Sharing Flexibly Resizable Multidimensional Arrays

Bei LI † Tatsuo TSUJI † Ken HIGUCHI †

† Graduate School of Engineering, University of Fukui Bunkyo 3－9－1, Fukui-city, Fukui, 910-8507 Japan

E-mail: {libei,tsuji,higuchi}@pear.fuis.fukui-u.ac.jp

Abstract: Handling with large amount of data is common now. Employing extendible array to deal with increasing data is really effective. But in the conventional extendible arrays all of the operations such as extension and reduction can only occur to the surroundings of the array. This property limits the application of extendible arrays. So we proposed our flexibly resizable multidimensional arrays which can be inserted or deleted even in the midst of the array. In this paper we will propose a sharing scheme of resizable multidimensional arrays in a distributed environment which means the sharing of server side array by clients.

Keyword: Multidimensional arrays, Extendible arrays, Bitmap.

1. Introduction

Current distributed systems have to efficiently cope with many forms of dynamicity and be able to provide their service despite frequently structural changes in the data handled in the system. Things become much more complex when dynamicity becomes part of picture. *Extendible arrays* is very useful in handling this kind of dynamicity. They can extend or reduce without reorganizing existing array data.

But the current extendible arrays scheme can only extend or reduce at the surroundings of an array. We have proposed *flexibly resizable multidimensional array* scheme in which we can insert and delete subarrays even in the midst of the arrays. The scheme will greatly enlarge the application of the extendible arrays. In this research, with the objective of utilizing our flexibly resizable multidimensional arrays in a distributed environment, a sharing scheme with employing flexibly resizable multidimensional arrays between client and server is proposed and described.

The rest of the paper is organized as follows: Section 2 presents the background of our research, Section 3 describes flexibly resizable multidimensional arrays, Section 4 describes the simple sharing, Section 5 describes the proposed sharing scheme, Section 6 shows evaluation of the proposed sharing scheme on space and time costs, and Section 7 summarizes some conclusions.

2. Extendible Arrays

We employed multidimensional arrays originally for its fast accessing speed by fixing size in each dimension. But they cannot be dynamically extended or reduced unless we reorganize all the data which have been stored before. In order to solve these shortcomings we employed extendible arrays [4]-[6] which can be extended or reduced without any relocation of existing data. A n dimensional extendible array has a history counter h and three kinds of auxiliary tables for each dimension. See Fig.1. These tables are history table H_i, address table A_i and coefficient vector table C_i for each dimension $i(1 \leq i \leq n)$. H_i memorizes the extension history of subarrays, A_i memorizes the first address of subarrays, C_i memorizes the coefficients of the addressing function of each subarray. For an extendible arrays of size $[s_1, s_2, \ldots, s_n]$, when making an extension along dimension k, the contiguous storage of size $[s_1, s_2, \ldots, s_{k-1}, s_{k+1}, \ldots, s_n]$ will be dynamically allocated on the second storage. Then the history counter will be incremented by one and the value will be recorded in the corresponding H_i, the first address of this extended subarray is stored in the corresponding A_i. Note that an extended subarray is one to one corresponding with its history value. As is well known that the address of element $<i_1, i_2, \ldots, i_n>$ in the usual fixed size multidimensional array was computed using addressing function like:

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

Here $<s_2 s_3 \ldots s_n, s_3 s_4 \ldots s_n, \ldots, s_n>$ is called *a coefficient vector* of the subarray.

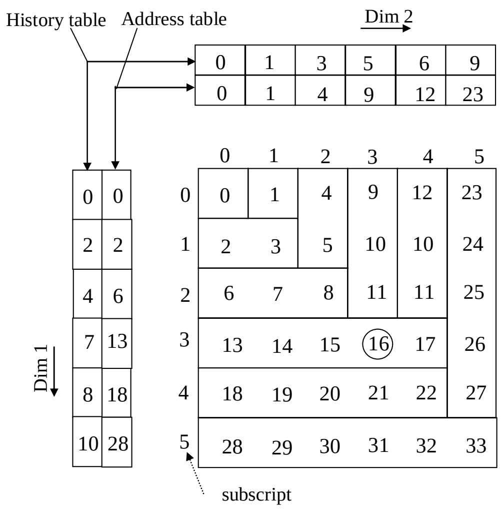


Fig.1 Extendible Multidimensional Arrays

Consider the element <3, 3> in Fig.1 as an example. The address computation procedures are the followings. Compare the history value, $H_1[3] = 7$ and $H_2[3] = 5$. Since $H_1[3] = 7 > H_2[3] = 5$, the element belongs to the subarray which occupies the address from 13 to 17 and the offset from the first address of the subarray is 3, so we can finally get the address of <3, 3> to be 16. Note that the element address can be computed out by employing some small auxiliary tables.

3. Resizable Multidimensional Arrays

We have proposed resizable multidimensional arrays [8], in which subarrays can be freely inserted or deleted even in the midst of the array. See Fig.2 and Fig.3. However, such insertion or deletion would influence the logical location of other elements such as element (A) in Fig.2 and Fig.3. But their physical locations remain in the same place. Therefore the offset computation by the addressing function which has been described in Section 2 cannot be applied here.

Let *e* be an element to be accessed and its coordinate be $<i_1, i_2, \ldots, i_n>$. The subarray including *e* is called as the *principal subarray* of *e*. Let *k* be the dimension to which this subarray belongs. Each subarray corresponding to the subscript i_p $(p \neq k)$ that belongs to dimension *p* is called as a *subordinate subarray* of *e*. It should be noted that the dimensions for which the compensation is necessary are those other than *k*.

In order to compensate, besides the three kinds of auxiliary tables mentioned in Section 2, we employed the insertion and deletion bitmap with the set of pairs <*history value*, *bit sequence*>, where each history value in this set is used for selecting the bit sequence to be used for calculating the extension or reduction compensation. An additional auxiliary table called *revised subscript table* is added to map the subscripts of the logical layout to the compensatory layout. See Fig.2 and Fig.3. The compensation value δ_k is:

$$\delta_k = (\textit{reduction compensation value}) - (\textit{extension compensation value})$$

And then make compensation to the coordinate $<i_1, i_2, \ldots, i_n>$ as $i'_k = i_k + \delta_k$ for each dimension $k (1 \leq k \leq n)$.

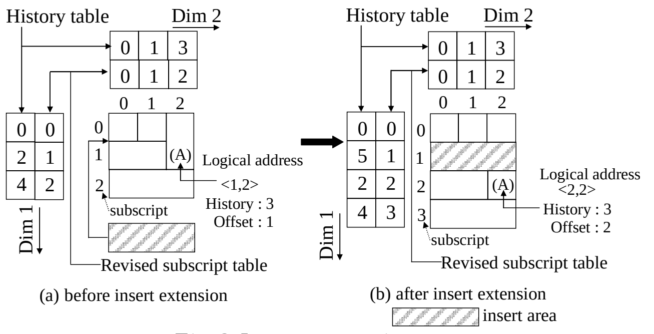


Fig.2 Insert extension

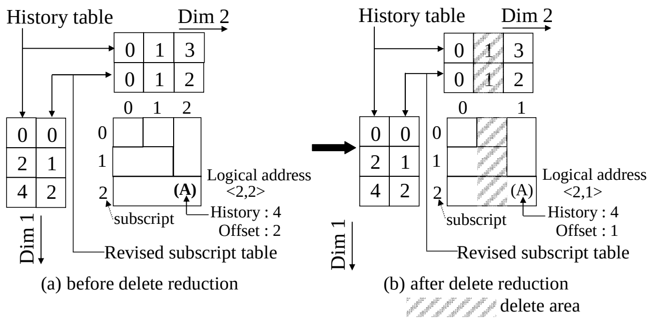


Fig.3 Delete reduction

Dimension 1

	History of insert extension
	Bit sequence
5	0100

Dimension 2

	History of delete extension
	Bit sequence
5	010

(a)Insertion bitmap of Fig.2 (b) Deletion bitmap of Fig.3

Fig.4 Insertion and deletion bitmap

The insertion bitmap of Fig.2 can be seen in (a) of Fig.4. The deletion bitmap of Fig.3 can be seen in (b) of Fig.4

Here for the sake of paper size, we ignore the detailed compensation procedures. Interested reader should consult [8] for more details.

4. Simple Sharing Scheme

In this section, we will give an outline of sharing in a distributed environment which means a part of or all of server can be shared by many clients [7]. In sharing there are two kinds of arrays:

(a) *host array*: the shared extendible array stored in server side.

(b) *client array*: the extendible array stored in client side that shares the host array.

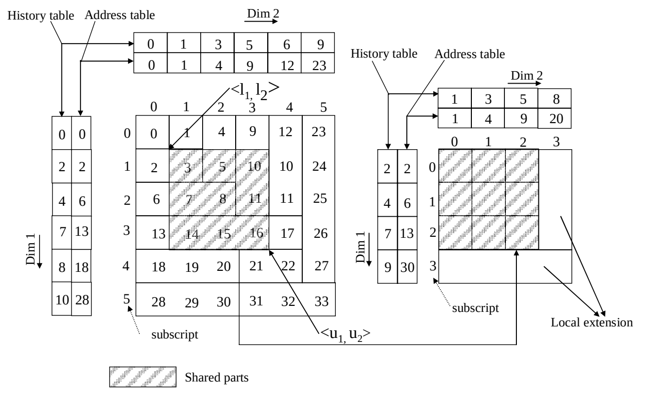


Fig.5 Simple sharing scheme

Let A be the host array stored at the server side. Assume that B is an client array at a client side sharing the portion of A from the subscript l_k to u_k of dimension $k(1 \leq k \leq n)$. See Fig.5. In order to make B work as an extendible array we need to prepare the history table and address table for each dimension k of B. Let H_k^A and L_k^A denote the history table and address table of A in host array. H_k^B and L_k^B denote those tables of B in the client array. H_k^B and L_k^B can be obtained by copying the corresponding portion of the host array. After initial sharing of the host array, local extensions at the surroundings can be performed in client array. But this simple kind of sharing suffers from the following two problems:

(1) Only the contiguous host array area on the server can be shared by client.

(2) Once the initial sharing parts on the host array have been settled down, the local extension or newly dynamically shared subarray of the host array can only be handled at the surroundings of client array.

5. Flexible Sharing Scheme

In order to solve the two problems described in the previous section, the concept of flexibly resizable multidimensional arrays [8] will be employed for sharing in a distributed environment. There are two kinds of sharing.

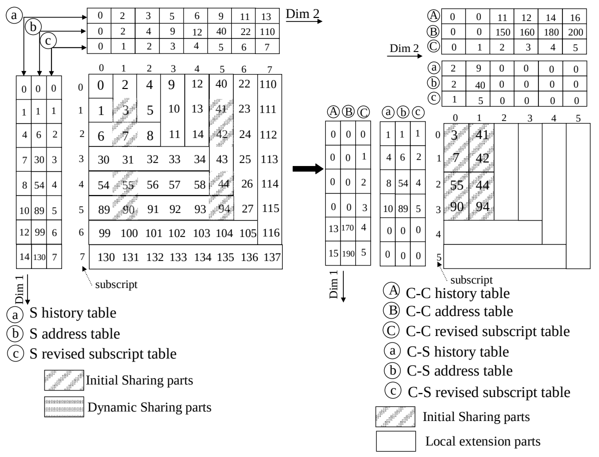
(a) *Initial sharing.*

Random noncontiguous subarrays of host array can be shared by client. This solves problem (1) in Section 4. See Fig.6.

(b) *Dynamic sharing.*

After initial sharing, other unshared parts of host array can be loaded into sharing and logically be inserted into client array at any position, invoking to the capability of flexibly resizable multidimensional arrays described in Section 3. This will solve problem (2) in Section 4. See Fig.10.

Although the flexible sharing scheme solves the two problems mentioned before, the sharing algorithm will cause the similar anomalies which have been mentioned in Section 3 to elements of the client array.



(a) Host array (b) Client array

Fig.6 Flexible sharing scheme

To exemplify, in Fig.6 element <1, 5> in host array exists in the principal subarray S of history value 9(dimension 2, subscript 5). The element exists at the offset 1 of S. So, the element address is computed as 40+1=41. After the flexible sharing, the logical location of the elements becomes <0, 1>in Fig.6 (b). The element exists at the offset 0 of S. So, the address is computed as 40+0 = 40. Such kind of anomaly should be reconciled to get the correct physical location. That is an efficient mapping mechanism to convert a user specified coordinate of an array element (i.e. tuple of subscripts) into its correct physical location.

5.1 Layout of the client array

There are three kinds of subarrays existed in client side.

(a) *shared subarray*, which is caused by sharing of the host array and logically loaded into the client array.

(b) *local subarray*, which is caused by local extension and stored in the client side.

(c) *complementary subarray,* which is caused by sharing and stored in client side. Shared subarray will be handled as a local insertion and continues storage of size $[s_1, s_2, \ldots, s_{k-1}, s_{k+1}, \ldots, s_n]$ will be sequentially allocated in client side. But the size of shared subarray is smaller than the size of newly allocated subarray. In order to make it logically work as a flexibly resizable multidimensional arrays. The complementary subarray will be created. Its size equals to total size of newly allocated subarray subtracts the size of shared subarray. Like the subarray of history 22 (dimension 1, subscript 5) which not including the subarray with horizontal lines. (See C-C auxiliary tables ⒶⒷⒸ of Fig.10). Note that only contiguous storage size of complementary subarray will be allocated in client storage.

So there are three kinds of compensation algorithms for calculating the address of an element depending on the kind of subarray it belongs to. Note that all the address compensation is done at the client side.

There are two sets of auxiliary tables in a client array.

(a) *C − S auxiliary tables*: The tables are for the shared subarrays. The portion of client side subarrays are all set as 0. See the tables of ⓐⓑⓒ in Fig.6 (b) and Fig.10.

(b) *C − S auxiliary tables*: The tables are for the client subarrays and complementary subarrays. Except the revised subscript table, the portion of shared subarrays are all set as 0. See the tables ⒶⒷⒸ in Fig.6 (b) and Fig.10.

5.2 Compensatory data structures

Two extra kinds of bitmap are introduced for offset compensation. One is *S-bitmap* which records the situation of interleaves in the host array. That is, the corresponding bits of shared subarray in S-bitmap will be set as 1 while bits of unshared host subarray will be set as 0. See Fig.7. Another is the C-bitmap which is used for reflecting the interleaved situation of shared subarrays and client side locally extended subarrays. Namely, the corresponding bits of shared subarray in C-bitmap will be set as 1 while client side locally extended subarrays will be set as 0. See Fig.8.

Dimension	Bit sequence
dimension 1	01101100
dimension 2	01000100

Fig.7 S-bitmap of Fig.5

Dimension	Bit sequence
dimension 1	111100
dimension 2	110000

Fig.8 C-bitmap of Fig.5

5.3 Address computation procedures

(1) Check the position of the element.

The C-bitmap will be inspected. By searching the bits corresponding to coordinate of element in each dimension, the position of element can be determined. If all of the corresponding bits are 1, the element belongs to a shared subarray. If all of the bits are 0, the element belongs to a local subarray. If all of bits are composed by 0 and 1, it belongs to a complementary subarray.

(2) Determine the principal subarray of the element.

(a)The element belongs to a shared subarray.

The C-S auxiliary tables will be selected. See the element <5, 1> and ⓐⓑⓒ in Fig.10.

(b)The element belongs to a local subarray.

The C-C auxiliary tables will be selected. See the element <6, 5> and ⒶⒷⒸ in Fig.10.

(c)The element belongs to a complementary subarray.

The C-C auxiliary tables will be selected. See the element <5, 5> and ⒶⒷⒸ in Fig.10.

By comparing corresponding history table value of each dimension, the principal subarray of dimension k will be determined. The compensation is needed for the dimensions other than *k*.

(3) Do compensation and compute out the address of elements. There are 3 cases:

(a) The element belongs to a shared subarray.

The compensation value λ_k is calculated as:

$$\lambda_k = (\text{sharing compensation value}) - (\text{inverse compensation value}) - (\text{interleave compensation value})$$

Here *sharing compensation value* is computed by inspecting S-bitmap. The value is the total number of unset bits. The number is counted up to corresponding bit of the element coordinate in dimension *k*. *Inverse compensation value* is computed by inspecting C-S revised subscript table. (See table ⓒ in Fig.10) The value is the number of shared subarray which its C-S revised subscript value is bigger than the C-S revised subscript value of element in dimension *k*. *Interleave* *compensation value* is computed by inspecting C-bitmap, the value is the total number of unset bits. The number is counted up to the corresponding bit of element coordinate in dimension *k*. The compensated coordinate $< i'_1, i'_2, \ldots, i'_n >$ of *e* will be computed as $i'_k = i_k + \lambda_k$ for each dimension $k(1 \le k \le n)$. See element <5, 1> in Fig.10.

(b) The element belongs to a local subarray.

The compensation value δ_k is computed as:

$$\delta_k = (\textit{reduction compensation value}) - (\textit{insertion compensation value})$$

The compensated coordinate $< i'_1, i'_2, \ldots, i'_n >$ of *e* will be computed as $i'_k = i_k + \delta_k$ for each dimension $k(1 \le k \le n)$. See element <6, 5> in Fig.10.

(c) The element belongs to a complementary subarray.

The compensated offset θ is computed as:

$$\theta = s_{i_1} s_{i_2} \ldots s_{i_{n-1}} e'_n + s_{i_1} s_{i_2} \ldots s_{i_{n-2}} e'_{n-1} + \ldots + s_{i_1} e'_2 + e'_1 - (j_{i_1} j_{i_2} \ldots j_{i_{n-1}} a_n + j_{i_1} j_{i_2} \ldots j_{i_{n-2}} a_{n-1} + \ldots + j_{i_1} a_2 + j_{i_1})$$

Here if $e_m > j_m$, $a_m = j_m - 1$ else $a_m = e_m$. j_m is the size of shared subarray in dimension m, e'_m is the compensated coordinate of element which has been subtracted interleaved compensation value from original coordinate of dimension *m*. s_m is the size of the client array in dimension *m*. See element <5, 5> in Fig.10.

5.4 An example of sharing compensation

Fig.9 and Fig.10 show an example of dynamic sharing. Fig.9 is the layout of the host array. Fig.10 is the layout of a client array. After the initial sharing, insertion in a local client array at subscript 3 of dimension 2 will be performed first. Then a dynamic sharing of a host subarray will be performed. The dynamically shared subarray will be logically inserted into subscript 5 of dimension 1 in the client array. See Fig.10

Fig.11 and Fig.12 are the S-bitmap and C-bitmap of Fig.9 and Fig.10 respectively. For elements <5, 1>, <6, 5>, <5, 5> in Fig.10, the procedures of dynamic sharing compensation will be described.

(1) Element <5, 1> in a shared subarray.

(i) The element belongs to the principal subarray of C-S history value 9 (dimension 2, subscript 1). See the C-S auxiliary tables ⓐ ⓑ ⓒ in Fig.10. The compensation is needed for dimension 1.

(ii) The S-bitmap in Fig.11 is inspected. The bit sequence 01111100 of dimension 1 will be searched. The C-S revised subscript value of the element in the C-S revised subscript table of dimension 1 is 3 (See C-S revised subscript table Ⓒ in Fig.10). The total number of unset bits is counted up to bit 3 of the bit sequence. So the *sharing compensation value* is concluded to be 1.

(iii) The C-S revised subscript value of <5, 1> is 3 in dimension 1. (See C-S revised subscript table Ⓒ of Fig.10). Its C-C revised subscript value is 5 in dimension 1. (See C-C revised subscript table Ⓒ of Fig.10). With the C-S revised subscript value 3, the C-S revised subscript table Ⓒ of dimension 1 will be searched from 0 to 5. The server side revised subscript value 5 and 4 is bigger than 3, so the *inverse compensation value* is 2.

(iv) The C-bitmap in Fig.12 will be inspected. The C-C revised subscript value of the element is 5 in dimension 1. (See C-C revised subscript table Ⓒ of dimension 1 in Fig.10). The bit sequence 1111010 of dimension 1 will be searched. The number of unset bits up to 5 is 1, so the *interleave compensation value* is concluded to be 1.

Now the total compensation value of dimension 1 is $\lambda_1 = 1 - 2 - 1 = -2$, so the compensated offset is $5 - 2 = 3$, hence the address of the element is $40 + 3 = 43$.

(2) Element <6, 5> in a local subarray.

(i) The element proves to belong to the principal subarray of C-C history value 19 (dimension 1, subscript 6). See the C-C auxiliary tables ⒶⒷⒸ in Fig.10.

(ii)The client side insertion bitmap and client side deletion bitmap will be inspected. See Fig.13. The insertion bit sequence 0001000 and deletion bit sequence 0000000 will be searched. The C-C revised subscript value of <6, 5> is 5 in dimension 2. See C-C revised subscript table Ⓒ of dimension 2 in Fig.10. The total number of set bit is counter up to 5. For client side insertion bitmap, the number is 1, for the client side deletion bitmap is 0. So the compensation value is concluded to be $\delta_2 = 1$.

Hence the compensated offset value of <6, 5> is concluded to be $5 - 1 = 4$ and the address is $190 + 4 = 194$.

(3) Element <5, 5> in a complementary subarray

(i) The element belongs to the principal subarray of C-C history value 22 (dimension 1, subscript 5). See the C-C auxiliary tables ⒶⒷⒸ in Fig.10. The size of the shared array in dimension 2 is 2. So the compensation value is concluded to be -2.

(ii)The C-C revised subscript value of the element in dimension 2 is 5. See the C-C revised subscript table Ⓒ in Fig.10.

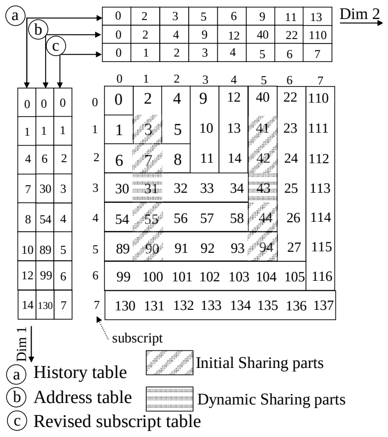


Fig.9 Dynamic sharing (server side)

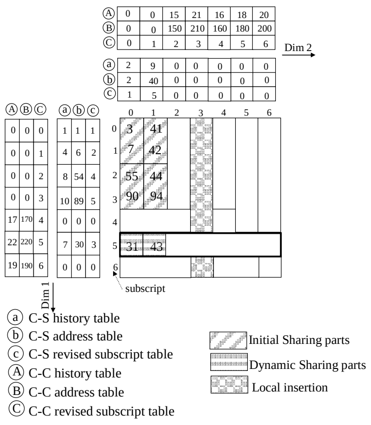


Fig.10 Dynamic sharing (client side)

Dimension	Bit sequence
dimension 1	01111100
dimension 2	01000100

Fig.11 S-bitmap

Dimension	Bit sequence
dimension 1	1111010
dimension 2	1100000

Fig.12 C-bitmap

History counter		Bit sequence
dim1	21	0000010
dim2	21	0001000

(a) Client side insertion bitmap

History counter		Bit sequence
dim1	21	0000000
dim2	21	0000000

(b) Client side deletion bitmap

Fig.13 Client side insertion and deletion bitmap

So the compensated offset of <5, 5> is concluded to be $5-2=3$ and the address is $220+3=223$.

5.5 Sharing procedures

In this section, the procedures and the effects of flexible sharing will be explained.

5.5.1 Operation on bitmaps

(1)Initial sharing:

If the initial sharing is specified, an S-bitmap and C-bitmap will be created. For dimension $k(1 \leq k \leq n)$, the length of bit sequence in the S-bitmap equals to the current size of the host array in dimension *k*. The length of bit sequence in the C-bitmap equals to the current size of the client array in dimension *k*. All the bits of the bit sequence in dimension *k* of S-bitmap and C-bitmap are initially all 0.

If subscript *i* in dimension *k* of the host array is shared, and logically loaded at subscript $i^{'}$ of client side array. For the S-bitmap, bit *i* of the bit sequence in dimension *k* is set as 1. See Fig.7. For the C-bitmap, bit $i^{'}$ of dimension *k* is set as 1. See Fig.8.

(2)Dynamic sharing

After the initial sharing, assume that subscript *i* of dimension *k* of a host array is specified to be shared and will logically inserted in subscript *i*' of dimension *k* of the client array. Note that a new pair of bit sequence with current history counter value of the client array is added to client side insertion bitmap while it is initially all 0 and all kinds of bitmap will be updated. Namely, except S-bitmap, bits after bit *i'* of each corresponding bit sequence of client side insertion bitmap, client side deletion bitmap, C-bitmaps will be shift to right by 1.

For the bit sequence of the S-bitmap, bit *i* of dimension *k* will be set as 1, for C-bitmap bit *i'* will be set as 1, for client side insertion bitmap will be set as 1 and for client side deletion bitmap will be set as 0. See Fig.11 and Fig.12.

(3) Host array insert extension.

If a host array insert extension at subscript *i* of dimension *k* is performed, a new pair of bit sequence with the current history counter value of server side will be added to server side insertion bitmap with initially all 0, three kinds of bitmaps will be updated. Namely the bits of dimension *k* after bit *i* will be shift to right by 1. For the S-bitmap, bit *i* of dimension *k* is set as 0, for server side insertion bitmap, bit *i* of each bit sequence is set as 1, for the server side deletion bitmap is set as 0.

(4) Host array delete reduction.

If a host array delete reduction at subscript *i* of dimension *k* is performed, two kinds of bitmap will be updated. That is, for the S-bitmap, bit *i* of dimension *k* is set as 0, for the server side deletion bitmap is set as 1. If it is still on sharing, the deletion will be delayed until this part will not be shared anymore.

(5) Local array insertion

If an insert extension of local client side on subscript *i* of dimension *k* is performed, a new pair of bit sequence with current history counter value of client side will be added to client side insertion bitmap with initially all 0. C-bitmap, client side insertion bitmap and client side deletion bitmap in dimension *k* will be updated. Namely, the bits of each bit sequence after bit *i* will be shifted to right by 1. For the C-bitmap, bit *i* of dimension *k* is set as 0, for client side insertion bitmap, bit *i* of each bit sequence is set as 1, and for client side deletion bitmap, the bit is set as 0.

(6) Local array deletion

If a delete reduction at subscript *i* of dimension *k* is performed, for client side deletion bitmap of dimension *k*, bit *i* of each bit sequence is set as 1.

6. Sharing Example

We give a sharing example of a graph database represented by two adjacent matrices G and G1, which are a host and a client array respectively. Suppose each of G and G1 represents a map of cities in a state. A node in each graph represents a city and each number on a bidirectional arrow represents the distance between two cities.

Initially cities ⓐⓒⓔ in G are shared by G1 for visiting from the cities ⓕⓖⓗ in G1 (Fig.14). Note that the route between ⓔ of G and ⓖ of G1 is also established in G1.

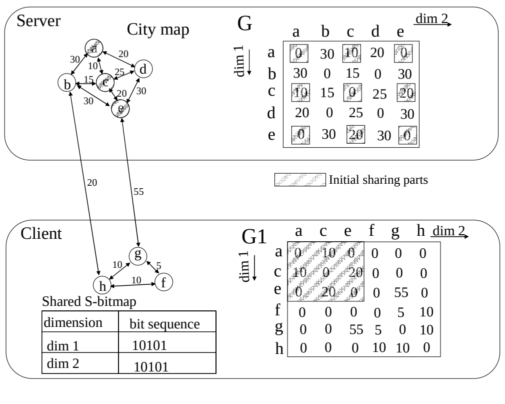


Fig.14 Initial sharing

After the initial visiting plan, suppose that ⓑ in G is added as a visiting city. In this situation node ⓑ would be dynamically shared from G. See Fig.15. Note that the route between ⓑ in G and ⓗ in G1 is also established in G1.

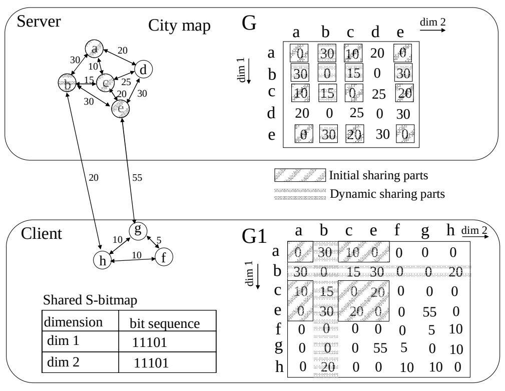


Fig.15 Dynamic sharing

7. Evaluations

In this section the implementation models of simple sharing and flexible sharing will be compared based on a constructed prototype system. The server side CPU is UltraSPARC , 200MHz and the client side CPU is UltraSPARC IV, 1050MHz.

We have the following three kinds of client array.

(a) SS: a client array of simple sharing.
(b) FS1: a client array that has undergone only initial sharing
(c) FS2: a client array that has undergone dynamic sharing after the same initial sharing as FS1.

Let the size of each dimension be *s*. In FS2, the dynamic sharing has undergone *s*/10 times in each dimension.

7.1 Storage cost

In FS1 and FS2, in addition to the storage required in SS, the S-bitmap, C-bitmap and an extra set of auxiliary tables will be stored beside the array body. Each array element occupies 4 bytes. As shown in Table 1, compared with array body, the auxiliary table size of SS, FS1 and FS2 is very small. The size of bitmap is depending on times of the insertion, deletion and dynamic sharing, but it is negligibly small.

Table 1 Storage cost

Dimensions (dim. size)	3(400)	4(90)	5(36)
Number of element	$6.4x10^7$	$6.27x10^7$	$6.05x10^7$
Array body size	244MB	239MB	231MB
Auxiliary table size of SS	144KB	54.5KB	85KB
Auxiliary table size of FS 1	379.5KB	135.6KB	204.3KB
Auxiliary table size of FS 2	410KB	140.6KB	206.5KB

7.2 Access time cost

7.2.1 Random accessing cost

For a client array whose total number of elements is *m*, the random accessing is performed *m*/100 times. From Fig.16 we can find that the FS1 and FS2 is little slower than the SS. This is due to the complex compensation procedures for the accessing element.

.

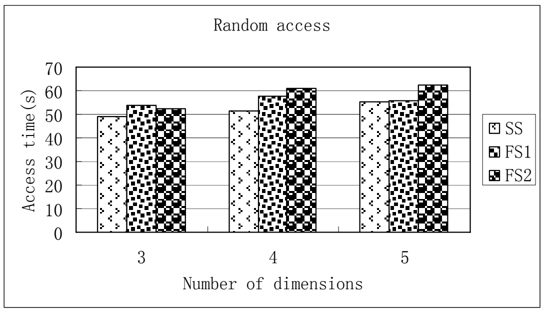


Fig.16 Random access

7.2.2 Range accessing cost

For a *n* dimensional array, all the elements in the set of *n*-1 dimensional slices of the corresponding dimension are retrieved. Fig.17 shows the retrieval time. The range access cost is much lower than the random access cost. There are two possible reasons. One is due to that the compensation procedure in random access is invoked for every element. While in the range access, the compensation value can be used for the same subscript of the same dimension without recomputing. The other is that in the range accessing, instead of sending the first address and compensated offset to the server, the searching range is sent to server, so communication cost can be saved.

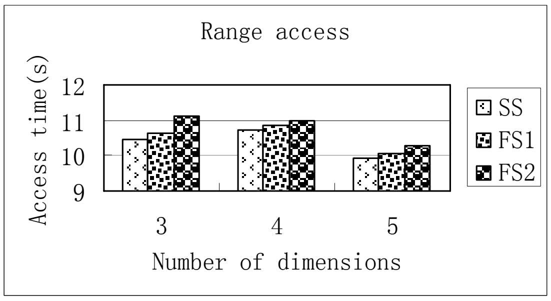


Fig.17 Range access

8. Conclusions

In this paper, we introduced a new sharing scheme for distributed system illustrating flexible sharing. From the cost model, we computed and compared with simple sharing scheme. In the most cases we found that the flexibility of the new scheme can be implemented efficiently only at the cost of small additive increase in both storage and retrieval time compared with simple sharing.

References:

[1] S.Sarawagi and M.Stonebraker, ‘Efficient Organization of Large Multidimensional Arrays’, Proc. of International Conference on Data Engineering, 328-336, 1994.

[2] Y.Zhao,K.Ramasamy, K. Tufte and J.L.Zhao, ‘Array-Based Evaluation of Multi-Dimensional Queries in Object Relational Database Systems’, Proc. of 14-th International Conference on Data Engineering, 241-249,1998.

[3] T.Tsuji, G.Mizuno, T.Hochin, K.Higuchi, ’A Deferred Allocation. Scheme of Extendible Arrays’, Transaction of IEICE, Vol.J86-D-I, No.5, 351-356, 2003.

[4] N.Widmann, P.Baumann, ‘Efficient Execution of Operations in a DBMS for Multidimensional Arrays’, Proc. of 10-th International Working Conference on Scientific and Statistical Database Management, 155-165, 1998.

[5] A.L.Rosenberg, ‘Allocating Storage for Extendible Arrays’, JACM, Vol.21, 652-670, 1974.

[6] A.L.Rosenberg and L.J.Stockmeyer, ‘Hashing Schemes for Extendible Arrays’, JACM, Vol.24, 199-221,1977.

[7] T.Tsuji, H.Kawahara, T.Hochin, and K.Higuchi, ‘Sharing Extendible Arrays in a Distributed Environment’ Lecture Notes in Computer Science 2060, 41-53, 2001.

[8] M.Kumakiri, B.Li, T.Tsuji, K.Higuchi, ’Multidimensional Arrays Enabling Insert Extensions and Delete Reductions’, DBSJ Letters, Vol.5, 61-64, 2006.