DEWS2006 3C-o1

挿入拡張，中抜き縮小可能な多次元配列

熊切　正和† Li　Bei† 都司　達夫† 樋口　健†

† 福井大学大学院工学研究科，福井市
{m_kuma, libei, tsuji, higuchi}@pear.fuis.fukui-u.ac.jp

あらまし 最近，MOLAP 等，多次元データの格納データ構造として多次元配列の有用性が再認識されている．通常，データベースで用いられる多次元配列は，要素に高速にランダムアクセスできるために，全ての次元のサイズが固定である．そのため，配列データの再配置無しに拡張や縮小など配列のサイズ変更を行えない．拡張可能配列では，データの再配置を必要とせずに，どの次元方向に対しても拡張が行える．しかし，拡張可能配列は，配列の外縁に対してのみしか拡張ができないという制限がある．本稿では，中抜き縮小や挿入拡張を導入することで，柔軟にサイズの動的変更が可能な拡張可能多次元配列について述べる．

キーワード 拡張可能配列，多次元配列, MOLAP

Flexibly Resizable Multidimensional Arrays

Masakazu KUMAKIRI†, Bei LI†, Tatsuo TSUJI†, and Ken HIGUCHI†

† Graduate School of Engineering, University of Fukui, Fukui-shi, 910-8507, Japan
{m_kuma, libei, tsuji, higuchi}@pear.fuis.fukui-u.ac.jp

Abstract Recently, multidimensional arrays are becoming important data structures for storing large scale multidimensional data; e.g., in scientific databases or MOLAP databases. Size of multidimensional arrays used in such database is fixed in every dimension in order to be benefited by the fast random accessing capability to array elements. While such a fixed size array cannot extend or shrink without relocating all of the elements, an extendible array can extend its size along any directions without any relocation. However that the existing extendible arrays can always extend only at the surrounding is a strict restriction. In this paper, we propose a new flexible extendible array organization, in which a subarray can be inserted or removed even in the midst of the array.

Key words Extensible Array, Multidimensional Arrays, MOLAP

1. まえがき

科学技術アプリケーションではしばしば科学事象のモデリングや分析において多次元配列が用いられる．大規模な科学技術データを効率よく取り扱いという強い影響により，主メモリや二次記憶上での多次元配列の構造化技法やそれらの実装技術の研究が行われてきた [1] [2] [4] [5]．また近年，多次元配列を用いた MOLAP（Multidimensional On-Line Analytical Processing）の研究 [3] [6] が盛んに行われるようになり，会社や組織が保有する大量データの統計的な分析を基に経営や販売戦略の意思決定に使われ始めている．

たとえば，MOLAP システムでは，多次元配列に対する問い合わせを迅速に処理するために，配列の高速参照が要求される．要素参照の速度に加えて，多次元配列に対する効率よい演算操作機能は科学技術アプリケーションや MOLAP アプリケーションにとって重要な要求であり，参照速度を劣化させないために，対象とする配列を固定サイズで確保し，要素の格納アドレスを求めるための高速なアドレス関数を使用する．しかし，新たなカラム値を持つレコードが挿入されたときには，それを配列に格納するためにはその次元方向に 1 大きい新たな固定サイズ配列を確保して，新たなアドレス関数を使って，元の配列要素すべてを再配置する必要があり，大規模データを扱うようなアプリケーションでは，著しくリアルタイム性が阻害される．

拡張可能配列は実行時に動的に任意の次元方向にそのサイズが拡張できる配列である [8] [9] [10]．動的配列（dynamic array）[7] が実行時に各次元のサイズが確定し，配列実体が新たに動的に割り付けられるのに対して，拡張可能配列の場合は拡張部分のみが動的に割り付けられ，拡張前の配列要素のデータは再配置することなくそのまま利用される．上記アプリケーションをはじめ，配列サイズがあらかじめ予測不可能な場合や，配列サイズが実行環境の変化に応じて動的に変化しうるような様々なアプリケーション分野において拡張可能配列は有利とな

る．拡張可能配列の実装上のポイントは，配列要素のアクセス性能や記憶域の利用率を犠牲にせずに，拡張可能性を実現することであると言える．

[8] [9] ではハッシュ関数を使用した拡張可能配列の実現技法が提案されており, [10] [11] では補助テーブルを使用した実現技法が論じられ, [10] ではハッシュ関数による方法に対する優位性が述べられている．また, [10] の方法について, [12] では，補助テーブルの構造化について述べられている．また, [13] ではクライアントサーバ環境下での拡張可能配列の共有方式を提案されており, [14] では拡張可能配列の遅延割付方式が提案されている．

これらの拡張可能配列の実現モデルでは, [13] [14] を除いて，すべて次元方向の外縁部からの拡張のみ可能である. [13] [14] では不要になった領域の縮小を次元方向に可能としているが，配列の外縁部からの縮小のみであり，応用上，問題となっている．本論文ではこれらの問題点に対して，解決を試みる．すなわち，拡張可能配列の内部に対してサブ配列を挿入して配列を拡張する（挿入拡張）こと，および，内部のサブ配列を除去して配列を縮小する（中抜き縮小）ことを可能にするための方式を提案し，実装評価する．

2. 拡張可能配列の実装方式

n 次元拡張可能配列 A は 1 つの経歴値カウンタと次元毎に 3 種類の補助テーブルを有している． これらの補助テーブルは経歴値テーブル，アドレステーブルおよび係数テーブルと呼ばれる．経歴値テーブルは 1 次元配列であり，配列拡張が行われるたびに現在の経歴値カウンタが 1 インクリメントされ，その値がテーブルに順次記録される．ある次元方向の配列拡張はその次元を除く $n-1$ 次元の配列断面に相当するサイズの連続する記憶領域をシステムのメモリ動的割り付け機能（例えば UNIX における動的メモリ割付機能である malloc ()）を使って確保し，A に追加することによって行われる（図 1）．この連続記憶領域は $n-1$ 次元の通常の固定サイズ配列であり，以後，A のサブ配列と呼ぶ．拡張可能配列およびサブ配列の各次元の添字はいずれも 0 から始まり，次元は 1 から数えるものとし，配列の 1 要素のサイズは 1 とする．

例えば，各次元のサイズが $[s_1, s_2, s_3, s_4]$ の通常の固定サイズの 4 次元配列要素をメモリ上に次元 4 ～ 1 の順に線形に割り付ける場合，要素 $\langle i_1, i_2, i_3, i_4 \rangle$ のアドレスはよく知られているように，

$$s_2 s_3 s_4 i_1 + s_3 s_4 i_2 + s_4 i_3 + i_4 \quad (1)$$

なる $i_1 \sim i_4$ に関する 1 次関数を計算して得られる．これに対して，例えば現在のサイズが $[s_1, s_2, s_3, s_4]$ の 4 次元拡張可能配列の場合には，次元 2 の方向に 1 つ拡張する時，サイズ $[s_1, s_3, s_4]$ の 3 次元サブ配列 S が動的に確保される．アドレステーブルは各サブ配列の先頭アドレスを保持する 1 次元配列である．

A が 3 次元以上の拡張可能配列の場合には，サブ配列内の要素のアドレスを計算する 1 次関数の $n-2$ 個の係数からなる係数ベクトルをサブ配列毎に記録する係数テーブルを各次元について必要とする．例えば，上記サブ配列 S の要素 $\langle i_1, i_2, i_3 \rangle$ のアドレスは（1）式と同様，1 次関数

$$s_3 s_4 i_1 + s_4 i_2 + i_3 \quad (2)$$

となる．以後，この $(s_3 s_4, s_4)$ を S の係数ベクトル，サブ配列内の要素のアドレスをサブ配列内オフセットと呼ぶ．配列が使用できるプログラミング言語の場合，通常この係数ベクトルの値は配列サイズよりコンパイラが計算し，この 1 次関数を計算するコードを生成する．拡張可能配列の場合には，係数ベクトルの値は拡張時の A の各次元のサイズに依存しているので，拡張時に係数ベクトルを計算し，それを拡張する次元の係数テーブルの当該スロットに書き込む．なお，A が 2 次元の拡張可能配列の場合には，サブ配列は 1 次元配列となるので，このような係数テーブルは必要としない（図 1）．

図 1 において，次元 1 方向および次元 2 方向の経歴値テーブルをそれぞれ H_1，H_2 とし，またアドレステーブルをそれぞれ A_1，A_2 とする．例えば配列要素 $\langle 3,4 \rangle$ のアドレス計算は次のように行われる．$H_1[3] < H_2[4]$ であるから，要素 $\langle 3,4 \rangle$ を含むサブ配列 S は経歴値 $H_2[4] = 7$ の時に割付けられ，その先頭アドレスは $A_2[4] = 60$ である．また，要素 $[3,4]$ の S におけるオフセットは 3 なので，求めるアドレスは 63 となる．

このように，n 次元拡張可能配列の場合には，最大経歴値を調べるための表引きコストと経歴値比較コストが必要となる．しかし，そのサブ配列は $n-1$ 次元であるために，n 次元固定配列のアドレス計算より，乗算回数，加算回数がそれぞれ 1 回ずつ減少する（(1), (2) 式）．

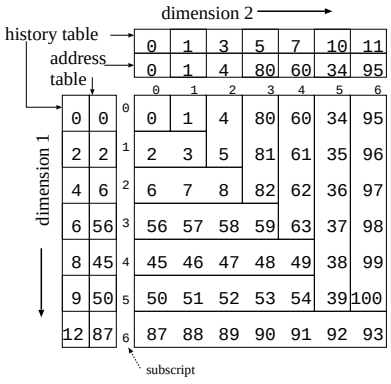


図 1　主記憶上の拡張可能配列の実現モデル

経歴値テーブルやアドレステーブルの記憶コストは各次元毎に 1 次元配列を確保するのみなので，$O(n)$ である．一方，係数テーブルはその次元のサイズに相当する個数の係数ベクトルを収容する．この係数ベクトルは $n-2$ 個の定数からなり，係数テーブルのスロットにサブ配列毎に記録される．次元毎に係数テーブルが存在し，係数テーブルはその次元のサイズに相当する個数の係数ベクトルを収容する．従って，係数テーブル全体の記憶コストは $n(n-2)$ に比例するので $O(n^2)$ となる．特に MOLAP や大規模な数値行列演算などへの応用においては次元数 n およびサイズともに大きくなるので，補助テーブルの合計サイズは相当大きくなる．したがって，補助テーブル（索引）を主記憶上に配置するためには，そのサイズ抑制が重要な

課題になる．

また，我々の実現モデルでは，配列の外側に値を追加する通常の拡張のみではなく，任意の位置へサブ配列を挿入できる挿入拡張，同時に縮小を可能としている．挿入拡張や縮小を許すことで，論理的配置と物理的配置間の補正が必要となるが，この問題は次節に記す．なお，従来の実現モデルでは挿入拡張や縮小可能性は扱われていない．

3. 挿入拡張，中抜き縮小を許す構造

3.1 挿入拡張，中抜き縮小により生じる問題点

従来の拡張可能配列における拡張と異なり，挿入拡張や中抜き縮小は既に配列に格納されている要素の論理的な位置，即ち添字の組へ影響を与える．図 2 の拡張例では，次元 1 へ新たな値を挿入拡張することで，要素 A $\langle 1,1\rangle$ の論理的な位置は $\langle 2,1\rangle$ となる．また，縮小の際には補助テーブルの縮小箇所を詰めるため，図 3 の縮小例において要素 B $\langle 2,2\rangle$ は $\langle 1,2\rangle$ に変わる．実際に要素が含まれているサブ配列やアドレスに変化は無いが，上記のように拡張可能配列中の論理的位置がずれることで，サブ配列中の正しいオフセットの計算ができず，結果として異なったアドレスを導いてしまう．

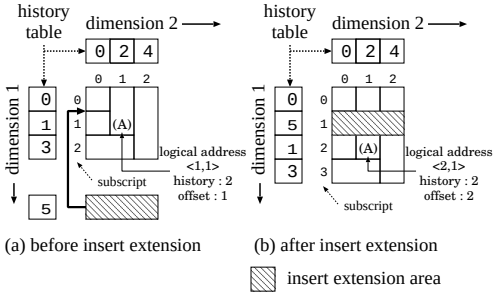


図 2　拡張によって生じるオフセットのずれ

例えば，図 2 の配列において，要素 $A\langle 1,1\rangle$ は次元 2 の添字 1 のサブ配列中に存在し，次元 1 の添字が 1 であるためオフセットは 1 が導かれる．要素 A は次元 1 の 1 列目へ挿入拡張を行った後でも，拡張前と変わらずに次元 2，添字 1 のサブ配列に含まれ，要素 A を含むサブ配列は挿入拡張が行われる前と同様のものが求まる．しかし，要素 A の挿入拡張が行われた次元である次元 1 における添字は挿入拡張前の値と異なりオフセットは 1 ではなく 2 が導かれてしまう．これにより挿入拡張前と後にて同じアドレスが導けず，正しい結果は得られない．縮小においても同様の問題が生じる．なお，以後本稿では，縮小によって開放された領域を斜線で図示したレイアウトを補正レイアウト，縮小した領域を詰めたユーザの視点からのレイアウトを論理レイアウトと呼ぶ．

3.2 補正のための構造

3.1 節で記した問題を解決するために，挿入拡張・縮小で生じたオフセットのずれを補正し，正しい配列要素の位置を導くことのできる構造を考える．補正が必要な次元は要素を含むサブ配列の所属次元 i 以外の次元である．ここで次元 i 以外の添字に対応するサブ配列を補参照サブ配列と呼ぶ．また，補正値

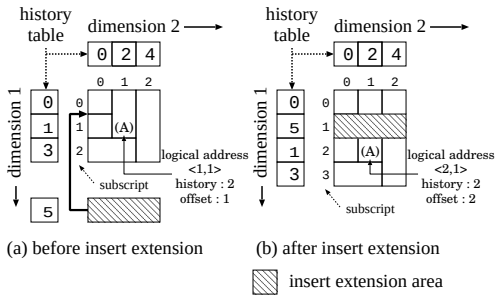


図 3　縮小によって生じるオフセットのずれ

の管理は，要素の補正値を，アクセスする要素の位置や拡張・縮小の経緯に影響されること無く，ランダムに求められるよう，bitmap を用いて行う．bitmap は次元毎に管理し，挿入拡張によるオフセットのずれを補正する拡張補正値と縮小によるオフセットのずれを補正する縮小補正値をそれぞれ個別に求める．したがって，それぞれの次元毎に拡張補正 bitmap，縮小補正 bitmap を保持し，各次元毎に

要素の論理添字 − 拡張補正値 + 縮小補正値

を計算し，サブ配列内のオフセット演算に用いる．それぞれの bitmap は複数の bit 列と補正値演算に用いる bit 列選択のための経歴値の組で構成される．また，本方式では図 4 の例のように，添字テーブルへ新たに論理レイアウトの添字に対応する補正レイアウトにおける添字のテーブル，補正添字テーブルを追加する．次元 i のそれぞれの補正値は，次元 i の bitmap 中から，要素を含むサブ配列の経歴値と bit 列と対応している経歴値から補正値演算に用いる bit 列を決定し，次元 i の補参照サブ配列の補正添字 bit までの 1 の bit の総数をカウントすることで求める．なお，それぞれの bitmap 中の bit 列は自次元にて挿入拡張ないし中抜き縮小が行われた際に，挿入拡張されたサブ配列，縮小補正値であれば縮小により開放されたサブ配列の補正添字 bit を 1 とし，拡張，縮小の経歴によっては新たに bit 列を追加する．これらについては次節以降にて詳しく触れる．

3.2.1 拡張補正値の管理

まず，挿入拡張によるオフセットのずれを補正するための拡張補正値について記す．拡張補正 bitmap は複数の bit 列で構成され（図 5），各次元毎に管理される．補正値を求める際は，補正値を求めたい次元の bitmap を参照する．また，bitmap は挿入拡張時の配列の状態によって，値が 0 の新たな bit 列を追加する必要がある．この時，その時点での経歴値カウンタの値を拡張時経歴値として bit 列毎に保持する．補正値を求める際に参照する bit 列の選択には拡張経歴値と要素を含むサブ配列の経歴値を用いる．これは要素を含むサブ配列の割り付けられた時期によって補正箇所の数が異なるためである．

例えば図 4 において，要素 A と要素 B の次元 2 方向の補正箇所を考える．要素 A と要素 B の次元 2 方向の論理添字は同じ 5 であるが，補正箇所は図 4 より，要素 A は次元 2 の添字 2 と 4 の 2 箇所，要素 B は添字 2 の 1 箇所のみ，と異なることが確認できる．本方式では，目的要素を含むサブ配列の経歴値

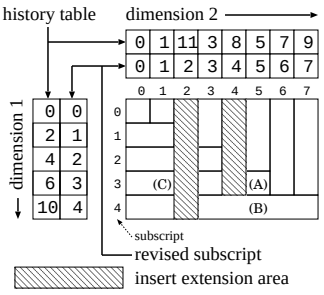


図 4　拡張例：補正レイアウト

を用いて，そのサブ配列における補正箇所情報を持つ bit 列を拡張時経歴値を目印に選択する．また，補正値は補正する要素の論理添字にも依存し，同じ bit 列を参照した場合でも異なった値となる．例えば図 4 の要素 A と要素 C は，同じサブ配列中に存在するため補正値を求める際の bit 列も同じものを参照するが，要素 A の補正箇所は 2 箇所，要素 C の補正箇所は 0 箇所となり異なる．そこで，補正 bit 列は bit の位置を要素の位置情報である補正添字に対応させる．次元 i の補正値を求める際に参照する bit 列の bit$P$ が 1 であった場合は，要素を含むサブ配列が，次元 i に属する補正添字 P のサブ配列の挿入拡張により，ずれが生じている状態であることを意味する．本方式では，次元 i の補正値を求める場合，目的要素の次元 i 方向の補参照サブ配列の補正添字が P であったなら，次元 i の参照 bit 列の bit$P$ までの 1 の bit 総数を補正値とする．なお，拡張補正値を求めるという操作は，拡張補正 bitmap を用いて，目的要素を含むサブ配列が他次元の挿入拡張により影響を受けた箇所を復元することを意味する．次に bitmap を用いて補正値を求める手順を具体的に説明する．

dimension 2

history of insert extension	
	bit sequence
8	0010100
11	0010000

図 5　図 4 の次元 2 の拡張補正 bitmap

図 4 にて要素 A $\langle 3,5\rangle$ を参照する場合について考える．要素 A は次元 2 添字 5 のサブ配列より経歴値の大きい次元 1 添字 3 のサブ配列に含まれる．このときサブ配列内オフセットを求めるには次元 2 の論理添字である 5 を補正する必要がある．図 4 における次元 2 の拡張補正 bitmap を図 5 に示す．次元 2 の補正値を求めるにはまず，図 4 の bitmap 中から参照する bit 列を選択する．上記のように bit 列の選択は，bit 列の拡張経歴値と要素を含むサブ配列の経歴値によって行う．詳しくは，拡張時経歴値が目的要素を含むサブ配列の経歴値を超える値の bit 列の内，拡張時経歴値が最小の bit 列を参照対象の bit 列として選択する．要素 A の補正値を求める場合は，拡張時経歴値がサブ配列の経歴値の 6 を超えている 8 と 11 の bit 列の内，拡張時経歴値の小さい 8 の bit 列を参照対象とする．要素 B を参照する場合は，要素を含むサブ配列の経歴値が 10 であるため，拡張時経歴値 11 の bit 列が対象となる．補正値は要素 A の次元 2 方向の補参照サブ配列の補正添字が 5 なので，対象 bit 列 0010100（左端の bit を bit0 とする）の bit5 までの 1 の bit 総数，すなわち 2 となる．

3.2.2　縮小補正値の管理

次に縮小補正値について記す．縮小補正用の bitmap は基本的な部分は 3.2.1 節で記した拡張補正 bitmap とほぼ同様で，縮小が行われた時点の経歴値カウンタの値を持つ縮小時経歴値とそれに対応する bit 列の組，複数個で構成され，各次元毎に管理される．bit 列の bit の位置とサブ配列の補正添字が対応している点も同様で，縮小補正 bit 列において，bit$Q$ が 1 であるということは補正添字が Q のサブ配列が縮小済みであることを意味する．補正値の求め方も拡張補正値を求める場合とほぼ同様である．次に具体的な例を挙げ説明する．

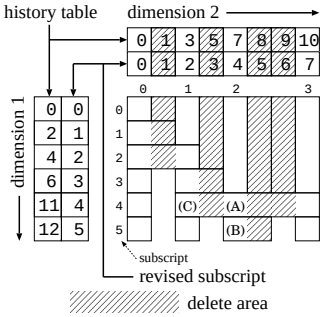


図 6　縮小例：補正レイアウト

図 6 の縮小例について説明する．図 6 は拡張・縮小の経歴やサブ配列のオフセットを見易いように縮小部分を斜線で図示ている．また，図 7 は図 6 から中抜き縮小された部分を除き，実際に拡張可能配列が記憶している情報を図示している．縮小補正値を導くということは図 7 と図 8 の縮小補正 bitmap から図 6 の補正レイアウトを復元するということでもある．

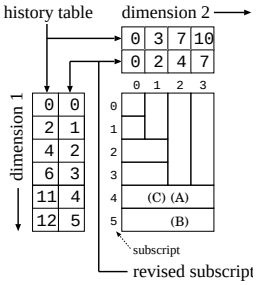


図 7　縮小例：図 6 の論理レイアウト

dimension 2

history of delete	
	bitmap
5	01010110
12	00010110
13	00000100

図 8　図 6 の縮小補正 bitmap

ここで縮小補正値を求める例として要素 A $\langle 4,2\rangle$ を参照する場合を挙げる．まず従来と同様に，要素 A は次元 1，経歴値 11

のサブ配列と次元 2，経歴値 7 のサブ配列の経歴値比較により，次元 1，添字 4，経歴値 11 のサブ配列 S_A に含まれていることがわかる．要素 A のサブ配列内オフセットは，図 7 だけで考えた場合，拡張可能配列の次元数が 2，要素を含むサブ配列 S_A が次元 1 に属するので，従来方式と同様に要素の次元 2 方向の添字 2 がオフセットとなる．しかし，補正後の図 6 から解るように正しいサブ配列内オフセットは 3 である．これは論理レイアウトにおける補助テーブルが拡張・縮小の経歴情報を保持していないためである．

そこで図 8 の縮小補正 bitmap を用いた補正値計算により，次元 2 方向の縮小差分 1 を求める．具体的には，まず要素 A を含むサブ配列 S_A の経歴値が 11 なので，次元 2 の縮小補正 bitmap 中から経歴値 11 を超え，且つ最小の縮小時経歴値と組になる bit 列として，縮小時経歴値 12 の bit 列 00010110 を補正値を求めるために参照する bit 列として選択する．次に次元 2 方向の補参照サブ配列の補正添字である bit4 までの 1 の bit の総数をカウントすることで，縮小補正値 1 が求まる．この縮小補正値 1 を，要素 A の次元 2 方向の添字 2 へ加算し，オフセットは 3 となる．なお，次元数が 3 以上の拡張可能配列であれば，補正済みの添字の組と要素を含むサブ配列の係数ベクトルを用いてオフセット演算を行う．上記の手順で求まったサブ配列内オフセットと S_A の先頭アドレスより，要素 A のアドレスが導かれる．

また，要素 B，要素 C は図 7 の論理レイアウト上では要素 A と隣接しているが，補正値はそれぞれ異なった値である．これは要素 B が格納されている次元 1，添字 5 のサブ配列 S_B は，S_A が影響を受けている次元 2，補正添字 3 の中抜き縮小の影響を受けていないため，S_B 格納要素の補正値を求める場合と S_A 格納要素の補正値を求める場合では参照 bit 列が異なるからである．要素 C は要素 A と同じくサブ配列 S_A に含まれ論理レイアウトでは一見隣接しているが，本来は次元 2 の中抜き縮小により，縮小部分が詰められているため，その部分の補正分，補正値は異なった値となる．

3.3 補正値の求め方（混在する場合）

挿入拡張と縮小が複雑に入り混じった状態での補正値の求め方を具体的な例を用いて説明する．

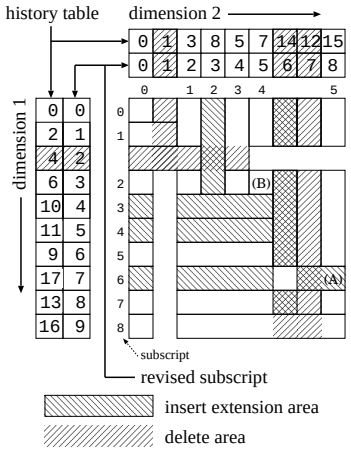


図 9　拡張・縮小例：補正レイアウト

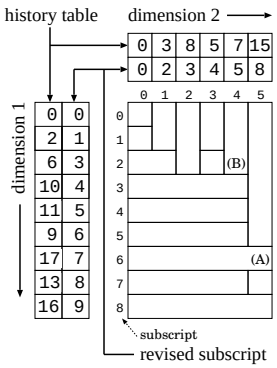


図 10　拡張・縮小例：図 9 の論理レイアウト

dimension 1

history of insert extension	
	bitmap
10	0000110100
17	0000000100

history of delete	
	bitmap
7	0010000000

dimension 2

history of insert extension	
	bitmap
8	000100100
14	000000100

history of delete	
	bitmap
5	010000110
17	000000110
18	000000010

図 11　図 9 の補正 bitmap

図 10 のサイズ $[9, 6]$ の 2 次元配列において，要素 A $\langle 6, 5\rangle$ へアクセスする場合を例として挙げる．図 10 は縮小部を詰めた論理レイアウトであり，補正レイアウトを図 9 に示す．まず，要素を含むサブ配列を求める．次元 1，添字 6 のサブ配列の経歴値は 17，次元 2，添字 5 のサブ配列の経歴値は 15，次元 1 のサブ配列の経歴値のほうが大きいので要素 A は次元 1，添字 6 のサブ配列に格納されている．次に拡張・縮小，それぞれの補正値を求める．要素 A を含むサブ配列の所属次元が 1 なので，補正値は次元 2 について求めればよい．補正値を求める際に参照する bit 列は，要素を含むサブ配列の経歴値を超え，且つその中で最小の拡張時ないし縮小時経歴値と組の bit 列であるため，要素 A の補正値を求める際に参照する縮小補正用 bit 列は，経歴値 17 を超える最小の縮小時経歴値 18 の bit 列 000000010 である．また，拡張補正用 bit 列は次元 2 の拡張補正 bitmap 中に拡張時経歴値が 17 を超える bit 列が存在しないため，拡張補正値は 0 とする．縮小時補正値は bit 列 000000010 に対して，次元 2 の補参照サブ配列の補正添字が 8 なので bit8 までの 1 の bit 総数をカウントすることで求める．bit7 が 1 であり総数が 1 なので縮小補正値は 1 となる．次元 2 の拡張，縮小補正値がそれぞれ求まったので，次元 2 の添字から，拡張補正値を減算，縮小補正値を加算することで，次元 2 方向の添字を補正する．求める次元 2 方向の添字は 5 なので，$5 - 0 + 1 = 6$ へ補正される．図 10 の例では配列の次元数が 2 なのでサブ配列内オフセットはそのまま 6 となる．

次に先と同じ図 10 の 2 次元配列において要素 B $\langle 2, 4\rangle$ にアクセスする場合を例に挙げる．要素 B は，次元 1，添字 2，経歴値 6 のサブ配列と次元 2，添字 4，経歴値 7 のサブ配列の経歴値を比較することで，次元 2，添字 4 のサブ配列に格納され

ていることがわかる．要素を含むサブ配列の所属次元が 2 なので，次元 1 方向の補正値を求める．サブ配列の経歴値が 7 なので，拡張補正値は拡張時経歴値が 10 の bit 列 0000110100 を参照し，縮小補正値は次元 1 の縮小補正 bitmap 中に縮小時経歴値が 7 を超える bit 列が無いため 0 とする．拡張補正値は，次元 1 方向の補参照サブ配列の補正添字が 3 であるため，拡張補正 bit 列 0000110100 の bit3 までの 1 の総数をカウントした 0 が拡張補正値となる．補正値が共に 0 であったため，次元 1 方向の添字は補正前と変わらず 2 のままとなり，オフセットも 2 となる．

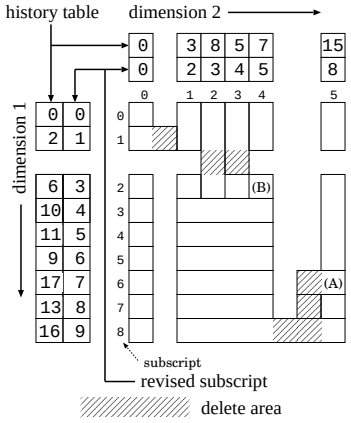


図 12　拡張・縮小例：図 9 の物理レイアウト

3.4　挿入拡張，縮小時の動作

次に挿入拡張・中抜き縮小があった時の拡張可能配列の補助テーブルや補正 bitmap に対する操作を説明する．

- 補助テーブルに対する操作
  - 挿入拡張時

次元 i，添字 j へ新たな値を挿入拡張する場合，まず次元 i の補助テーブルの j 番目以降のテーブル情報をすべて後ろへ 1 シフトする．このとき j 番目以降のテーブル情報の補正添字をシフト時にインクリメントしておく．空いた添字 j のテーブルには挿入拡張により追加されるサブ配列の情報を格納する．挿入サブ配列の補正添字は添字 $j-1$ のサブ配列の補正添字に 1 加えた値とする．

  - 中抜き縮小時

次元 i の添字 j の値を中抜き縮小する場合，次元 i，添字 j のサブ配列本体を開放する．ただし，このとき実際に開放できる領域は次元 i 添字 j 番目のサブ配列本体のみであり，その他の縮小により使用されなくなる領域，例えば図 10 の例では図 12 の斜線領域は開放されない．次に次元 i の補助テーブルの，添字が j 以降のサブ配列情報をそれぞれ一つ前へシフトして，テーブルを詰める．

- 補正 bitmap に対する操作
  - 挿入拡張時

次元 i，添字 j へ新たな値を挿入拡張する場合について記す．次元 i の拡張補正 bitmap，縮小補正 bitmap 双方の全ての bit 列に対し，bit$j$ 以降を 1 右シフトし，bit 列の bit$j$ は，拡張補正 bitmap 中 bit 列は 1，縮小補正 bitmap 中 bit 列は 0 に変更する．

  - 中抜き縮小時

次元 i，添字 j の値を中抜き縮小する場合について記す．次元 i の縮小補正 bitmap 中全ての bit 列に対し，bit$j$ を 1 とする．

ここで実際の実装における bit 列の扱いについて触れておく．それぞれの次元の補正 bitmap 中の bit 列は，その次元の縮小箇所も含めたサイズ，すなわち補正レイアウトでのサイズが bit 列の長さとなるため，配列がある程度大きくなった場合を考慮し，bit 列は long 型変数をいくつか組み合わせて構成されている．bit 列の構造を図 13 に示す．long 型変数は 32（あるいは 64）bit からなり，拡張により自次元のサイズが大きくなり，bit 列追加時に用意した long 型変数では足りなくなった時は，新たな long 型変数を動的に追加する．また，計算速度向上のために，bit 列の long 型配列と同じ長さの char 型配列も確保し，それぞれの char 型変数には対応する long 型変数までの bit 列中の 1 の bit の合計を記憶しておく．例えば図 13 では，char 配列にはそれぞれ 3,5,6 が格納される．bit66 までの 1 の bit の総数を求める場合は，bit63 までの 1 の bit の合計を格納している 2 個目の char 変数の値 5 と，3 個目の long 変数の bit64〜66 までの 1 の bit の総数を加算すればよい．図 13 では bit64,65,66 は全て 0 の bit なので，bit66 までの 1 の bit 総数は $5+0=5$ となる．この実装により，bit 列を格納している変数から 1 の bit 総数をカウントする演算回数が減り，補正値計算の時間短縮が望める．

	bit structure	
	bit line (type 32 bit long)	total sum of 1
0~31	00100000000010100000000000	3
32~63	00000001100000000000000000	5
64~95	00000000010000000000000000	6

図 13　bit 列の構造

- 補正 bitmap へ bit 列を追加するタイミング

補正 bitmap 中の bit 列は挿入拡張・縮小のタイミングにより新たに bit 列を追加する必要がある．以下に拡張補正 bitmap，縮小補正 bitmap において新たに bit 列を追加するタイミングについて記す．また，追加される bit 列の値は 0 である．

  - 拡張補正 bitmap

次元 i の拡張補正 bitmap は次元 i の拡張 E_{i1} と，E_{i1} の前に次元 i にて行われた拡張 E_{i2} との間に，i 以外の次元で挿入拡張 E_j が行われていた場合に新たに bit 列を追加する．bit 列は拡張 E_{i1} が行われるタイミングで追加し，追加した bit 列の挿入拡張時経歴値は拡張 E_{i1} が行われたときの拡張可能配列の経歴値カウンタの値となる．

  - 縮小補正 bitmap

縮小補正 bitmap は次元 i の縮小 D_{i1} と，D_{i1} の前に次元 i にて行われた縮小 D_{i2} との間に，i 以外の次元で拡張 E_j が行われていた場合に新たに bit 列を追加する．bit 列は縮小 D_{i1} が行われるタイミングで追加し，追加する bit 列の縮小時経歴値は縮小 D_{i1} が行われたときの拡張可能配列の経歴値カウンタの値となる．

4. 評　　価

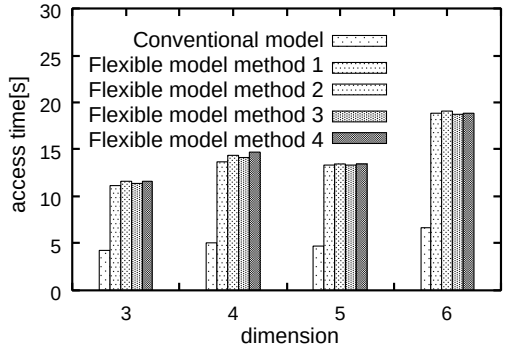
4.1 従来の方式との比較

従来の拡張可能配列と本方式である挿入拡張・中抜き縮小を可能とする拡張可能配列を比較し，評価を行う．

4.1.1 時間コスト比較

まず時間コスト比較として単一要素へのアクセス速度の比較を行う．次元数が n の拡張可能配列において，従来方式と本方式での要素へのアクセス時の操作を考える．本方式におけるアクセス時の操作は，従来方式でのアクセス時の操作に補正値を求めるという操作を加えたものとなる．この補正値演算操作は，1 回の要素アクセスにて $(n-1)$ 回の bit 列中の 1 の bit の総数演算が必要である．補正値演算操作の手間は絶対的には大きくないが，要素へのアクセス操作全体と比較すると決して小さいと言えず，アクセス速度の低下は避けられない．

実際に測定した結果のグラフを図 14 に示す．図 14 は要素数が m の多次元配列に対し，$m/10$ 回要素を参照した際にアクセスに要した時間のグラフである．また，本方式においては縮小，挿入拡張の回数に依るアクセス速度への影響を調べるために，縮小と挿入拡張が共に行われていない状態の方式 1，縮小が何度か行われている状態の方式 2，挿入拡張が何度か行われている状態の方式 3，共に行われた状態の方式 4 の 4 種類の状態で測定を行う．なお，これら 4 種類の状態はそれぞれ次元数，サイズ，要素数が同じであり，縮小，挿入拡張を行っている状態ではそれぞれ各次元に対し，各次元のサイズの 1/10 回行っている．



次元数（次元サイズ）	3（400）	4（90）	5（35）	6（20）
配列全体の要素数	6400 万	6561 万	5252 万	6400 万

※従来方式（挿入拡張 不可，縮小 不可）
本方式 1（挿入拡張 無，縮小 無），本方式 2（挿入拡張 無，縮小 有）
本方式 3（挿入拡張 有，縮小 無），本方式 4（挿入拡張 有，縮小 有）

図 14　時間コスト比較：ランダムアクセス

図 14 より本方式にてアクセスに要する時間は，従来方式にて要した時間の約 2.8 倍であることが確認できる．この従来方式，本方式のアクセスに要する時間の差は，補正値演算時の bitmap 操作と新たなテーブルの表引き操作に依るものである．また，従来方式，本方式共に次元数が大きくなるにつれ，1 回のアクセスに要する時間が増加している．これは配列の次元数が大きくなることで，サブ配列の次元数も大きくなり，アドレス計算のための乗算および，本方式においては補正個所が増えるためである．なお，本方式における縮小，挿入拡張の有無に依る時間変化はほぼ見られず，縮小，挿入拡張回数に依るランダムアクセスにおける要素へのアクセス速度低下は予想されない．

単一要素へのアクセス時間については，従来方式に比べ本方式では約 2.8 倍と好ましくない結果であったが，多次元配列の格納データ構造として考えた場合，重要となるのは単一要素へのアクセス速度ではなく，MOLAP 等で頻繁に使用されるスライス検索等の範囲要素へのアクセス速度である．次に範囲アクセスについて比較，評価を行う．なお，従来方式，本方式共に，範囲アクセスでは範囲内の要素を含むサブ配列毎に要素へのアクセスを行う．このため係数ベクトルを用いたサブ配列内オフセットの演算は，サブ配列単位で再帰的に行うことができ，乗算回数を大幅に減らすことができる．これにより従来方式，本方式共に要素アクセスの平均時間は大幅に短縮される．また，それに加え，同一サブ配列内では補正値を共用できるため本方式における補正値演算回数も大幅に減らすことができ，従来方式との速度の差も減少する．なお，次元数が n，各次元サイズが m のサブ配列 S 全体がアクセス範囲内に含まれる場合，サブ配列 S 全体の要素をアクセスする際に必要な演算回数は，係数ベクトルを用いたサブ配列内オフセット演算の乗算回数が $m^{(n-1)}$ 回，補正値演算の回数が m^n 回である．

次に範囲アクセスの速度比較として，多次元配列の全要素を参照したときのアクセスに要する時間を図 15 に示す．図 15 では従来方式と本方式について比較を行うが，図 15 におけるランダムアクセス速度比較のとき同様に，本方式については縮小，挿入拡張回数の違う 4 種類について測定を行っている．

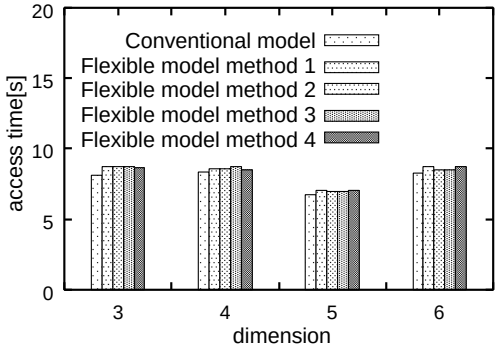


図 15　時間コスト比較：シーケンシャルアクセス

図 15 より，範囲アクセスでは従来方式と比較し，本方式にてアクセスに必要な時間は，従来方式と 5% ほどしか差が無いことが確認できる．ランダムアクセス時は 2.8 倍であったことを考えると比較的有利な結果となっている．また，要素アクセスの平均時間はランダムアクセスと比べ，範囲アクセスでは従来方式では約 25%，本方式では約 8% と大幅に減少している．これらの結果はスライス検索等，範囲アクセスが重要となる多次元配列では非常に望ましい結果といえる．なお，従来方式，本方式共に拡張可能配列配列自身の次元数の増加による速度低下もほとんど見られない．挿入拡張・中抜き縮小の回数に拠る速度低下や，挿入拡張によるサブ配列単位の検索の複雑化に依る速度低下も見られず，アクセス速度は挿入拡張，縮小の有無に影響を受けていないと言える．

4.1.2 空間コスト比較

空間コストについて比較する．従来の拡張可能配列では，サブ配列自身の領域に加え，補助テーブルとしてサブ配列毎にサブ配列本体の先頭アドレス，経歴値，自身の形状を記憶する係数テーブル（次元数 – 2 サイズの配列）を保持している．本方式でも同様に，サブ配列本体の領域，補助テーブルを保持する．本方式における補助テーブルは従来の方式の補助テーブルのパラメータである配列本体へのアドレス，経歴値，係数テーブルに加え，自身の論理的な位置を記す補正添字テーブルで構成される．また，その他に補正 bitmap を次元毎に保持する．

表 1　空間コスト比較：配列本体

次元数（次元サイズ）	3（400）	4（90）	5（35）	6（20）
配列全体の要素数	6400 万	6561 万	5252 万	6400 万
従来方式	245MB	252MB	201MB	245MB
本方式	245MB	251MB	201MB	245MB
従来方式 補助テーブル	約 25kB	約 9kB	約 5kB	約 4kB
本方式 補助テーブル	約 30kB	約 10kB	約 6kB	約 5kB
未開放領域	0.61MB	2.78MB	5.724MB	12.297MB

空間コストについての具体的な例を表 1 に示す．表 1 は配列要素を long 型（32bit）として，多次元配列を確保したときの配列全体と補助テーブルの領域サイズである．表 1 よりわかるように，補助テーブルサイズは，数十〜数ｋ B であり配列全体と比較して非常に小さい．よって従来方式と本方式の補助テーブルの空間コスト差は配列全体から見た場合，無視できるサイズであると言える．bitmap のサイズは bitmap 中の bit 列の本数 × bit 列のサイズで計算できる．bit 列のサイズは，挿入拡張もしくは縮小が行われ bit 列が追加される時点の配列のサイズから求まり，配列のサイズにより変動する．表 1 の次元 3，各次元サイズ 400 のサブ配列において，最も bit 列のサイズが大きくなる条件を考えた場合，追加される bit 列のサイズは 78Byte ほどである．また，bit 列の本数も挿入拡張や縮小の行われた経歴により異なるが，最大でも縮小回数，挿入拡張回数と同数である．以上より，bitmap のサイズも配列全体のサイズと比較した場合では，ほとんど無視できるサイズである．

一方，中抜き縮小，挿入拡張を実装した本方式において，従来方式と空間コストにおいて大きく差がでる要因として中抜き縮小の回数と行われたタイミングがある．これは，次元 i，添字 j の値の中抜き縮小を行うことで実際に開放できる領域が，次元 i，添字 j のサブ配列領域だけであり，他の領域は自サブ配列が開放されるまでは開放できないためである．以後値の中抜き縮小により開放することのできない縮小領域を未使用領域と呼ぶ．未使用領域のサイズは，配列の次元数が大きくなるほど肥大化し，また，初期からある値の中抜き縮小した際に大きな値となる．表 1 にて，各条件において最も未使用領域が大きくなる場合のサイズを示す．なお，表 1 中の未使用領域サイズは最悪の条件の場合であり，その他の条件ではかなり緩和される（最良の場合は 0 である）．ただし，縮小が繰り返し行われるにつれ，未使用領域が増えることは確実であるため，ある時点にて配列の再配置を行う必要があると考えられる．

5. ま と め

本稿では従来の拡張可能配列に，挿入拡張，中抜き縮小操作を導入することで，より柔軟なサイズ変更が可能な多次元配列の実現方式を提案した．本方式では挿入拡張，中抜き縮小の補正値演算を，bit 列を用いることで，挿入拡張，中抜き縮小の回数に依らずランダムに計算することが可能である．これにより，MOLAP 等で重要なスライス検索などの範囲アクセスを，従来方式とほぼ同等の速度で行うことできる．空間コストも bitmap や従来の拡張可能配列の性質を用いることで，肥大化を抑えている．

今後は，サーバにおかれた本方式に基づく拡張可能配列をクライアント側で部分的・動的に共有しながら，その共有部分とクライアントに局所的に確保した部分とを併せて，論理的に 1 つの拡張可能な配列として取り扱えるような拡張可能共有配列の機構について考察したい．

文　献

[1] K.E.Seamons and M.Winslett, 'Physical Schemas for Large Multidimensional Arrays in Scientific Computing Applications', Proc. of 7-th International Working Conference on Scientific and Statistical Database Management, 218-227, 1994.

[2] S.Sarawagi and M.Stonebraker, 'Efficient Organization of Large Multidimendional Arrays', Proc. of International Conference on Data Engineering, 328–336,1994.

[3] H.Gupta, V.Harinarayan, A.Rajaraman, and J.D.Ullman, 'Index Selection for OLAP', Proc. of 13-th International Conference on Data Engineering, 208-219, 1997.

[4] 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.

[5] N.Widmann, and 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.

[6] 都司，一色，樋口，宝珍，'MOLAP のための多次元配列の実現方式とその性能評価', 電子情報通信学会論文誌 D-I, Vol.J87-D-I, No.2, pp.244-255, 2004.

[7] Niklaus Wirth, 'Programming in Modula-2', Springer-Verlag, 1983.

[8] A.L.Rosenberg, 'Allocating Storage for Extendible Arrays', JACM, Vol.21, 652-670, 1974.

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

[10] E.J.Otoo and T.H.Merrett, 'A Storage Scheme for Extendible Arrays', Computing, Vol.31, 1-9, 1983.

[11] A.Novacek, 'Using Time Stamps for Storing and Addressing Extendible Arrays', Computing, Vol.37, 303-313, 1986.

[12] D.Rotem and J.L.Zhao, 'Extendible Arrays for Statistical Databases and OLAP Applications', Proc. of 7-th International Working Conference on Scientific and Statistical Database Management, 108-117, 1996.

[13] T.Tsuji, H.Kawahar, K.Higuchi, T.Hochin, 'Sharing Extendible Arrays in a Distributed Environment', Lecture Notes in Computer Science 2060, pp41-53, 2001

[14] 都司，水野，宝珍，樋口: ' 拡張可能配列の遅延割付け方式', 電子情報通信学会論文誌 D-I, Vol.J86-D-I, No.5, pp.351-356, 2003.