasets.

3.1 Data Structures for History-Pattern Encoding

Figure 1 illustrates the required data structures for the history-pattern encoding scheme. When an n-dimensional extendible array A extends, a fixed size sub-array equal to the size of the current A in every dimension is attached to the extended dimension. The number enclosed in Fig. 1 represents the insertion order of the element whose coordinate is corresponding to the tuple.

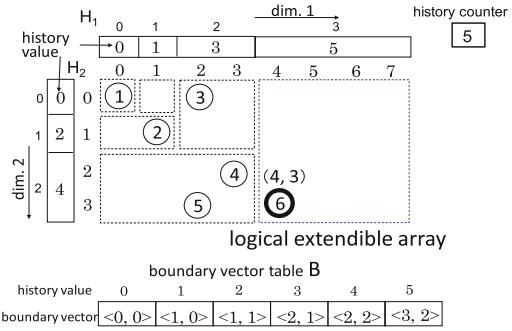


Fig. 1. Data structures for history-pattern encoding

The data structures for A consist of *history tables* and a *boundary vector table* that maintain extension history of A.

History Table

For each dimension i ($i = 1, \ldots, n$), the history table H_i is maintained. Each history value h in H_i represents the extension order of A along the i-th dimension and identifies the past *shape* of A when the history counter value was h; the counter value is initialized to 0 and incremented by one each time A is extended. H_i is a one-dimensional array, and each subscript k ($k > 0$) of H_i corresponds to the subscript range from 2^{k-1} to $2^k - 1$ of the i-th dimension of A covered by the sub-array S_h; S_h was attached to A at the extension along the i-th dimension when the history counter value was h.

For example, consider the coordinate (7, 10). Because $(7, 10) = (111_{(2)}, 1010_{(2)})$, the bit pattern sizes are 3 and 4, respectively. Therefore, $H_1[3]$ and $H_2[4]$ keep the history counter values when A was extended along the first and second dimension, respectively. It can also be determined that the subscript ranges of A covered by the attached sub-arrays are 4–7 for the first dimension and 8–15 for the second dimension.

Boundary Vector Table

The *boundary vector table* B is a single one-dimensional array whose subscript is a history value. Each element of B maintains the past *shape* of A represented by the

corresponding boundary vector when the history counter was a given value. Together with the boundary vector, B also maintains the dimension of A extended at the given history counter value. At initialization A includes only the element (0, 0, …, 0), and the history counter is initialized to 0. B[0] includes <0, 0, …,0> as its boundary vector.

Assume that the current history counter value is h, and B[h] includes $<b_1, b_2, \ldots, b_i, \ldots, b_n>$ as its boundary vector. When the current A extends along the i-th dimension, B[$h + 1$] includes $<b_1, b_2, \ldots, b_i + 1, \ldots, b_n>$ as its boundary vector.

In the history-pattern encoding, an extendible array A has two size types: real and logical. Assume that the tuples in an n-dimensional dataset M have been converted to the set of coordinates. Let s be the largest subscript of dimension k, and $b(s)$ be the bit size of s. Then, the real size of dimension k is $s + 1$, and the logical size is $2^{b(s)}$. The real size is the cardinality of the k-th attribute. In Fig. 1, the real size and the logical size are [4, 5] and [4, 8] respectively.

3.2 Array Extension

Suppose that a tuple, whose k-th attribute value is new, is inserted in M. This insertion increases the real size of A in dimension k by one. If the increased *real size* + 1 of dimension k does not exceed the current logical size $2^{b(s)}$, then A is not physically extended, and neither history table H_k nor the boundary vector table B is updated. However, if the *real size* of dimension k exceeds the current logical size, then A is logically extended. That is, the current history counter value h is incremented to $h + 1$, and this value is set to $H_k[b(s + 1)]$. Moreover, the boundary vector in B[h] is copied to B[$h + 1$], and the dimension k of the boundary vector is incremented (Fig. 1).

Note that h has one-to-one correspondence with its boundary vector in B[h] and uniquely identifies the past (logical) shape of A when the history counter value was h. To be more precise, for the history value $h > 0$, if the boundary vector in B[h] is $<b_1, b_2, \ldots, b_n>$, the shape of A at h was $[2^{b1}, 2^{b2}, \ldots, 2^{bn}]$. For example, as shown in Fig. 1, because the boundary vector for the history value 3 is <2, 1>, the shape of A when the history counter value was 3 is $[2^2, 2^1] = [4, 2]$. Note that h also uniquely identifies the sub-array that is attached to A at extension when the history counter value was $h - 1$, and vice versa.

3.3 Encoding/Decoding

Using the data structures described in Sect. 3.1, an n-dimensional coordinate $I = (i_1, i_2, \ldots\ldots, i_n)$ can be encoded to the pair $<h, p>$ of *history value* h and *bit pattern* p. The history tables H_i ($i = 1, \ldots, n$) and the boundary vector table B are used for the encoding. The history value h is determined as the maximum value in $\{H_k[b(i_k)] \mid 1 \leqq k \leqq n\}$, where $b(i_k)$ is the bit size of the subscript i_k in I. For each history value h, the boundary vector in B[h] gives the bit pattern size of each subscript in I. According to this boundary vector, the coordinate bit pattern p can be obtained by concatenating the subscript bit pattern of each dimension in descending order (from the lower to the higher bits of p). The storage unit for p can be one word length, i.e., 64 bits.

In Fig. 1, the history-pattern encoding <*h*, *p*> of array element (4,3) is shown as an example. $H_1[b(4)] = H_1[b(100_{(2)})] = H_1[3] = 5$ and $H_2[b(3)] = H_2[b(11_{(2)})] = H_2[2] = 4$. Since $H_1[b(4)] > H_2[b(3)]$, *h* is $H_1[3] = 5$. So element (4,3) is known to be included in the *sub-array* on dimension 1 at history value 5. Therefore, the boundary vector to be used is <3, 2> in *B*[5]. In (4, 3) to be encoded, the subscript 4 of the first dimension and the subscript 3 of the second dimension form the upper 3 bits and lower 2 bits of *p*, respectively. Therefore *p* becomes $10011_{(2)} = 19$. Eventually, the element (4, 3) is encoded to <5, 19>. Generally, the bit size of history value *h* is rather small compared to that of pattern *p*; if the storage size for the pair is assumed to be 16 bits, typically the upper 4 bits are for *h*, and the lower 12 bits are for *p*.

Conversely, to decode the encoded pair <*h*, *p*> to the original *n*-dimensional coordinate $I = (i_1, i_2, \ldots, i_n)$, the boundary vector in *B*[*h*] is known. Then, the subscript value of each dimension is sliced out from *p* according to the boundary vector. Note that the procedure for extending an extendible array described in Sect. 3.2 assures the following important property on <*h*, *p*>.

[Property 1] History value *h* denotes the bit size of its coordinate bit pattern *p*.

It should be noted that due to this property, *h* can be used as the header of the en coded tuple <*h, p*>. This enables the output file of the encoded results to be a sequential file of packed variable length records.

3.4 Implementation of a Multidimensional Dataset

As well as the core data structures (the history tables and the boundary vector table) presented in Sect. 3.1, three additional types of data structures are required to implement a multidimensional dataset M using history-pattern encoding.

CVT_i (attribute subscript conversion tree) converts the attribute values of dimension $i (i = 1, \ldots, n)$ to their subscript values of the corresponding extendible array. CVT_i is implemented using a B^+ tree. CVT_i is used for tuple encoding.
AVT_i (attribute value table) is a one-dimensional array for dimension i ($i = 1, \ldots, n$). If attribute value *v* is mapped by CVT_i to subscript *k*, the *k*-th element of AVT_i keeps *v*. AVT_i is used for tuple decoding.
ETF (encoded tuple file) is an output file of the encoded results for M.

We call the implementation scheme of M using the core data structures in Fig. 1 together with the above data structures as HPMD (History Pattern implementation of Multidimensional Datasets).

Figure 2 shows encoding of the tuple set M at the left side using HPMD, and Fig. 3 is the produced CVTs and AVTs. In fact, Fig. 1 is the core data structures constructed from this example. Tuples in M are encoded sequentially in the input order. Each attribute value is converted to its coordinate subscript using the CVT of the corresponding dimension. The subscript values are assigned from 0 in the order that the new attribute value appears.

	Company	Item		coordinate	boundary vec	<history, pattern>
1	D	Pencil	→	(0, 0)	<0,0>	<0, .> = <0,0>
2	B	Scissors	→	(1, 1)	<1,1> ○	<2, 1.1> = <2,3>
3	C	Pencil	→	(2,0)	<2,1> ○	<3,10.0> = <3,4>
4	A	Ruler	→	(3,2)	<2,2> ○	<4,11.10> = <4,14>
5	C	Eraser	→	(2,3)	<2,2>	<4,10.11> = <4,11>
6	E	Scissors	→	(4,1)	<3,2> ○	<5,100.01> = <5,17>

Remark : ○ denotes that the insertion of the tuple causes the extension of the logical size of the le array

Fig. 2. An encoding example of a multidimensional dataset

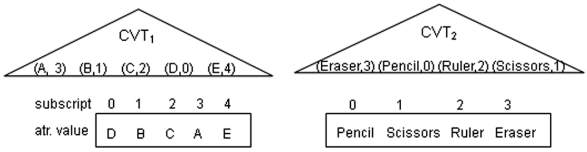


Fig. 3. Produced CVTs and AVTs for Fig. 2

3.5 Adding a New Attribute

In HPMD, the cost of adding a new attribute dynamically after the schema definition is very small. For example consider the case where a new attribute value "Price" is added to the tuple set shown in Fig. 2. Dimensional extension of HPMD is done from two dimensions to three dimensions; for the third dimension "Price", a history table H_3 is created and added to the core data structures shown in Fig. 1, and attribute subscript conversion tree CVT_3 and attribute value table AVT_3 are created and added for the new dimension attribute.

Moreover, boundary vectors in the boundary vector table become three dimensional as is shown in Fig. 4. Existing <history, pattern>s already encoded according to the two dimensional boundary vectors are not necessary to be re-encoded, and also the boundary vector table is unnecessary to be reorganized. This is because that the boundary vectors before extension can be treated as three dimensional after the dimensional extension; the value of the dimension three of the vector can be treated as 0 and this implies that the bit string for the subscript of the dimension three is a null string.

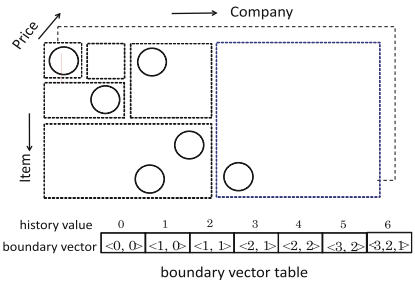


Fig. 4 Dimensional extension of Fig. 1

4 Storage Scheme for Dynamic Tree

In this section, we describe the mapping scheme of a dynamic tree to HPMD.

4.1 Mapping Dynamic Tree

Each depth level of a tree is mapped to a dimension of an extendible array. The number that represents each node's relative position among its siblings is mapped to the subscript value of the corresponding array dimension. Therefore, each node can be uniquely specified by its corresponding n-dimensional coordinate in the array (See Fig. 5). The coordinate can be encoded to its corresponding <*history value*, *pattern*>. Note that each level of a tree is mapped to the dimension number of the array in ascending order. When the tree height increases by one, the HPMD dimensionality also increases by one, as is shown in Fig. 5.

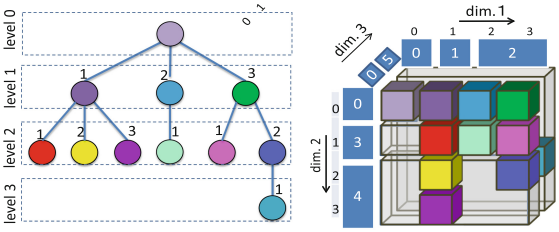


Fig. 5. Mapping dynamic tree to an extendible array

We can see from Fig. 5 that horizontal nodes insertion along some level in a tree graph results in the extension along the corresponding dimension. Also we can observe that the tree increases its height by one, a new dimension is added to the corresponding extendible array and this dimensional extension can be performed efficiently as was described in Sect. 3.5.

4.2 Handling Labeled Tree Graphs

The tree graph model adopted here follows XML tree [12] as :

"Each edge of a tree graph T has a label and each node in T stores its own data. More than one edge might be labeled with the same label. Nodes and edges can be dynamically added or deleted."

Figure 6(a) is an example of a tree graph subjecting to this model. A tree graph T will be implemented by the two kinds of graph embedding to an extendible array. See Fig. 6(b). One is the embedding of the tree graph T1 produced by dropping the edge labels of T. Each node in T1 is mapped to its associated coordinate of an extendible array and this coordinate is encoded to its <history, pattern> as in Fig. 5. Due to the mapping strategy, we can see that this encoded result hp keeps the position information of all its ancestor nodes and hp also its position among its brother nodes in T1. According to these position information we can perform the structural traversal in T1.

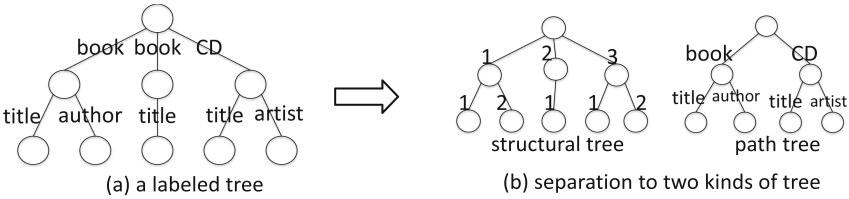


Fig. 6. Separation of a tree graph to two kinds of tree

The other kind of graph is for path retrievals in T. A path from the root node *r* of T to node *n* is the concatenation of the edge labels on the path. In our tree graph model, more than one edge might be labeled with the same label. Therefore a path in T might access more than one node. A tree graph T2 is a functional graph produced from T. T2 For each node *n* in T2 a path from the root node to *n* cannot access any other nodes. Note that T2 is *dataguides* in [3]. T2 is embedded in a HPMD; edge labels on the level *k* of T2 are mapped to their corresponding subscripts by CVT_k and each node in T2 is encoded by history-pattern encoding as in T1. T1 and T2 will be called structural tree and path tree respectively.

Both the structural information and path information of each node in T can be compactly aggregated in <history, pattern> in structural tree and path tree, respectively. This would provide the advantage of our implementation scheme of T both in storage cost and retrieval costs. In the following, the encoded result <history, pattern> for each node in the structural tree and the path tree will be called as nID (node ID) and pID (path ID), respectively.

5 Problems on Our Storage Scheme

There are following two problems on our storage scheme for tree graphs presented in Sect. 4.

5.1 Duplication of Ancestor Path

The first problem is the duplication of the ancestor path among the child nodes of the same parent node. This is an inherent problem in tree structures. Consider the example shown in Fig. 7. There exist three nodes on level 5. The coordinates of these nodes in the corresponding extendible array are (2, 3, 2, 1, 1), (2, 3, 2, 1, 2), (2, 3, 2, 1, 3), and the ancestor patterns corresponding to (2, 3, 2, 1) are stored in duplicate. If the tree grows vertically, the storage overhead caused by such duplication of ancestor patterns would be increased.

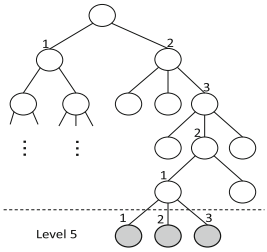
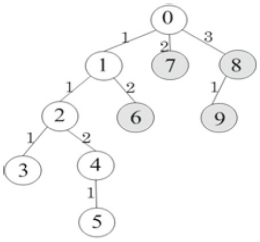


Fig. 7. Duplication of ancestors

5.2 Problem Caused by Expansion Order of Dynamic Tree

The second problem concerns the one caused by employing HPMD for implementing dynamic trees. Namely, depending on the order of vertical and horizontal growing of the trees, the higher dimensions of the coordinate pattern might be occupied by the redundant 0's and the storage cost would be degraded. Such redundancy might arise when the extendible array used for tree mapping extends its size along the dimension other than the maximum one; i.e., when a node insertion at a level lower than the maximum level of the tree causes an extension of the extendible array.

Figure 8(a) shows an example of growing tree and the boundary vector table of the related extendible array. The number of a tree node represents its insertion order. Figure 8(b) and (c) show the history tables and the boundary vector table of the extendible array for Fig. 8(a) respectively. Figure 9 shows the encoding sequence of the tree nodes according to the input order.



(a) a growing tree

subscript		0	1	2	3	4
dim.1	history	0	1	7		
dim.2	history	0	2	6		
dim.3	history	0	3	4		
dim.4	history	0	5			

(b) history table

history	0	1	2	3	4	5	6	7
boundary vec.	⟨0⟩	⟨1⟩	⟨1,1⟩	⟨1,1,1⟩	⟨1,1,2⟩	⟨1,1,2,1⟩	⟨1,2,2,1⟩	⟨2,2,2,1⟩

(c) boundary vector table

Fig. 8. A growing tree

	coordinate	boundary vec.	<history, pattern>		
node 0 →	(0)	<0>	<0, .>	=	<0,0>
node 1 →	(1)	<1> ○	<1,1>	=	<1,1>
node 2 →	(1, 1)	<1,1> ○	<2,1.1>	=	<2,3>
node 3 →	(1, 1, 1)	<1,1,1> ○	<3,1.1.1>	=	<3,7>
node 4 →	(1, 1, 2)	<1,1,2>○	<4,1.1.10>	=	<4,14>
node 5 →	(1, 1, 2, 1)	<1,1,2,1> ○	<5,1.1.10.1>	=	<5,29>
node 6 →	(1, 2, 0, 0)	<1,2,2,1>○	<6,1.10.00.0>	=	<6,48>
node 7 →	(2, 0, 0, 0)	<2,2,2,1>○	<7,10.00.00.0>	=	<7,64>
node 8 →	(3, 0, 0, 0)	<2,2,2,1>	<7,11.00.00.0>	=	<7,96>
node 9 →	(3, 1, 0, 0)	<2,2,2,1>	<7,11.01.00.0>	=	<7,104>

Fig. 9. The encoding of the tree nodes in Fig. 8

Since the insertions of node 1, 2, 3 increase the height of the tree, dimensional extensions occur and the related extendible array becomes 3 dimensional. At this point, the history counter value is 3. The insertion of node 4 extends the array along the dimension 3, since the subscript bit size of this dimension increases by 1. The insertion of node 5 causes dimensional extension and the maximum tree level becomes 4.

The insertions of node 6 and node 7 cause array extensions along dimension 2, and dimension 1 respectively, since both of the subscript bit sizes of these dimensions increase from 1 bit to 2 bits. At this point the dimension bit sizes of the array becomes <2, 2, 2, 1> and the history counter value becomes 7. The insertions of node 8 and node 9 do not exceed these dimension bit sizes, so no array extensions occur by these insertions. The levels (i.e., dimensions) of node 6, 7, 8, 9 are all under this maximum level.

As we can see from Fig. 9, the coordinates for node 6, 7, 8, 9 are encoded to the <history, pattern> s as <6,1.10.00.0>, <7,10.00.00.0>, <7,11.00.00.0>, <7,11.01.00.0> respectively using their boundary vectors. Note that the history value of the coordinate (i_1, i_2,…, i_k, 0, 0,…, 0) where $i_j \neq 0$ ($1 \leq j \leq k$), can be determined by (i_1, i_2,…, i_k) by comparing history values in $H_j[i_j]$ ($1 \leq j \leq k$) as explained in Sect. 3.3.

Note also that higher dimensions of the pattern parts of these encoded results might be occupied by redundant 0's. Such redundancies arise frequently when the tree gains its size in the vertical direction at its early stage of growth as in Fig. 8(a). Such situation is typical in the case when an XML document is transformed to its XML tree in *preorder* and loaded into main memory. According to our prior experimentation for constructing HPMD from a tree graph, greater part of the output ETF is proved to be occupied by these redundant 0's.

6 Encoding Methods for Eliminating Redundancy

In this section, we will present two kinds of an encoding method to eliminate the redundancy arising in the tree encoding described in Sect. 5. Since the path trees are generally very small compared with the structural trees, we will apply our improved schemes to only structural trees.

6.1 Horizontal Partitioning

This method separates the output file ETF of the encoded results by tree level. Let the current maximum level of a tree be n. If the level of a tree node is k ($1 \leq k \leq n$), the encoded result of the node is stored in ETF_k. When the current maximum level of the tree increases, ETF_{n+1} is created.

For the nodes 7, 8 in Fig. 8(a), their encoded results become <7, 10.> = <7, 2> and <7, 11.> = <7, 3> being the redundant 0's on dimensions 3, 4, 5 dropped out and stored in ETF_1 sequentially. For the nodes 6, 9, their encoded results become <6, 1.10> = <6, 6> and <7, 11.01> = <7, 13> being the redundant 0's on dimensions 4, 5 dropped out and are stored in ETF_2.

Decoding of each <*h, p*> in ETF_k can be done by knowing the boundary vector in B[*h*] = <b_1, b_2,…, b_k,……>; pattern *p* can be decoded to its *k* dimensional coordinate (i_1, i_2,…, i_k) by using b_1, b_2,…, b_k. For example, in Fig. 9, for the encode <6, 6> in ETF_2, first B[6] = <1,2,2,1> is inspected. Since <6, 6> = <6, $110_{(2)}$> is stored in ETF_2, the tree level of the node is 2. Therefore the first and the second elements of the boundary vector is used for decoding, so the pattern part $110_{(2)}$ is split into $1.10_{(2)}$, namely 1 bit and 2 bits. Hence the <6, 6> is decoded to the coordinate (1,2) which specifies node 6. In Fig. 9(b), we can know that the encode of node 6 is actually encoded to <6, 48> = <6,1.10.00.0> according to the usual encoding scheme shown in 2.3; 3 redundant bits 0's.

This method of separating ETF has the advantages:

(1) Storage overhead by the redundant 0's does not arise.
(2) When the levels of the retrieval target can be narrowed in advance, only their corresponding separate ETF files are sufficient to be searched.

Conversely, the disadvantage of this method includes:

(3) Retrieval that requires traversing tree structure might be slow, since the horizontal partitioning of ETF breaks down the paths from the root node.

Organization of ETF in the Horizontal Partitioning
Each ETF_k above is organized to reflect the cross reference e between a node of structural tree and its node of path tree in Fig. 6(b). For the cross reference, ETF_k is simply organized as a sequential file keeping the pairs of nID (node ID) and its corresponding pID (path ID). See Sect. 4.2 on nID and pID.

6.2 Attaching Level of a Node as Meta Data

Let *k* be the current maximum level of a tree and (i_1, i_2,…, i_k) be the coordinate of a tree node, we will add the tree level *l* of the node to the coordinate as meta data like (*l*, i_1, i_2,…, i_l, i_{l+1},…, i_k) (for $l \leq j \leq k$, $i_j = 0$). This coordinate is encoded to its <*h*, *p*> using the corresponding HPMD. If the boundary vector B[*h*] = <b_l, b_1, b_2,…, b_k>, the coordinate (*l*, i_1, i_2,…, i_l) is encoded according to the boundary vector <b_l, b_1, b_2,…, b_l>. Figure 10 shows the encoding of the tree nodes in Fig. 8(a) attaching the node level. For example, the encoded results of node 6, 7, 8, 9 in Fig. 9 are <9, 21>, <10, 6>, <10, 7>, <10, 45> respectively with the redundant 0's being dropped out.

Conversely, the decoding of <*h*, *p*> can be done as follows. The boundary vector B[*h*] = <b_l, b_1, b_2,…, b_k> is inspected and the upper b_l bits are sliced out from *p*. If the value of this b_l bits is *l*, the rest of *p* is decoded according to the boundary vector <b_1,b_2,…,b_l> to obtain its coordinate (i_1, i_2,…, i_l). For example, consider the node whose encode is <10, 6>. Since B[10] = <3, 2, 2, 2, 1> and 6 is divided to 001.10, the upper 3 bits of 6 is $001_{(2)}$, so the level of the node is known to be 1 and decoded as (2, 0, 0, 0). Note that 2 + 2 + 1 = 5 bits of 0 can be saved in the encode.

This method of attaching level of a node has the advantages:

	coordinate	boundary vec.	<history, pattern>		
node 0 →	(0, 0)	<0,0>	<0, . >	=	<0,0>
node 1 →	(1, 1)	<1,1> ○	<2, 1.1>	=	<2,3>
node 2 →	(2, 1, 1)	<2,1,1> ○	<4, 10.1.1>	=	<4,11>
node 3 →	(3, 1, 1, 1)	<2,1,1,1> ○	<5, 11.1.1.1>	=	<5,31>
node 4 →	(3, 1, 1, 2)	<2,1,1,2>○	<6, 11.1.1.10>	=	<6,62>
node 5 →	(4, 1, 1, 2, 1)	<3,1,1,2,1> ○	<8, 100.1.1.10.1>	=	<8,157>
node 6 →	(2, 1, 2, 0, 0)	<3,1,2,2,1>○	<9, 010.1.10.00.0>	=	<9,21>
node 7 →	(1, 2, 0, 0, 0)	<3,2,2,2,1>○	10, 001.10.00.00.0	=	<10,6>
node 8 →	(1, 3, 0, 0, 0)	<3,2,2,2,1>	10, 001.11.00.00.0	=	<10,7>
node 9 →	(2, 3, 1, 0, 0)	<3,2,2,2,1>	10, 010.11.01.00.0	=	<10,45>

Fig. 10. The encoding of the tree nodes in Fig. 8(a) with meta data

(1) Storage overhead by the redundant 0's does not arise.
(2) Retrieval along a tree path might be efficient since the method preserves the tree structure. Conversely, the disadvantage of this method includes:
(3) Storage cost for keeping level of a node would arise.

7 Vertical Partitioning

The disadvantage of horizontal partitioning method stated in Sect. 6.1 is serious for structural oriented retrieval along tree paths as will be confirmed in Sect. 8.3. Here we present another method for partitioning large scale dynamic tree into a set of subtrees using the method stated in Sect. 6.2. This avoids both of the two problems stated in Sect. 5.

7.1 Vertical Partitioning of a Structural Tree

Let T be a structural tree. T is partitioned into a set of subtrees $\{T_0, T_1, \ldots, T_m\}$. The scheme might split a path from the root node of T at the root node of some subtree T_i ($1 \leq_i \leq n$). Figure 11 shows an example of vertical partitioning of T. Two kinds of HPMD would be generated according to the organization of ETF, namely the output file of the encoded tuple file. One is for T_0 and the other is for $\{T_1, \ldots, T_m\}$. Here T_0 functions as an index for accessing the vertically partitioned subtrees and implemented by using a B +tree. T_i ($1 \leq i \leq m$) is implemented by using a sequential file.

7.2 Splitting a Path

Let r be a root node of T and l be a level of T. Let $a_1\_a_2\_\cdots\_\ a_{l-1}\_a_l$ be a subscript path reaching to the node n_l, which is the root node of the subtree T_i ($1 \leq i \leq m$); namely $(a_1,a_2,\cdots,a_l)$ is the coordinate of the node $n_{l..}$ Note that T_i has more than one node including its root node n_l. For a node n_p in T_i, the subscript path $p = ra_1\_a_2\_\cdots\_a_l\_a_{l+1}\_\cdots\_a_p$ that reaches n_p can be split into two sub-paths; $sp = ra_1\_a_2\_\cdots\_a_l$ and $lp = n_l a_{l+1}\_\cdots\_a_{p,.}$ Note that the sp can be shared among the path from r to an arbitrary node in T_i. Here the node

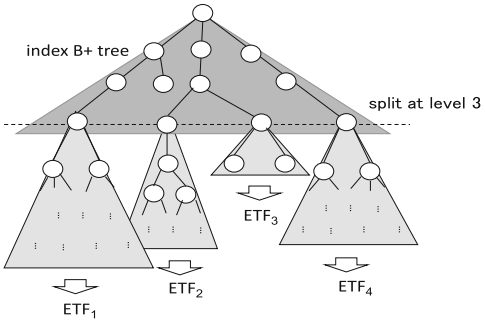


Fig. 11. Vertical partitioning of a dynamic tree

n_l will be called as the split point of the subscript path p. sp and lp are called as shared path and local path for T_i respectively. Each root node in T_i ($1 \leq i \leq m$) is a split point.

Figure 12 shows an example showing this splitting scheme at level 3. In the example, the path $r2\_3\_2\_1\_9$ is split into two sub-paths $r2\_3\_2$ and n_l1_9, where $n_l = r2\_3\_2$ is the split point.

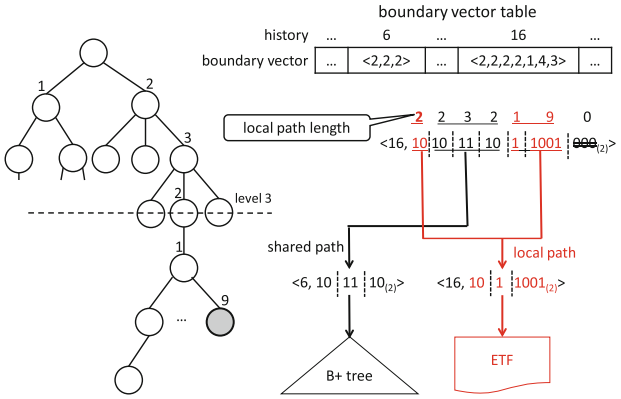


Fig. 12 Encoding shared path and local path

7.3 Encoding Shared Path and Local Path

For a node n_p in the subtree T_i, let the subscript path from the root node r of T be $ra_1\_a_2\_\cdots\_a_l\_a_{l+1}\_\cdots\_a_p$, where $ra_1\_a_2\_\cdots\_a_l$ be the shared path. The encoding of the shared path is straightforward. The coordinate $(a_1,a_2,\cdots,a_l)$ of the shared path is encoded to its <history, pattern>, which is stored in the index B+tree in the key part; the data part is the, reference to the corresponding subtree. On the other hand, the local path $n_la_{l+1}\_\cdots\_a_p$ is encoded according to the scheme stated in Sect. 6.2 in order to avoid the redundant storage occupied by the trailing 0's in the pattern part. Namely, the local path

length *p-l* is attached to the coordinate (a_{l+1},a_{l+2},⋯,a_p) as its meta information, so actually (*p-l*, a_{l+1},a_{l+2},⋯,a_p) is encoded.

Figure 12 shows the actual implementation and encodings for a shared path and a local path. Note that the local path length is attached on the top of the full path coordinate and this length is used in encoding the local path. Note also that the redundant trailing 0's are not necessary to be stored due to this local path length.

In Fig. 12, the coordinate of the full path *r*2_3_2_1_9 is (**2**, 2, 3, 2, 1, 9, **0**); the first 2 is the local path length and the last 0 is the redundant trailing 0. Let this coordinate be encoded to the history value 16 and the boundary vector B [16] be <2, 2, 2, 2, 1, 4, 3>. The coordinate of the shared path is encoded to <6, $10.11.10_{(2)}$> = <6, 46> and this is stored in the index B+tree with its reference to the corresponding subtree. The coordinate of the local path should be (1, 9, **0**) due to the second problem in Sect. 5.2. But, in order to suppress the redundant 0 on the third subscript, the first 2 of the full path coordinate is attached to the local path coordinate. Hence the local path coordinate becomes (**2**, 1, 9). This coordinate is encoded to <16.10.1.1001> by referring the boundary vector B [16], and the result is entered into the corresponding ETF file.

8 Experimental Evaluations

Using the constructed prototype systems, experimental evaluations were conducted on our partitioning schemes for dynamic trees.

8.1 Evaluation Environment

(1) Used machine:
CPU: Intel Core i7 920, 2.67 GHz
Memory: 48 GB
OS: Cent OS 5.7 (Linux 2.6.18)

(2) Evaluated implementations:
For the horizontal partitioning scheme explained in Sect. 6.1, the following implementation is used. We denote this implementation as HP in the following.

(a) Let n be the maximum level of the current dynamic tree T. For each horizontal level k of ($1 \leq k \leq n$,) of T, the encoded <history, pattern>s of all the nodes on the level k are stored in the single ETF_k.
For the vertical partitioning scheme presented in 7, the following two kinds of implementation were used for evaluations, depending on the organization of ETF (Encoded Tuple File). See Fig. 11.

(b) The encoded <history, pattern>s for nodes in each subtree T_i ($1 \leq i \leq m$) are stored in the single ETF_i.

(c) The encoded <history, pattern>s for all nodes in all subtrees T_i ($1 \leq i \leq m$) are stored in a single ETF file. In this ETF file, the encoded <history, pattern>s of the nodes in each subtree T_i are kept in the same single page list.
We denote the implementations (b) and (c) above as VP_1, and VP_2 respectively in the following.

(3) Input tree data file:
The input tree data file was produced from an XML file, which was generated by using *xmlgen* program provided by XMark [13]. The produced data file consists of only the pairs of <*element_name*> and </*element_name*> in the generated XML file by removing all attributes and texts. The produced tree data file is:
file size: 332,807,472 Bytes
number of nodes: 16,701,210
maximum tree level: 12

8.2 Storage Cost and Construction Cost

For HP in Sect. 8.1, the total size of ETF_k $(1 \leq k \leq n)$, for VP-1, the total size of index B+tree and ETF_k $(1 \leq k \leq n)$, and for VP_2, the total size of index B+tree and ETF were measured. See Fig. 13. Since HP is the simplest organization, the storage size is the smaller than VP_1 and VP_2, both of which keeps the index B+tree. Vertical partitioning of the structural tree enables sharing ancestor paths contributes to suppress the storage size, however index B+tree occupies larger storage size. VP_2 is larger than VP_1 due to the internal fragments in the page list.

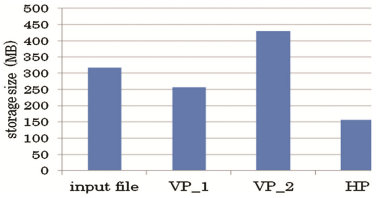


Fig. 13. Storage cost

Figure 14 shows the constructing HPMD data structures from the input tree data file. Since constructing index B+tree is time consuming, VP_1 and VP_2 are much slower than HP.

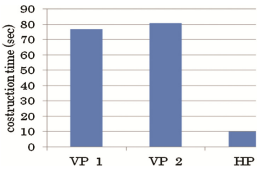


Fig. 14. Construction time

8.3 Retrieval Time

One percent of the total number of the structural tree nodes were randomly selected for retrieval.

Structural Retrieval

Each selected node is set as a current node, and various structural retrievals are performed. It can be observed in Fig. 15 that both VP_1 and VP_2 are much faster compared with HP. While HP must read every nID in ETF_k files of the related level sequentially, both VP_1 and VP_2 are suitable for structural retrieval; they only read the nIDs of the related subtrees The disadvantage of HP is distinct in the retrieval of *descendent* axis. See Fig. 15.

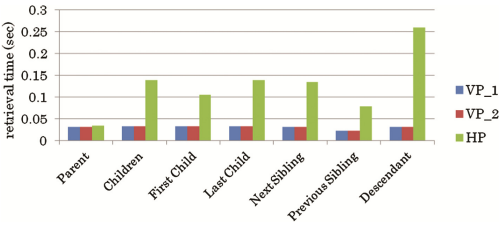


Fig. 15. Structural retrieval time

Path Retrieval

For the label path expressions to the selected structural tree nodes, their retrieval times are measured and averaged. See Fig. 16. VP_1 and VP_2 are both negligibly small compared with HP. This is because that in HP, if a label path expression is $l_1\_l_2\_\ldots\_l_k$ $(1 \le k \le n)$, all the ETF_i $(1 \le i \le k)$ files should be searched sequentially. On the contrary, in VP_1 and VP_2, only search the related subtrees as in the structural retrieval. See Fig. 16.

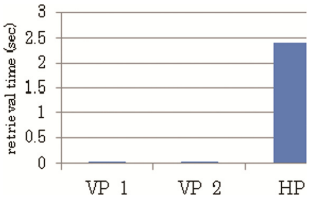


Fig. 16. Path retrieval time

9 Conclusion

We have presented and described partitioning schemes for big dynamic trees based on the history-pattern encoding. We have also evaluated them and confirmed that the horizontal partitioning can be compactly stored, but the traversal along the tree structure is unsuitable, and both the structural and the path retrievals are time consuming. On the other hand, the scheme of the vertical partitioning by path splitting is suitable for the tree traversal and the retrieval is very fast. We are now designing the load balancing

scheme among the partitioned ETFs in order to adapt our partitioning scheme to the distributed environments.

Acknowledgments. This work was supported by JSPS KAKENHI Grant Number JP26330132.

References

1. O'Neil, P.E., O'Neil, E.J., Pal. S., Cseri, I., Schaller, G., Westbury, N.: ORDPATHs: insert-friendly XML node labels. In: Proceedings of the ACM SIGMOD, pp. 903–908 (2004)
2. Li, C., Ling, T.W.: QED: a novel quaternary en-coding to completely avoid re-labeling in XML updates. In: Proceedings of CIKM 2005, pp. 501–508 (2005)
3. Goldman, R., Widom, J.: Dataguides: enabling query formulation and optimization in semistrucutured databases. In: Proceedings of VLDB, pp. 436–445 (1997)
4. Kanbara, T., Ueshima, S.: Distance range queries in spatial index tree. Trans. Inf. Process. Soc. Jpn.: Database **5**(1), 1–17 (2012)
5. Kido, K., Amagasa, T., Kitagawa, H.: A scheme for parallel processing of XML data using PC clusters. In: Proceedings of Database Engineering Workshop, 6B-oi3 (2006)
6. Makino, M., Tsuji, T., Higuchi, K.: History-pattern implementation for large-scale dynamic multidimensional datasets and its evaluations. In: Renz, M., Shahabi, C., Zhou, X., Cheema, M.A. (eds.) DASFAA 2015. LNCS, vol. 9050, pp. 275–291. Springer, Heidelberg (2015). doi:10.1007/978-3-319-18123-3_17
7. Makino, M., Tsuji, T., Higuchi, K.: History-pattern encoding for large-scale dynamic multidimensional datasets and its evaluations. IEICE Trans. **E99-D**(4), 989–999 (2016). (extended version of [6])
8. Li, B., Kawaguchi, K., Tsuji, T., Higuchi, K.: A labeling scheme for dynamic XML trees based on history-offset encoding. Trans. Inf. Process. Soc. Jpn.: Database **3**(1), 1–17 (2010)
9. Hasan, K.M.A., Tsuji, T., Higuchi, K.: An efficient implementation for MOLAP basic data structure and its evaluation. In: Kotagiri, R., Krishna, P.R., Mohania, M., Nantajeewarawat, E. (eds.) DASFAA 2007. LNCS, vol. 4443, pp. 288–299. Springer, Heidelberg (2007). doi: 10.1007/978-3-540-71703-4_26
10. Brown, P.G.: Overview of sciDB: large scale array storage, processing and analysis. In: Proceedings of the ACM SIGMOD, pp. 963–968 (2010)
11. Soroush, E., Balazinska, M.: Time travel in a scientific array database. In: Proceedings of ICDE, pp. 98–109 (2013)
12. Extensible Markup Language. http://www.w3.org/XML/
13. XMark – An XML Benchmark Project. http://www.xml-benchmark.org/

http://www.springer.com/978-3-319-55704-5

Database Systems for Advanced Applications
DASFAA 2017 International Workshops: BDMS, BDQM, SeCoP, and DMMOOC, Suzhou, China, March 27-30, 2017, Proceedings
Bao, Z.; Trajcevski, G.; Chang, L.; Hua, W. (Eds.)
2017, XIV, 444 p. 179 illus., Softcover
ISBN: 978-3-319-55704-5