PAPER Special Section on Data Engineering and Information Management

History-Pattern Encoding for Large-Scale Dynamic Multidimensional Datasets and Its Evaluations*

Masafumi MAKINO†, ***Nonmember***, **Tatsuo TSUJI**††a), ***and*** **Ken HIGUCHI**††, ***Members***

SUMMARY In this paper, we present a new encoding/decoding method for dynamic multidimensional datasets and its implementation scheme. Our method encodes an *n*-dimensional tuple into a pair of scalar values even if *n* is sufficiently large. The method also encodes and decodes tuples using only *shift* and *and/or* register instructions. One of the most serious problems in multidimensional array based tuple encoding is that the size of an encoded result may often exceed the machine word size for large-scale tuple sets. This problem is efficiently resolved in our scheme. We confirmed the advantages of our scheme by analytical and experimental evaluations. The experimental evaluations were conducted to compare our constructed prototype system with other systems; (1) a system based on a similar encoding scheme called history-offset encoding, and (2) PostgreSQL RDBMS. In most cases, both the storage and retrieval costs of our system significantly outperformed those of the other systems.

key words: *tuple encoding, dynamic multidimensional datases, large scale datasets*

1. Introduction

In general, an *n*-dimensional data tuple can be mapped to the *n*-dimensional coordinate of a multidimensional array element. The coordinate can be further uniquely mapped to its position in the array by calculating the addressing function of the array. However, in a dynamic situation where new attribute values can emerge, a larger array is necessary to cover the new values, and the positions of the existing array elements must be recalculated according to the new addressing function.

An extendible array (e.g., [10]) can extend its size along any dimension without relocating any existing array elements. History-offset encoding [13] is a scheme for encoding multidimensional datasets based on extendible arrays. If a new attribute value emerges in an inserted tuple, a subarray to hold the tuple is newly allocated and attached to the extended dimension. A tuple can be handled with only two scalar values, *history value* of the attached subarray and *position* of the element in the subarray regardless of the dimension *n*. Dynamic tuple insertions/deletions can be performed without relocating existing encoded tuples due to the underlying extendible array

Many of the tuple encoding schemes, including history-offset encoding, use the addressing function of a multidimensional array to compute the position. However, there are two problems inherent in such encodings. First, the size of an encoded result may exceed the machine word size (typically 64 bits) for large-scale datasets. Second, the time cost of encoding/decoding in tuple retrieval may be high; more specifically, such operations require multiplication and division to compute the addressing function, and these arithmetic operations are expensive. To resolve these two problems without performance degradation, we present a *history-pattern* encoding scheme for dynamic multidimensional datasets and its implementation scheme called History-Pattern implementation for Multidimensional Datasets (HPMD). Our encoding scheme ensures significantly smaller storage and retrieval costs.

Our scheme encodes a tuple into a pair of scalar values <*history value*, *pattern*>. The core data structures for tuple encoding/decoding are considerably small. An encoded tuple can be a variable length record; the *history value* represents the extended subarray in which the tuple is included and also represents the bit size of the *pattern*. This approach enables the output file organization of the encoded results to be a sufficiently small sequential file**.** Additionally, our scheme does not employ the addressing function, hence avoiding *multiply* and *divide* instructions. Instead, it encodes and decodes tuples using only *shift* and *and/or* register instructions. This makes tuple retrieval significantly fast and further provides an efficient scheme for handling large-scale tuples whose encoded sizes exceed machine word size.

In this paper, first the related work is explained, then after history-offset encoding is outlined, our history-pattern encoding is presented. Next HPMD and an implementation scheme for large-scale tuple sets is described. Then the retrieval strategy using HPMD is explained. Lastly the implemented HPMD is evaluated and compared with other systems.

2. Related Work

Substantial research has been conducted on multidimensional indexing schemes based on the mapping strategy in which a multidimensional data point is transformed to a single scalar value. Such mapping strategies include a space-filling curve, such as the Z-curve [3] or the Hilbert curve [4], which enumerates every point in a multidimensional space.

Manuscript received July 1, 2015.
Manuscript revised October 3, 2015.
Manuscript publicized January 14, 2016.

†The author is with NTT Neo-meito, Corporation, Osaka-shi, 540–0026 Japan.
††The authors are with University of Fukui, Fukui-shi, 910–8507 Japan.

*This paper is an extended version of [17] presented in DASFAA 2015.

a) E-mail: tsuji@u-fukui.ac.jp (Corresponding author)

DOI: 10.1587/transinf.2015DAP0025

They preserve proximity, which suggests that points near one another in the multidimensional space tend to be near one another in the mapped one-dimensional space. This property of preserving proximity ensures better performance for range key queries against a dynamic multidimensional dataset; however, an important problem with these space-filling curves is that retrieval performance degrades abruptly in high-dimensional data spaces because of the required recursive handling. The UB-tree [2], [5], [6] maps a spatial multidimensional data point to a single value using a Z-curve; however, a UB-tree has the critical problem that its parameters (e.g., the range of attribute values) need to be properly tuned for effective address mapping [6]. This requirement restricts the usability and performance of the UB-tree.

In contrast to these approaches, in our history-pattern encoding scheme for an n dimensional tuple, encoding and decoding costs are both $O(n)$, even if n is very large, because these operations are performed using only *shift* and *and/or* machine instructions. Furthermore, the problem of the UB-tree approach is not present in our encoding scheme because of an unlimited extensibility of an extendible array.

The most common scheme for mapping multidimensional data points to scalar values [1] is to use a fixed-size multidimensional array. Much research, such as [7]–[9], has been performed using this scheme for the paging environment of secondary storage. The *chunk offset* scheme [9] is a well-known scheme in which multidimensional space is divided into a set of chunks; however, it is not extendible, and a new attribute value cannot be dynamically inserted.

Extendible arrays [10]–[13] provide an efficient solution to this non-extendible problem. In [10], Otoo and Rotem described a method for reducing the size of the auxiliary data structure for addressing array elements. [13] proposes the history-offset encoding, by which wider application areas such as in [16] can be developed.

One of the drawbacks inherent in the existing research that uses the addressing function of a multidimensional array is the time cost of decoding for tuple retrieval. The approaches presented in existing research require division operations using the coefficients of the addressing functions, which are very expensive. Such a drawback is not present with our history-pattern encoding, and the costs are significantly small. [14] presents the basic idea of the history-pattern encoding.

Another drawback of existing research is that the address computed by the addressing function may exceed machine word length. Some application areas for ho-encoding scheme are presented. [16] provides a labeling scheme of dynamic XML trees. In these applications, however, this address space saturation problem makes it difficult to handle large-scale datasets. One of the popular approaches against this problem is to use multiple precision functions such as in [18], but they would be much time consuming. Chunking array elements is a means to expanding the address space [9], [15], but it only delays saturation of the space. In [15], Tsuchida et al. vertically partitioned the tuple set to reduce dimensionality; however, overhead emerges that increased storage costs.

This paper presents an implementation scheme of history-pattern encoding in order to resolve or alleviate these two drawbacks and provides an efficient implementation for large-scale multidimensional datasets as will be confirmed by the analytical and experimental evaluations. As far as we know, there is no research similar to ours.

3. History-Pattern Encoding

3.1 Preceding Encoding Model

As a preceding encoding model, we first introduce *history-offset* encoding [13] based on an extendible array.

An n-dimensional extendible array A (see Fig. 1) has a *history counter* h, *history table* H_i, and a *coefficient table* C_i for each extendible dimension i ($i = 1, \ldots, n$). H_i memorizes the extension history of A. If the current size of A is $[s_1, s_2, \ldots, s_n]$, for an extension that extends along dimension i, an $(n-1)$-dimensional subarray S of size $[s_1, s_2, \ldots, s_{i-1}, s_{i+1}, \ldots, s_{n-1}, s_n]$ is attached to dimension i. Then, h is incremented by one and memorized on H_i. In Fig. 1, the array size is $[3, 3]$ when h is 4. If the array is extended along dimension 1, a one-dimensional subarray of size 3 is attached to dimension 1, and h is incremented to 5 (and is held in $\mathrm{H}_1[3]$). Each history value can uniquely identify the corresponding extended subarray.

As is well known, element $(i_1, i_2, \ldots, i_{n-1})$ in an $(n-1)$-dimensional fixed-size array of size $[s_1, s_2, \ldots, s_{n-1}]$ is allocated on memory using an addressing function such as:

$$f(i_1, \ldots, i_{n-1}) = s_2 s_3 \ldots s_{n-1} i_1 + s_3 s_4 \ldots s_{n-1} i_2 + \ldots + s_{n-1} i_{n-2} + i_{n-1} \quad (1)$$

We call $<s_2 s_3 \ldots s_{n-1}, s_3 s_4 \ldots s_{n-1}, \ldots, s_{n-1}>$ a *coefficient vector*. The vector is computed at array extension and is held in *coefficient table* C_i of the corresponding dimension. Specifically, if $n = 2$, the subarrays are one-dimensional and $f(i_1) = i_1$. Therefore, the coefficient tables can be void if n is less than 3 as in Fig. 1.

Using the above three types of auxiliary tables, history-offset encoding (denoted as *ho-encoding* in the following) of array element $e(i_1, i_2, \ldots, i_n)$ can be computed as $<h, offset>$, where h is the history value of the subarray in which e is included and *offset* is the offset of e in the subarray computed by (1); e.g., element $(3, 2)$ is encoded to

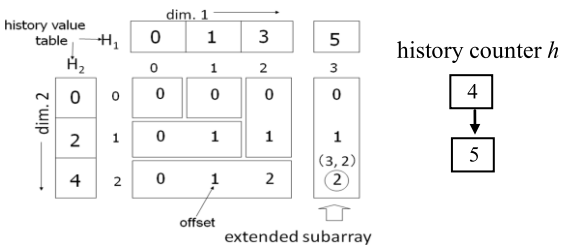


Fig. 1 Two-dimensional extendible array

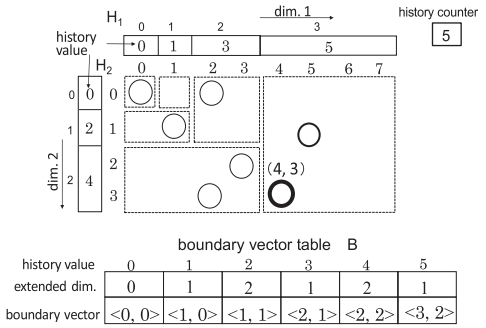


Fig. 2 Data structures for hp-encoding

<5, 2>.

Conventional schemes for storing fixed size arrays using an addressing function like (1) above do not support dynamic extension of an array. Therefore addition of a tuple including new attribute value is impossible if the number of the existing attribute values attains to the corresponding dimension size. In this case, a larger fixed size array is created and all the elements in the old array should be re-encoded according to the new addressing function provided for the newly created array. However, in the history-offset encoding scheme, a new subarray is dynamically created to hold the new tuple, and the existing elements in the array are not necessary to be re-encoded.

3.2 Data Structures for History-Pattern Encoding

Figure 2 illustrates the required data structures for the history-pattern encoding (denoted as *hp-encoding* in the following). Unlike ho-encoding, when logical extendible array A in hp-encoding extends its size, a fixed-size subarray equal to the size of the current A in every dimension is attached to the extended dimension. The data structures for A consist of the following two types of tables preserving extension information.

(History table) For each dimension i ($i = 1, \ldots, n$), history table H_i is maintained. Each history value h in H_i represents the extension order of A along the i-th dimension. 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. This range is covered by the subarray along the i-th dimension attached to A at the extension when the history counter value is h. For example, as shown in Fig. 2, since $\mathrm{H}_1[3]$ is 5, the subscript 3 of H_1 corresponds to the subscript range from 4 to 7 of the first dimension of A.

(Boundary vector table) The *boundary vector table* B is a single one-dimensional array whose subscript is a history value h. It plays an important role for hp-encoding. Each element of B maintains the extended dimension and the boundary vector of the subarray when the history counter value is h. More specifically, the boundary vector represents the past size of A in each dimension when the history counter value is h. For example, the boundary vector in B[3] is <2, 1>; therefore, the size of A at the history counter value 3 is $[2^2, 2^1] = [4, 2]$. Together with the boundary vectors, B also maintains the dimension of A extended at the given history counter value. A includes only the element $(0, 0, \ldots, 0)$ at its initialization, and the history counter is initialized to 0. B[0] includes 0 as its extended dimension and $<0, 0, \ldots, 0>$ as its boundary vector.

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

(Logical size and real size) In hp-encoding, A has two size types, i.e., real and logical. Assume that the tuples in n-dimensional dataset M are converted into 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; for example, in Fig. 2, the real size is $[6, 4]$, whereas the logical size is $[8, 4]$. In Fig. 3 later, the real size is $[6, 6]$, and the logical size is $[8, 8]$. Note that in ho-encoding logical and real size are the same.

(Array extension) Suppose that a tuple whose k-th attribute value emerged for the first time is inserted. This insertion increases the real size of A in dimension k by one. If the increased real size exceeds the current logical size $2^{b(s)}$, A is logically extended along dimension k. That is, current history counter value h is incremented by one, and this value is set to $\mathrm{H}_k[b(s+1)]$. Moreover, the boundary vector in B[h] is copied to B[$h + 1$] and dimension k of the boundary vector is incremented by one; k is set to the *extended dimension* slot in B[$h + 1$], as illustrated in Fig. 2.

Note that h is one-to-one correspondent with its boundary vector in B[h]; this uniquely identifies the past (logical) shape of A when the history counter value is h. To be more precise, for 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 is $[2^{b_1}, 2^{b_2}, \ldots, 2^{b_n}]$. For example, in Fig. 2, 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 subarray attached to A at extension when the history counter value was $h-1$. This subarray will be called the *principal subarray* on dimension k at h. For example, in Fig. 2, the *principal subarray* on dimension 2 at $h = 4$ is the subarray specified by [0..3, 2..3].

3.3 History-Patten Encoding/Decoding

Using the data structures described in Sect. 3.2, 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 of the coordinate. The history tables H_i ($i = 1, \ldots, n$) and the boundary vector table B are used for the encoding. The

history value h for I is determined as $\max\{H_k[b(i_k)] \mid k = 1, \ldots, n\}$, where $b(i_k)$ is the bit size of the subscript i_k in I. For each history value h, the boundary vector 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 by placing in descending order of dimensions on the storage for p from the lower to the higher bits of p. The storage for p can be one machine word length; typically, 64 bits.

For example, consider the array element (4, 3) in Fig. 2. According to the above encoding procedure, $H_1[b(4)] > H_2[b(3)]$ because $H_1[b(4)] = H_1[b(100_{(2)})] = H_1[3] = 5$ and $H_2[b(3)] = H_2[b(11_{(2)})] = H_2[2] = 4$. So h is proved to be $H_1[3] = 5$, and element (4, 3) is known to be included in the *principal subarray* (Sect. 3.2) on dimension 1 at history value 5. Therefore, the boundary vector to be used is <3, 2> in $B[5]$. So the subscript 4 of the element (4, 3) forms the upper 3 bits of p as $100_{(2)}$ and the subscript 3 of the element forms the lower 2 bits of p as $11_{(2)}$. Therefore, p becomes $10011_{(2)} = 19$. Eventually, the element (4, 3) is encoded to <5, 19>. Generally, the bit size of the history value h is much smaller than that of pattern p.

Conversely, to decode the encoded pair $<h, p>$ to the original n-dimensional coordinate $I = (i_1, i_2, \ldots\ldots, i_n)$, first 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. For example, consider the encoded pair $<h, p> = <5, 19>$. The boundary vector in $B[h]$ is <3, 2>, so $p = 10011_{(2)}$ can be divided into $100_{(2)}$ and $11_{(2)}$. Therefore <5, 19> can be decoded to the coordinate (4, 3).

Note that as in history-offset encoding, re-encoding existing array elements is not necessary even if a tuple is inserted including a new attribute value which causes the array to be extended.

In both of ho-encoding and hp-encoding, dynamic addition of a new attribute is possible with a very small cost. When a new attribute is added dynamically to an n-dimensional extendible array, the array becomes $n + 1$-dimesional. The array before the extension is specified by the subscript 0 of the new dimension (the $n+1$-th dimension) of the extended subarray. A history table for the new dimension is prepared and initialized. The detail can be found in [18]. This dynamic addition of a new attribute will not be treated in this paper.

3.4 Hp-Property

From the construction procedure of the boundary vector table B in Sect. 3.2, the following simple, but important property for our hp-encoding can be known. This property will be called *hp-property* in the following.

[Property 1 (hp-property)] Let $<h, p>$ be an encoded history-pattern of a tuple. h is the total sum of the element values of the boundary vector in B[h] and represents the bit size of the coordinate pattern p for an arbitrary element in the subarray at h.

In our hp-encoding, the favorable property of an extendible array is reflected in the hp-property above. Namely, for the tuples inserted in the subarrays created at the early stage of array extension occupy smaller storage. Consequently, the size of p can be much smaller than in the usual case where each subscript value occupies fixed size storage. It should be noted that the boundaries among the subscript bit patterns in p can be flexibly set to minimize the size of p.

Moreover, the hp-property states that h represents the bit pattern size of p. This simple property together with *shift* and *and/or* register instructions for encoding/decoding makes our encoding scheme to be applied for implementation of large scale multidimensional datasets efficiently with no significant overhead. From this property even if the bit size of p is doubled, h increases only by 1. For example, if the p's current bit size is 255 bits, h is only 1 byte. Therefore our hp-encoding scheme can provide unlimited (logical) history-pattern space size for large and high dimensional dataset with a very small additional storage cost for keeping h.

3.5 Comparison of the Two Encoding Schemes

In this section, we compare hp-encoding with ho-encoding. Let the real size of the extendible array be $[s_1, s_2, \ldots, s_n]$ for both encodings.

(1) Storage Costs for Core Data Structures

For ho-encoding, the core data structures are the history tables and the coefficient tables presented in Sect. 3.1; for hp-encoding, they are the history tables and the boundary vector table presented in Sect. 3.2. These data structures guarantee the extensibility of an extendible array.

In ho-encoding, let m and c be the fixed size in bytes of the max history value and coefficient values in the coefficient tables respectively. We estimate the storage cost of core data structures for ho-encoding as follows:

(a) History tables: $m * (s_1 + s_2 + \ldots\ldots + s_n)$

(b) Coefficient tables: $c * (n - 2)(s_1 + s_2 + \ldots\ldots + s_n)$ $(n > 2)$

For hp-encoding, the storage cost is as follows:

(c) History tables: $\lceil \log_2 s_1 + 1 \rceil + \lceil \log_2 s_2 + 1 \rceil + \ldots\ldots + \lceil \log_2 s_n + 1 \rceil$

(d) Boundary vector table:

$$n * (\lceil \log_2 s_1 \rceil + \lceil \log_2 s_2 \rceil + \ldots\ldots + \lceil \log_2 s_n \rceil + 1)$$

Table 1 shows the seven extendible arrays used in the

Table 1 Used extendible arrays

	number of dimensions	dimension size
(5-1)	5	[8, 8, 8, 8, 8]
(5-2)	5	[8, 16, 32, 64, 128]
(5-3)	5	[512, 512, 512, 512, 512]
(10-1)	10	[8, 8, 8, 8, 8, 8, 8, 8, 8, 8]
(10-2)	10	[2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]
(10-3)	10	[128, 128, 128, 128, 128, 128, 128, 128, 128, 128]
(10-4)	10	[256, 256, 256, 256, 256, 256, 256, 256, 256, 256]

Table 2 Max. history value

	ho-encoding	hp-encoding
(5-1)	36(1)	15(1)
(5-2)	244(1)	25(1)
(5-3)	2556(2)	45(1)
(10-1)	71(1)	30(1)
(10-2)	2037(2)	55(1)
(10-3)	1271(2)	70(1)
(10-4)	2551(2)	80(1)

Remark: "( )" denotes the size in byte to represent the history value

Table 3 Storage costs for the core data structures (bytes)

	history tbls ho-encoding	history tbls hp-encoding	coeff. tbls ho-encoding	boundary vec. tbl hp-encoding
(5-1)	40	20	240	80
(5-2)	248	30	2232	130
(5-3)	5120	50	30720	230
(10-1)	80	40	2560	310
(10-2)	4092	65	49104	560
(10-3)	2560	80	81920	710
(10-4)	5120	90	-	810

Remark: "-" denotes that the offset size overflows 64 bits

comparison of ho-encoding and hp-encoding, For both encoding schemes, Table 2 shows the maximum history values for each extendible array in Table 1, when it is extended to the specified dimension size. We can notice that the history value of hp-encoding is suppressed very small even if the dimension sizes become large due to the hp-property stated in the previous section.

Table 3 shows the storage costs for history tables, coefficient tables (ho-enconding) and boundary vector table (hp-encoding) computed according to the equations in (a)~(d) above. We can observe that the coefficient tables in ho-encoding become very large when the dimension sizes become large. Note that in the array (10-4), the coefficient vector size overflows 64 bits, so the *offset* space of ho-encoding overflows 64 bits and ho-encoding is not applicable.

(2) Encoding and Decoding Performance
In hp-encoding, encoding/decoding are performed using only *shift* and *and/or* register instructions. These instructions do not refer to memory addresses, so encoding/decoding can be executed quickly compared with ho-encoding in which multiplications and divisions are required. In Sect. 7.3, this will be experimentally confirmed in tuple access and retrieval times.

4. Implementation of History-Pattern Encoding

4.1 Implementation of Core Data Structures

HPMD is an implementation scheme based on hp-encoding for n-dimensional dataset M. In addition to the core data structures presented in Sect. 3.2, HPMD includes the following additional data structures:

(1) CVT_i $(1 \le i \le n)$ is implemented as a B^+ tree. The key value is an attribute value of dimension i; the data value is the corresponding subscript of the extendible array.

Table 4 Insertion of the two-dimensional tuples

	inserted tuples	history value h	coordinate pattern p	boundary vector
1	<*d, p*>	0	.	<0,0>
2	<*b, p*>	1	1.	<1,0>○
3	<*b, q*>	2	1.1	<1,1>○
4	<*c, q*>	3	10.1	<2,1>○
5	<*f, q*>	3	11.1	<2,1>
6	<*e, q*>	4	100.1	<3,1>○
7	<*f, t*>	5	011.10	<3,2>○
8	<*a, r*>	5	101.11	<3,2>
9	<*a, p*>	4	101.0	<3,1>
10	<*a, u*>	6	101.100	<3,3>○
11	<*b, r*>	5	001.11	<3,2>
12	<*e, s*>	6	100.101	<3,3>
13	<*d, r*>	5	000.11	<3,2>

Remark 1: the leftmost number represent the insertion orders of the tuples **Remark 2:** "." in coordinate bit pattern is a separator between subscripts **Remark 3:** ○ denotes that the insertion of the tuple causes the extension of the logical size of the extendible array

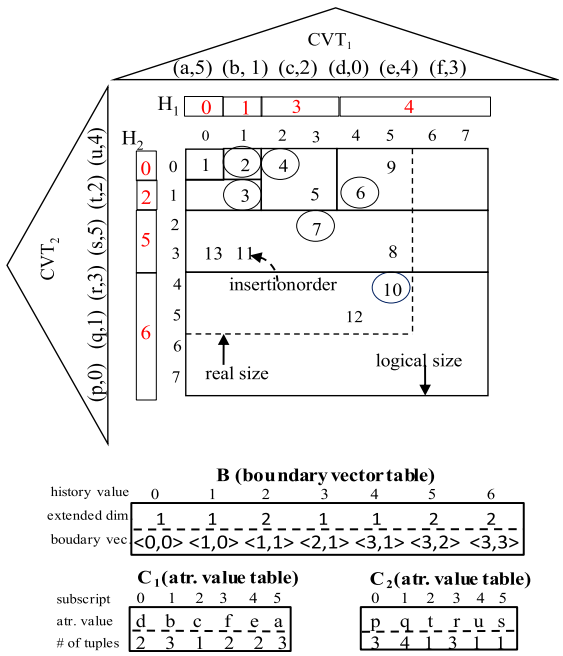


Fig. 3 HPMD data structure

(2) C_i $(1 \le i \le n)$ is a one-dimensional array serving as the *attribute value table*. If attribute value v is mapped by CVT_i to subscript k, the k-th element of C_i keeps v. The element further includes the number of tuples in M, whose attribute value of dimension i is v. This number is used to detect the retrieval completion, which will be described in Sect. 6.2.

Note that M can be also implemented based on ho-encoding using (1) and (2) above. We call this implementation scheme as HOMD.

Table 4 shows an example in which two-dimensional tuples are successively inserted. Figure 3 shows the constructed HPMD using the inserted tuples in Table 4. By carefully inspecting the Table 4 and Fig. 3, we can trace the

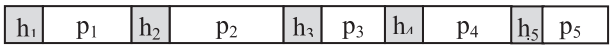


Fig. 4 Storing variable length encoded tuples sequentially in ETF

change in the related data structures and the produced encoded results generated by each insertion.

4.2 Output File Organization of Encoded Results

In ho-encoding, as discussed in Sect. 3.1, both the history value and offset of an encoded tuple occupies fixed-size storage, which degrades storage performance. In contrast, in our hp-encoding scheme, we adopt variable length storage scheme according to the pattern size of the encoded results based on the *hp-property* given in Sect. 3.4.

We can ensure the hp-property by observing h and p in Table 4 and boundary vector table B in Fig. 3. By this property, the history value h can be used as a header of coordinate bit pattern p. It represents p's bit size. Therefore, $<h, p>$ can be treated as a variable length record with size known by h. The hp-property enables to store encoded results in a sequential output file called *Encoded Tuple File* (ETF), as illustrated in Fig. 4. The encoded results are stored sequentially in the insertion order of the tuples, similar to a conventional RDBMS. Compared with the size of p, the size of h is sufficiently small, and its size should be fixed. Typically, the size of h is 1 byte and can specify p up to 255 bits.

5. Implementation for Large Scale Datasets

Unfortunately, in hp-encoding, the history-pattern space often exceeds machine word length for high-dimensional and/or large volume datasets. In this section, we provide a scheme to handle such a large history-pattern space with minimal degradation in encoding/decoding speed.

5.1 Extending History-Pattern Space

To handle large-scale tuple datasets using hp-encoding, the coordinate bit patterns can range over multiple machine words to eliminate the pattern size limitation. For example, Fig. 5 shows the layout for a 162-bit coordinate pattern according to boundary vector <25, 16, 13, 23, 15, 8, 6, 10, 7, 20, 15, 4> of 12 dimensions; this requires three 64-bit words. Compared with a bit pattern within a single word, no storage cost overhead arises with this multiword bit pattern. Furthermore, encoding to and decoding from this multiword bit pattern do not cause significant overhead, since they can be performed by using only *shift, mask, or* register instructions as in a single bit pattern with a little *cut and paste* cost. We omit the details. In contrast, overhead caused by using a multi-precision library such as [19] in ho-encoding would significantly degrade retrieval performance. Note that the *hp-property* introduced in Sect. 3.4 is also guaranteed for multiword bit patterns.

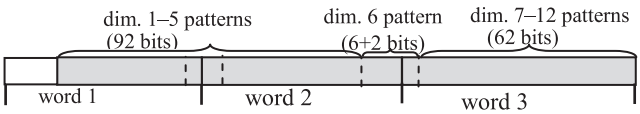


Fig. 5 Layout for 162-bit coordinate pattern

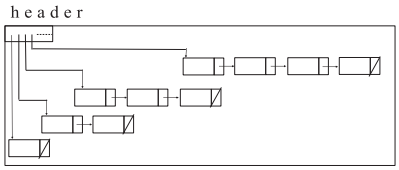


Fig. 6 Node block lists in ETF

5.2 Further Storage Optimizations

Here, we present two optimization strategies for storing encoded results in ETF.

(1) Sharing history value

Due to the *hp-property,* the set of the encoded $<h, p>$ pairs can be partitioned into the subsets depending on h. The pairs in the same subset have the same history value h, so h can be shared among these pairs and the bit size of their pattern p equals to h. Thus the p's of the same h are stored in the same node block list as in Fig. 6. Since h is one-to-one correspondent with its subarray, the node block list keeps all patterns of the elements in the subarray. If the size of h is one byte, and the total number of tuples is m, a total of $m-1$ bytes can be saved by this optimization.

(2) Multi-boundary vector

For the multiword bit patterns described in Sect. 5.1, two types of arrangement of the bit patterns in a node block are considered; (a) *byte-alignment*, (b) *word-alignment*.

In (a), storage cost can be saved, but retrieval cost would increase. But, the situation is just converse in (b). We present the following method to avoid the retrieval overhead inherent in (a) but take advantage of its storage cost savings.

Assume that the machine word occupies w bytes, and p bytes are necessary to store a single coordinate bit pattern. Let l be the least common multiple of w and p, and let bv be the boundary vector for a single coordinate bit pattern described in Sect. 5.1. Multi-boundary vector mbv is a set of single boundary vectors bvs and can be obtained by recalculating and arranging bv sequentially l/p times. Here we omit the description of its details. Note that mbv can be used as if it were a single boundary vector. Using mbv, l/p single coordinate bit patterns can be stored consecutively in a node block in ETF by byte alignment, while they can be retrieved by word alignment.

A storage scheme based on the above multi-boundary vector increases the size of the boundary vector table; however, the size is negligibly small compared with ETF size. Consequently, this multi-boundary vector further

contributes to generate ETFs compactly without degradation of retrieval performance. We refer to HPMD based on the *multiword bit pattern scheme* described in Sect. 5.1 simply as HPMD, and the HPMD based on the *multi-boundary vector* using node block lists as M-HPMD.

6. Tuple Retrieval

6.1 History Value Dependency

We can notice the following important property in both HPMD and M-HPMD.

[Property 2] Let h be the history value of the principal subarray PS (see Sect. 3.2) of the subscript k on dimension i. The array elements with subscript k on dimension i are included only in PS or the subarrays with history values greater than h and extended along the dimension other than dimension i.

This property is shown in Fig. 7. The dotted line represents the real size of the extendible array, and the grey colored parts are the candidates of retrieval. We can see that it is not necessary to decode all the elements in ETF, but only the grey colored parts due to the above property. An element in the non-grey subarrays can be checked by its history value, and can be skipped without decoding the pattern part. The total size of the grey parts depends on the history value of the principal subarray of the subscript to be retrieved. This leads to the following property.

[Property 3 (history value dependency)] The subarrays to be decoded for the retrieval of an attribute value v depends on the history value corresponding to v.

6.2 Tuple Retrieval

In HPMD, like in a conventional RDBMS all tuples in ETF should be searched sequentially. Nevertheless, according to Property 2, non-candidate tuples can be skipped without inspecting bit pattern part p by only examining the history value part. In M-HPMD, each tuple is classified in terms of its history value and is stored in the corresponding node block list. Therefore, even the check of a history value can be avoided in the candidate node block lists.

Let *age* be an integer attribute. For a single value retrieval, such as $age = 20$, first the specified attribute value is searched in CVT_{age} to obtain its subscript value i. Both in HPMD and M-HPMD, the history values for candidates of retrieval are determined according to Property 2. If a candidate $<h, p>$ is encountered, p is decoded to get the subscript of dimension *age*. If it is i, $<h, p>$ is included in the retrieval results.

For a range value retrieval, such as $10 \leq age < 20$, the set of subscripts covered by the range is obtained by searching the sequence set of CVT_{age}. Based on the obtained subscript set S, the set of the history values for candidates of retrieval are determined according to Property 2. In HPMD, if a candidate $<h, p>$ is encountered, p is decoded to get the subscript i of dimension *age*. For the attribute value table (see Sect. 4.1(2)) C_{age}, if $10 \leq C_{age}[i] < 20$, $<h, p>$ is included in the retrieval results. Note that while single value retrieval requires only subscript matching, range value retrieval requires references to the attribute value table.

Note also that in both HPMD and M-HPMD, before checking all candidate tuples, when the number of matched tuples reaches the "number of tuples" kept in the related attribute value table, the retrieval can be terminated.

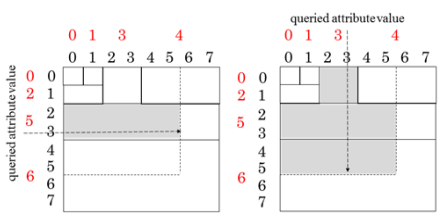


Fig. 7 Tuple retrieval in HPMD

7. Evaluation Experiments

In this section, performance evaluations are shown for HPMD and M-HPMD described in Sect. 5.2 on the implemented prototype system. These are compared with HOMD and PostgreSQL, which is one of the conventional RDBMS. They all output the tuples sequentially to the output file, and the tuple retrieval is also sequentially performed.

7.1 Evaluation Environment

Construction times, storage sizes, and retrieval times are measured under the following 64 bits computing environment.

CPU: Intel Core i7 (2.67GHz), Main Memory: 12GB,
OS: CentOS5.6 (LINUX),
PostgreSQL: Version 8.4.4 (64-bit version)

In the measurement of the retrieval times for PostgreSQL, the *timing command* was used. The command invokes the LINUX system call gettimeofday() and we also used this system call for HOMD, HPMD and M-HPMD. The retrieval time in these implementations includes the time to get the decoded tuples that satisfy the query condition. To suppress the performance deterioration caused by transaction processing in PosgreSQL, the transaction isolation level is set to the lowest level.

7.2 Evaluation Using Large Scale Dataset

The LINEITEM table (Table 5) in TPC-H benchmark data [20] is employed. The size of the input tuple file generated by TPC-H is about 2.43 GB in csv formatted including 23,996,604 tuples. Actually L_COMMENT column is variable length text. Since such data type is currently not supported in our implementations, the column was dropped

Table 5 LINEITEM Table

dim.	attribute name	type	cardinality
1	L_ORDERKEY	int	6000000
2	L_PARTKEY	int	800000
3	L_SUPPKEY	int	40000
4	L_LINENUMBER	int	7
5	l_QUANTITY	double	50
6	L_EXTENDEDPRICE	double	1079204
7	L_DISCOUNT	double	11
8	L_TAX	double	9
9	L_RETURNFLAG	char[1]	3
10	L_LINESTATUS	char[1]	2
11	L_SHIPDATE	char[10]	2526
12	L_COMMITDATE	char[10]	2466
13	L_RECEIPTDATE	char[10]	2555
14	L_SHIPINSTRUCT	char[25]	4
15	L_SHIPMODE	char[10]	7
	L_COMMENT	varchar[44]	15813794

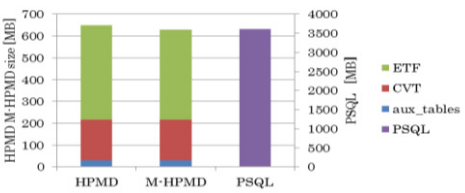


Fig. 8 Storage cost

Table 6 Construction cost (sec)

HPMD	M-HPMD	PSQL
159.95	180.15	224

out.

Note that HOMD cannot implement such large table due to the history-offset space overflow.

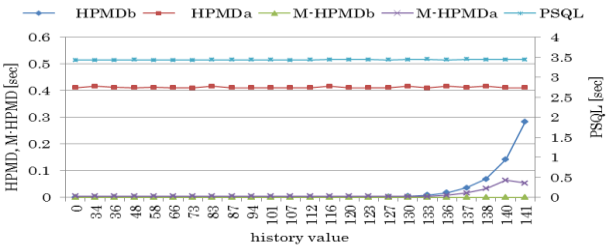
(1) Storage Cost

The total storage required to store a multidimensional dataset includes data structures for encoding/decoding shown in Fig. 3 and ETF to store the encoded tuples. In HPMD, ETF is a sequential file and in M-HPMD, it is a file of node block lists. Figure 8 shows the total required storage sizes for HPMD, M-HPMD, PostgreSQL (denoded by PSQL in the following). In HPMD and M-HPMD the breakdown of the total size is shown. "aux_tables" are the history tables, boundary vector table and attribute value tables in Fig. 3. The maximum history value in the constructed HPMD and M-HPMD was 141 (3 machine words).

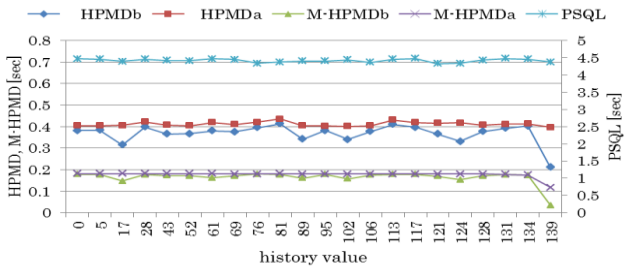
As can be seen in Fig. 8, the total sizes for HPMD and M-HPMD are about one-sixth of the size for the PSQL. This indicates that our hp-encoding scheme realizes significant reduction of the storage cost. In M-HPMD the size of ETF is 5% smaller than that in HPMD due to the sharing history value in M-HPMD stated in Sect. 5.2. It can be noted that while the size for PSQL is about 1.6 times larger than that for the *csv f*ormatted file, the size for HPMD or M-HPMD is about 30% of the *csv* file size.

(2) Construction Time

Table 6 shows the times spent for constructing databases from the *csv* source file. The spent times for



(a) dim. 1 (L_ORDERKEY)



(b) dim. 6 (L_EXTENDEDPRICE)

Fig. 9 Retrieval times of single value queries

HPMD and M-HPMD are about 71% and 80% of that for PSQL. The difference owes to the reduction of output I/O cost; output ETF file size of HPMD and M-HPMD is far less than the output file size of PSQL. It can be observed that the time spent for M-HPMD is 13% larger than that for HPMD. This owes to the time of M-HPMD spent for construction of node block lists.

(3) Retrieval time

(3-1) Retrieval for single value queries

Figure 9 shows the retrieval times of single value queries for LINEITEM table. The left side scale is for HPMD and M-HPMD and the right one is for PSQL. Each history value on the horizontal axis represents the leftmost subscript of the principal subarray (see Sect. 3.2) on the dimension 1 and 6. The retrieval time for the attribute value corresponding to the subscript was measured. We adopt the dimensions since the larger cardinality can better exhibit the properties of our schemes. As was mentioned in Sect. 5.2, both in HPMD and M-HPMD the retrieval can be terminated without checking all the candidate tuples in ETF by using "num. of tuples" in the attribute value table (See Fig. 3). The measurement was also done in the case all the candidate tuples are checked without using "num. of tuples". We will denote this case as HPMDa and M-HPMDa, and the case using "num. of tuples" as HPMDb and M-HPMDb. The denotations HPMD and M-HPMD will be used for both cases.

In PSQL and HPMDa, the retrieval times are nearly constant irrespective of the queried attribute values, since all the tuples are scanned through. The average times of HPMDa in dim. 1 and dim. 6 are 8.33 and 10.73 times faster than that of PSQL, respectively. In contrast, in M-HPMD only the candidate tuples are scanned and decoded. Therefore the history value dependency described in Sect. 6.1 can be better observed in M-HPMD than in HPMD as shown in

Table 7 Max. and min. ratios of retrieval times

	max. ratio	min. ratio
HPMDb/PSQL (dim. 1)	8.24%	0.000175%
M-HPMDb/PSQL (dim. 1)	0.000904%	0.000175%
HPMDb/PSQL (dim. 2)	10.90%	9.30%
M-HPMDb/PSQL (dim.2)	4.97%	4.30%
HPMDb/PSQL (dim. 6)	9.45%	4.82%
M-HPMDb/PSQL (dim. 6)	4.15%	0.79%
HPMDb/PSQL (dim. 13)	6.68%	6.60%
M-HPMDb/PSQL (dim. 13)	3.30%	3.17%

Table 8 Attribute value ranges used in the experiment

	range R1	range R2	range R3
dim. 1	100,000 – 820,000	1,000,000 – 3400,000	4,000,000 – 8,800,000
dim. 2	100,000 - 124,000	200,000 - 280,000	500,000 - 660,000
dim. 3	10,000 - 11,200	20,000 - 24,000	30,000 - 38,000
dim. 5	2 - 3	5 - 9	10 - 19

Fig. 9 (b). In M-HPMD the time decreases at the maximum history value in both dimension 1 (141) and dimension 6 (139).

The principal subarrrays corresponding to these history values are located at the end of the logical extendible array. So, the reason of the decrease is that the real size of the logical extendible array in these dimensions is less than that of the logical size, so the tuples in these subarrays do not fill out its logical space.

It can be observed that using "num. of tuples" in the attribute value table of dim. 1 is effective. Since the cardinality of dim. 1 is very large and its attribute values are uniformly distributed, the number of tuples of each attribute value is very small; i.e., less than 10. In HPMD, the attribute values of the same dimension are converted to the dimension subscripts in the ascending order, namely the earlier an attribute value appears, the smaller subscript is assigned to the attribute value. Since the encoded results are stored sequentially in the ETF file, the attribute values covered by the smaller history values are stored earlier in the ETF file, so the number of the tuples satisfying the query quickly attains to "num. of tuples". In M-HPMD, the tuples of the same history value can be directly accessible and in dim. 1 they can be confined in a single node block, so the retrieval times are almost 0. For dim. 6, the cardinality is smaller than that of dim. 1, and the attribute values are not so uniformly distributed as those of dim. 1, the above advantages for dim. 1 is decreased in both HPMDb and M-HPMDb as can be observed in Fig. 9 (b).

For HPMDb and M-HPMb the maximum and minimum ratios of the retrieval times to those of PSQL are shown in Table 7. It can be seen that the ratios are under 11%, and that the maximum retrieval times of M-HPMD are about a half of that of HPMD. This proves the benefit of M-HPMD described in Sect. 6.2.

(3-2) Retrieval for Range Queries

Table 8 shows the ranges of the attribute values for the range queries, on which the retrieval times were measured. The *selectivities* of the ranges R1, R2 and R3 on each dimension are about 3%, 10%, 20% respectively. Figure 10 shows the retrieval times for the range queries on dimension 1 and 6. It can be known from Fig. 10 (a) that the retrieval times for HPMDa is much larger than those of HPMDb. This also dues to the reason described in (3-1). For HPMDb and M-HPMDb, Table 9 shows the ratios of the retrieval times to those of PostgreSQL on the range queries in Table 8. It should be noted that in both HPMDb and M-HPMDb, dimension subscripts are assigned in ascending order. So, the subscript range corresponding to its attribute value range may often spread over wider than the attribute value range. This might alleviate the benefits of our HPMD and M-HPMD in some degree; as can be observed in Table 9, the performance on dim. 2 are degraded.

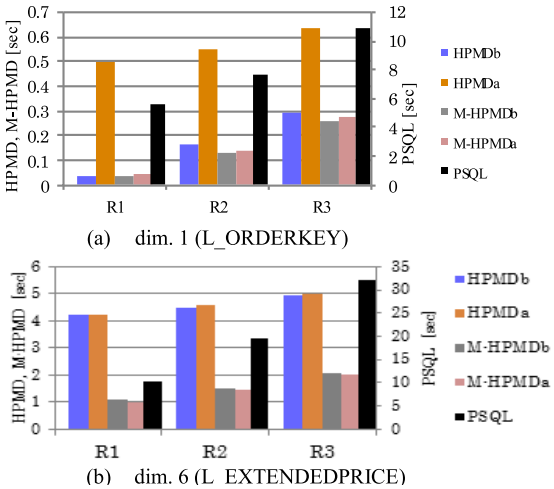


Fig. 10 Retrieval times of range queries

Table 9 Ratios of retrieval times for range queries

	range R1	range R2	range R3
HPMDb/PSQL (dim. 1)	1.40%	2.28%	2.76%
M-HPMDb/PSQL (dim. 1)	1.75%	2.82%	3.45%
HPMDb/PSQL (dim. 2)	20.64%	12.16%	8.47%
M-HPMDb/PSQL (dim.2)	11.81%	8.1%	6.42%
HPMDb/PSQL (dim. 3)	11.06%	7.29%	4.99%
M-HPMDb/PSQL (dim. 3)	6.98%	5.71%	4.50%
HPMDb/PSQL (dim. 5)	5.86%	4.27%	3.87%
M-HPMDb/PSQL (dim. 5)	4.14%	4.03%	4.02%

Table 10 Used dataset

# of dims	# of tuples	attribute type	cardinality
5	5,000,000	all integer	all 512

7.3 Comparison of HOMD and HPMD

The *ho-encoding* is a competitor of our *hp-encoding*. Its HOMD implementation cannot deal with large scale datasets without considerable degradation of retrieval performance. In this section, by using a moderate-scale dataset whose history-offset space is within the machine word size, we briefly compare the performance among HOMD, HPMD and PSQL. The dataset is artificially created. The specifications are shown in Table 10.

Table 11 Storage cost and construction time

	HOMD	HPMD	M-HPMD
ETF (kbytes)	50,000	34,843	30,669
const.time (sec)	20.65	7.04	12.45

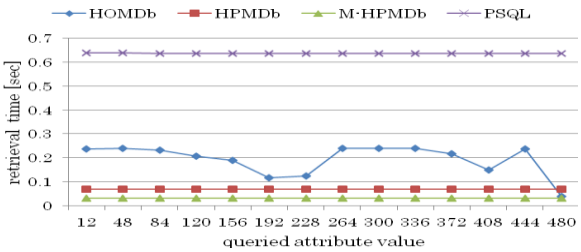


Fig. 11 Retrieval times for single value queries on the 5th dim.

Tuples in the dataset is uniformly distributed in the history-offset or history-pattern space to evaluate the basic performance of each scheme. In HOMD the maximum history value and offset size are 2556 and 36 bits respectively, while in HPMD the maximum history value and coordinate pattern size are 45 and 45 bits respectively. Therefore in HOMD, the size of history value and offset value are fixed in 2 and 8 bytes respectively, and in HPMD, the history value size is fixed in 1 byte and that of coordinate pattern is variable according to the history value. Both in HOMD and HPMD, the encoded results are output sequentially in the ETF files.

Table 11 shows the storage costs for ETF and database construction time. We can observe that the ETF size in HOMD is much larger than that of M-HPMD. This owes to the advantage of the M-HPMD implementation, in which the encoded tuples in ETF is variable length records arranged in byte alignment. The construction time in HOMD is about 2.9 times larger than that of HPMD due to the required encoding time.

Figure 11 shows the retrieval times for single value queries on the 5th dimension. We can see that the retrieval times in HPMDb and M-HPMDb are almost constant irrespective of queried attribute values like in PSQL. This is because that the tuples are uniformly distributed over the attribute domain. On the other hand, the retrieval times for HOMDb are depending on the queried attributes in spite of the uniform tuple distribution due to the attribute value sensibility of HOMD.

8. Consideration on Order Preserving Property

It should be noted that our hp-encoding does not always guarantee the minimal storage cost. For example, consider the element (5, 1) in Fig. 2. Since the element can be known to be included in the subarray of history 5, the boundary vector <3, 2> is used for encoding (5.1). Therefore, 5 and 1 is encoded to $101.01_{(2)}$. Note that the subscript 1 of the second dimension is prefixed by the redundant 0. Such kind of redundancy is inevitable in our hp-encoding.

On the other hand, such redundancy makes our hp-encoding very simple and fast. In particular, the hp-property in Sect. 3.4 affords an efficient sequential output file organization of the encoded tuples. Moreover, very small fixed size history value can treat a long sized variable length pattern. This makes a pattern to be associated with its history value with additional small storage cost and enables HPMD to preserve the input order of tuples in ETF. However, in M-HPMD this ordering information is lost because the encoded patterns are separated from their history values and classified according to their history values irrespective of their input order (see Fig. 6). Namely, the lower storage cost obtained by sharing a history value in M-HPMD is realized at the sacrifice of losing the order-preserving property.

This order preserving property is essential for some applications such as sensor stream data, in which receiving time order is necessary for analysis, or document processing, in which preserving sentence order in a document is required [16]. Important work includes formulation and analysis of these order preserving and non-order preserving implementation schemes for hp-encoding or other similar encodings.

9. Conclusion

We have presented a novel encoding/decoding scheme for dynamic multidimensional datasets. The advantage of the scheme lies in the following two points. One is that the scheme provides the minimal encoding/decoding costs avoiding multiplications and divisions inherent in the existing schemes based on multidimensional arrays. The other is that the scheme provides an efficient method to handle a large-scale dataset by alleviating the problem of the address space limitation. These advantages have been confirmed by the experimental evaluations in construction time, storage size, and retrieval time.

References

[1] R. Zhang, P. Kalnis, B.C. Ooi, and K.-L. Tan, "Generalized multidimensional data mapping and query processing," ACM Transactions on Database Systems, vol.30, no.3, pp.661–697, 2005.
[2] R. Fenk, R. Markl, and R. Bayer, "Interval Processing with the UB-Tree," Proc. IDEAS, pp.12–22, 2002.
[3] J.A. Orenstein and T.H. Merrett, "A class of data structures for associative searching," Proc. PODS, pp.181–190, 1984.
[4] C. Faloutsos and S. Roseman, "Fractals for secondary key retrieval," Proc. PODS, pp.247–252, 1989.
[5] R. Bayer, "The universal B-tree for multidimensional indexing: General concepts," Proc. Worldwide Computing and Its Applications, pp.198–209, 1997.
[6] F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, and R. Bayer, "Integrating the UB-tree into a database system kernel," Proc. VLDB, pp.263–272, 2000.
[7] K.E. Seamons and M. Winslett, "Physical schemas for large multidimensional arrays in scientific computing applications," Proc. SSDBM, pp.218–227, 1994.
[8] S. Sarawagi and M. Stonebraker, "Efficient organization of large multidimensional arrays," Proc. ICDE, pp.328–336, 1994.
[9] Y. Zhao, P.M. Deshpande, and J.F. Naughton, "An array based algorithm for simultaneous multidimensional aggregates," Proc. SIGMOD, pp.159–170, 1997.

[10] E.J. Otoo and D. Rotem, "A Storage Scheme for Multi-dimensional Databases Using Extendible Array Files," Proc. STDBM, pp.67–76, 2006.
[11] E.J. Otoo and D. Rotem, "Efficient Storage Allocation of Large-Scale Extendible Multi-dimensional Scientific Datasets," Proc. SSDBM, pp.179–183, 2006.
[12] E.J. Otoo, D. Rotem, and S. Seshadri, "Optimal chunking of large multidimensional arrays for data warehousing," Proc. DOLAP, pp.25–32, 2007.
[13] K.M. Azharul Hasan, T. Tsuji, and K. Higuchi, "An Efficient Implementation for MOLAP Basic Data Structure and Its Evaluation," Proc. DASFAA, pp.288–299, 2007.
[14] T. Tsuji, H. Mizuno, M. Matsumoto, and K. Higuchi, "A Proposal of a Compact Realization Scheme for Dynamic Multidimensional Datasets," DBSJ Journal, vol.9, no.3, pp.1–6, 2009.
[15] T. Tsuchida, T. Tsuji, and K. Higuchi, "Implementing Vertical Splitting for Large Scale Multidimensional Datasets and Its Evaluations," Proc. DaWaK, pp.208–223, 2011.
[16] T. Tsuji, K. Amaki, H. Nishino, and K. Higuchi, "History-Offset Implementation Scheme of XML Documents and Its Evaluations," Proc. DASFAA, pp.315–330, 2013.
[17] M. Makino, T. Tsuji, and K. Higuchi, "History-pattern implementation for large-scale dynamic multidimensional datasets and its evaluations," Proc. DASFAA, pp.275–291, 2015.
[18] Y. Chiba, S. Kitajima, T. Tsuji, and K. Higuchi, "An Implementation Scheme for Tuple Datasets Incorporating Dynamic Addition of Attributes," Proc. 77th National Convention of IPSJ, 1N-06, pp.1-611–1-612, 2015.
[19] Free Software Foundation, GMP, "The GNU Multiple Precision Arithmetic Library," http://gmplib.org, 2013.
[20] Transaction Processing Performance Council: TPC-H, http://www.tpc.org/tpch, 2014.

Masafumi Makino received his M.E. degree in Information Science from University of Fukui in 2014. He is currently with NTT Neomaito Corporation.

Tatsuo Tsuji received his Ph. D. degree in Information and Computer Science from Osaka University in 1978. He has been a professor in the Graduate School of Engineering, University of Fukui until 2015, and is currently a senior fellow of the same university. His research interests include database management system. He is a member of the IEICE, the IPSJ and the DBSJ.

Ken Higuchi received his Ph D. degree in communication and system engineering from the University of Electro-Communications in 1997. He is currently an associate professor in the Graduate School of Engineering, University of Fukui. His research interests include parallel and distributed database systems. He is a member of the IEICE, the IPSJ and the DBSJ.