DEWS2004 4-A-04

MOLAPのための拡張可能配列

原　章広† 水野　剛† 都司　達夫†† 樋口　健††

† 福井大学大学院工学研究科 〒910-8507 福井県福井市文京 3-9-1
†† 福井大学工学部 〒910-8507 福井県福井市文京 3-9-1
E-mail: †{hara,mizuno,tsuji,higuchi}@pear.fuis.fukui-u.ac.jp

あらまし MOLAP では基幹系のバックエンドシステムに蓄積された関係データベーステーブルのスナップショットがファクトテーブルとして多次元配列に格納される．多次元配列の高速なランダムアクセス性を生かして，ファクトテーブルに対して，集約演算をはじめ種々の統計処理演算を効率よく行うことができる．この高速アクセス性は配列の各次元サイズが固定であることに依っており，したがって，新たなカラム値を追加することができない．通常，バックエンドシステムからのスナップショットはより大きい固定サイズ配列を用意して，そこに新たにダンプされる．本論文では新たなカラム値の追加に対して，当該次元のサイズを動的に拡張できる拡張可能配列システムを MOLAP のための基盤として提供する．これにより，前回ダンプした配列データやデータキューブを再配置することなしに，インクリメンタルな集約演算が可能となる．MOLAP の基盤として，本システムを機能させるための問題点を述べ，その対策法を提案する．

キーワード MOLAP，多次元配列，拡張可能配列

Extendible Arrays for MOLAP

Akihiro HARA†, Go MIZUNO†, Tatsuo TSUJI††, and Ken HIGUCHI††

† Graduate School of Engineering, Fukui University, Fukui-shi, 910-8507, Japan
†† Faculty of Engineering, Fukui University, Fukui-shi, 910-8507, Japan
E-mail: †{hara,mizuno,tsuji,higuchi}@pear.fuis.fukui-u.ac.jp

Abstract In MOLAP systems, multidimensional arrays are employed to store fact tables extracted from the back-end relational database. On these fact tables, various kinds of statistical computations including aggregation can be performed efficiently by invoking the fast random element accessing capability of arrays.But, if a new column value emerges after constructing the fact table in a usual fixed size array, it cannot involve the value. In this paper, we provide an extendible multidimensional array system for MOLAP. Such an array can extend its size dynamically along an arbitrary dimension without any relocation of existing data. This property enables an incremental aggregating without relocating any array data or data cube dumped at the last time. Some problems of this system in working as the basis for MOLAP are stated, and their countermeasures are proposed.

Key words MOLAP, multidimensional array, extendible array

1. まえがき

近年，多次元データベースを用いた MOLAP の研究が盛んに行われるようになり (例えば [1] [2])，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だしてきている．MOLAP システムでは，基幹系システムに蓄積された関係データベーステーブルのスナップショットがファクトテーブルとして多次元配列にダンプされる．多次元配列の高速なランダムアクセス性を生かして，ファクトテーブルに対して，集約演算をはじめ種々の統計演算処理を効率よく行うことができるが，この高速アクセス性は配列の各次元サイズが固定であることに依っている．通常，基幹系システムからのスナップショットは定期的にダンプされるが，その都度，固定サイズ配列を用意して，そこに新たにファクトテーブルとしてダンプし，多次元分析に利用する．

本研究では，新たなカラム値の追加に対して，現在存在している配列データを再配置することなく対応次元の配列サイズを拡張することができる多次元配列を用いて MOLAP のための基盤データ構造を提案する．このような配列は拡張可能配列 [4] [5] [6] と呼ばれる．これにより，前回ダンプしたファクト

テーブルやそれから派生するデータキューブに対して，前回以降，新たに追加されたデータレコードのみをファクトテーブルに追加したり，インクリメンタルな集約演算により，新たなデータキューブを生成することが可能となる．

一般に，MOLAP システムにおいて用いられる従来の固定サイズの多次元配列には次のような問題がある．

(1) 基幹系システムの関係テーブルの各カラムを多次元配列の各次元に対応させた場合，テーブルレコードは多次元配列の要素を指定する配列添字の組で表されるが，通常，全配列要素の数に対して極端に少ない（疎配列問題）．関係データベースのテーブルをそのまま SQL 演算を使って分析する ROLAP (Relational OLAP) では，このような問題は存在しない．

(2) 基幹系システムの関係テーブルの各カラムはデータ型を持つが，配列要素を指定する添字の組は，通常，非負整数値である．したがって，関係テーブルの各カラム値を非負整数値に変換する機構が必要となる．

また，拡張可能配列を MOLAP システムの基盤データ構造として機能させるためには次のような問題点がある．

(3) 従来の固定サイズ配列と比較して，拡張可能配列は要素アクセス速度および記憶効率の低下を最小限に抑える必要がある．通常，任意の次元に沿って拡張可能であることとこれら 2 つの性能を低下させないことを両立させることは困難である．

(4) 上記問題点 (3) に関して，拡張可能配列においても同様に変換機構が必要となる．しかし，拡張可能配列の場合には，新たなカラム値が入力される時間順に配列添字に順にマッピングされる．したがって，カラム値の順がマッピングされた添字の値順と一致しないので，カラム値の範囲検索が不利になる．

本論文では，[6] に沿って拡張可能配列の主記憶上の実現モデルを述べた後，上記 (1)〜(4) の問題点に対する解決策を提案し，二次記憶上における本システムの実現について述べる．なお，基幹系システムの関係テーブルとは異なり，ある時点でのテーブルレコードを拡張可能配列へインクリメンタルに追加するのみであり，MOLAP の性質上，レコードの更新や削除はない．

2. 拡張可能配列の主記憶上の実現モデル

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動的に割付け，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，A のサブ配列と呼ぶ．

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 $[s_1, s_3, s_4]$ の各次元サイズが固定の通常の 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 $\langle i_1, i_2, i_3 \rangle$ のアドレスは良く知られているように，1 次関数

$$s_3 s_4 i_1 + s_4 i_2 + i_3$$

となる．この $(s_3 s_4, s_4)$ を S の係数ベクトルと呼ぶ．A の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．

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 $[3, 4]$ は S では要素 $\langle 3 \rangle$ であるので，求めるアドレスは 63 となる．

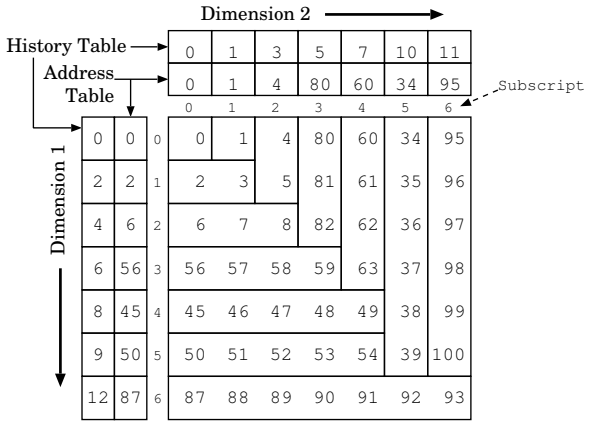

図 1 主記憶上の拡張可能配列の実現モデル

このインデックス配列モデルは補助テーブル用の記憶域を付加することにより高速に配列要素を参照することができる．ハッシュ関数を利用した他の方式 [4] より記憶コストと要素アクセスコストの両面で優れている [5]．しかし，上記の 3 つの補助テーブルの合計の記憶コストは $O(n^2)$ であり，特に，係数テーブルは $(n-2)$ 次元の係数ベクトルを収容する必要があるのでそのコストは大きい．MOLAP においては次元数 n および次元サイズともに大きくなるので，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．[6] では，主記憶上の拡張可能配列に対して係数テーブルのスロットの遅延割付手法を提案して補助テーブルサイズの低減を図っているが，二次記憶上の拡張可能配列に対してはより抜本的な対策が必要になる．

3. 提案する実現モデル

2. 節で提案した実現モデルも含めて，従来の拡張可能配列の実現モデルでは，配列本体は主記憶上に確保されることを前提とした．次元数も多く大規模な配列を扱うことが多い MOLAP アプリケーションのためには，配列を二次記憶上に配置する必要がある．しかし，拡張可能配列を二次記憶上の永続データとして格納するときは，主記憶上で展開したデータ構造がそのま

ま二次記憶においても有効であるとは限らない．二次記憶のアクセス特性を十分考慮してデータ構造を検討する必要がある．本節ではこの永続化の問題も含めて，1. 節の各問題を解決するための拡張可能配列の実現モデルを提案する．

3.1 チャンク化

n 次元拡張可能配列の実現モデルでは配列添字の拡張に対して，配列要素の集合としての $n-1$ 次元サブ配列がメモリ上に割り付けられた．添字の組による配列要素の指定は 2. 節で述べたアドレス計算の手続きにより，直接，主記憶上の配列要素のアドレスにマッピングされた．ここでは，このサブ配列をチャンクと呼ぶ各次元サイズが同じである n 次元超立方体の集合に分割する．拡張可能配列を構成するチャンク集合は単一の二次記憶ファイルに格納される．2. 節における配列要素はここでは，チャンクに対応させる．2. 節で述べた 3 種類の補助テーブルはチャンクを要素とする配列（以後，チャンク配列と言う）に対して確保される．したがって，各補助テーブルのサイズは大幅に削減でき，主記憶上に配置することが可能になる．チャンク c の論理的なアドレスは c が所属するチャンクサブ配列を S_c として (S_c の経歴値，S_c における c の位置) なる対で指定される．ここで，S_c 内のチャンクは 2. 節で述べた配列要素のサブ配列の場合と同様にあらかじめ定められた次元順に並べられる．

3.2 疎配列問題への対応

1. 節で示した疎配列問題に対しては，チャンクオフセットと呼ばれる方式 [1] により圧縮する．チャンク内の要素はあらかじめ定められた次元順に記憶領域に配置される．これは有効要素についてのみ，そのチャンク内でのオフセットとデータ値の対の集合を格納する圧縮法である（図 2）．この対集合はオフセット値の順にソートされている．有効配列要素の取り出しは二次記憶領域中の圧縮チャンクを主記憶上にロードし，バイナリサーチすることにより決定される．

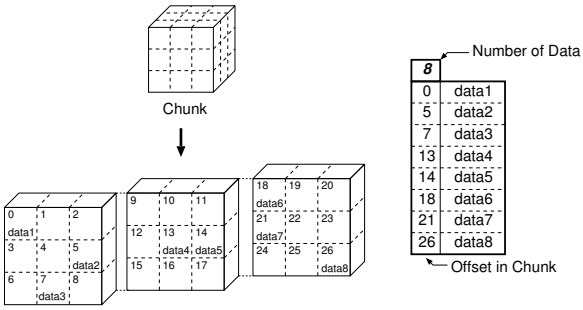

図 2　チャンクオフセット圧縮

3.3 カラム値の変換と配列要素のアドレス計算

関係テーブルの各カラムごとに，そのカラム値から拡張可能配列の添字の値に変換するために B^+ 木を使用する．カラム値の組を指定した配列要素のランダムアクセスおよびカラム値の範囲検索共に適応性があるからである．これらの B^+ 木のシーケンスセットの各ノードには 3.1 節でのチャンク単位の拡張可能配列の経歴値と B^+木 の位数以上，位数の 2 倍以下の（カラム値，添字値）対が格納されている（図 3）．3.1 節で述べたチャンクの各次元サイズは，これら B^+ 木の位数の 2 倍のサイズとしている．

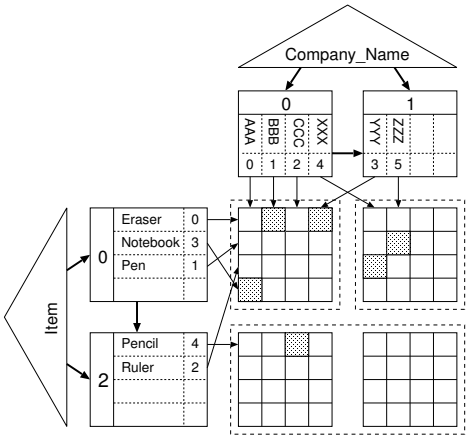

図 3　カラム値から添字値への変換

n カラムからなるテーブルレコードの各カラム値 $v_1, v_2, \ldots, v_n$ が指定されたときの該当する配列要素のアドレス計算は次のように行われる．チャンクの各次元サイズは $[q_1, q_2, \ldots, q_n]$ とする．

(i)　S_c 内のチャンク番号の計算

まず，各カラムの B^+ 木を検索して，カラム値が格納されているシーケンスセットノード内の添字値 $i_1, i_2, \ldots, i_n$ を得る．$\langle i_1/q_1, i_2/q_2, \ldots, i_n/q_n \rangle$（ただし，/は割り算の商を表す）により，配列要素が格納されているチャンクの添字値の組を求める．この添字値の組から各 B^+ 木のシーケンスセットをたどり，該当するノードの経歴値の組歴値 $h_1, h_2, \ldots, h_n$ を求める．この経歴値集合より，2. で述べた手順に従って，要素 $\langle i_1, i_2, \ldots, i_n \rangle$ を有するチャンクが含まれるチャンクサブ配列 S_c を求める．

S_c が次元 k に属しているとすると，$\langle i_1/q_1, \ldots, i_{k-1}/q_{k-1}, i_{k+1}/q_{k+1}, \ldots, i_n/q_n \rangle$（ただし，/は割り算の商を表す）により，配列要素が格納されているチャンクの S_c における添字値の組を求める．S_c 内ではチャンクは次元順に配置されるので，アドレス関数を使用して，得られた添字の組に基づいて，S_c 内のチャンク番号を求める．

(ii)　チャンク内オフセットの計算

$\langle i_1 \% q_1, i_2 \% q_2, \ldots, i_n \% q_n \rangle$（ただし，%は割り算の余りを表す）は (i) で求めたチャンクにおいてアクセスする配列要素の添字値の組となる．チャンク内要素の位置を求めるアドレス関数を使って，求める配列要素のチャンク内オフセットを計算する．

(iii)　配列要素の決定

(i) で求めたチャンクを主記憶上にロードする．これは，3.2 節で述べたように有効要素のチャンクオフセットテーブルであり，要素のチャンク内での位置の順にソートされている．このテーブルについてバイナリサーチを行い，求めるチャンクオフセットのデータ値を決定する．

ただし，本節で述べた実装には 1. 節の (4) で述べた難点があり，カラム値の範囲検索については不利である．次節では，この難点を克服した方式を提案する．

3.4 レコードの追加

関係テーブルのレコードは前回のインクリメンタルなダンプ

以降に追加されたレコードが現在の拡張可能配列に追加される．このとき，1. の問題点 (4) で述べたように，カラム値の範囲検索が不利になる．例えば，図 3 に示すように，B^+ 木には，新たなカラム値が入力される時間順に配列添字に順にマッピングされる．したがって，カラム値の順がマッピングされた添字の値順と一致しないので，一般にシーケンスセットの 1 つのノード中の添字値集合が複数のチャンクに割り当てられるので，カラム値の範囲を逐次検索するには，多くのチャンクをアクセスする必要がある．これは指定された範囲のカラム値に対する集約演算のパフォーマンスが低下することを意味する．なお，関係テーブルのレコードをその時点ですべて固定配列に格納し，以後のカラム値の追加を許さない通常の MOLAP システムではカラム値と添字値の順を一致させることが可能であり，このような問題は発生しない．

この問題に対して，本研究ではカラム値と添字値の変換機構として用いる B^+ 木のシーケンスセットのノードが新たなカラム値の挿入により分割されるのに同期して，関連するチャンクデータの分割と再配分を局所的に行う方式を提案する．

追加レコードの各カラム値がいずれも対応する B^+ 木に既登録であれば，既存のチャンクのいずれかにファクトデータを登録可能であり，拡張可能配列本体の拡張は行われない．また，B^+ 木に登録されていないカラム値が存在するときでも，そのシーケンスセットノードに空きがある場合には，対応するチャンクについて未使用の添字値が存在するので，その一つをカラム値に対応付ける．空きがない場合には，そのカラム値の追加により，シーケンスセットの当該ノード $n1$ にあふれが生じる（図 4）．この場合，新たなノード $n2$ が割り付けられる．また，それに同期して，当該次元方向に対して拡張可能配列本体が 1 チャンク分拡張され，ノード $n2$ に対応して，新たなチャンク集合が割り付けられる．ノード $n2$ にはノード $n1$ 中の（カラム値，添字値）の対集合の半分が移動する．この時，移動の対象となる対は $n2$ に移動後，その添字値が更新される．続いて，更新された添字値を持つ配列本体の要素は $n2$ に対応するチャンク集合に移動する．図 5 にこの様子を示す．

以上の方式により，1 つのノードに含まれるカラム値に対する添字値集合は複数のチャンクにまたがることなく，同一のチャンクに限定させることが可能である．これにより，1. 節の問題点 (4) による範囲検索の速度劣化を軽減することができる．なお，図 3 では，マッピングされる添字の値を拡張可能配列の $\langle 0,\ldots,0\rangle$ 要素を基点として付してあるのに対して，図 4 や 5 ではマッピングされたノード内に相対的に付している．これは，1 つのノードに含まれるカラム値に対する添字値集合を同一のチャンクに限定できることに依っている．

本節で提案したデータ構造による要素アドレスの計算アルゴリズムを以下に示す．

(i)　チャンク番号の計算

各カラムの B^+ 木を検索して，カラム値が格納されているノードを求め，対応する経歴値の組を求める．この経歴値集合より，2. 節で述べた手順に従って，要素を有するチャンクが含まれるサブ配列を得る．このサブ配列に対し，アドレス関数を用いてチャンク番号を求める．

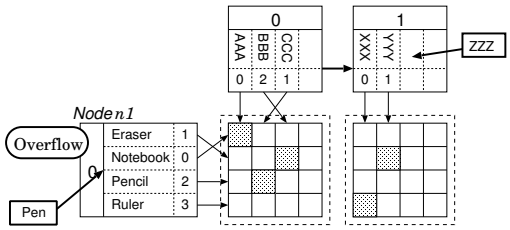

図 4　シーケンスセットノードのオーバーフロー

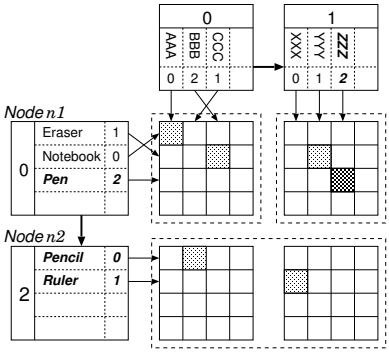

図 5　チャンクデータの分割と局所的再配分

(ii)　チャンク内オフセットの計算

シーケンスセットの各ノードに格納されている (カラム値，添字値) 対をもとに，アクセスする配列要素の添字値の組を求める．その後，チャンク内要素の位置を求めるアドレス関数を用いて，求める配列要素のチャンク内オフセットを計算する．

(iii)　配列要素の決定

3. 3 節で示したアルゴリズムと同様に，(i) で求めたチャンクを主記憶上にロードし，読みだしたオフセットテーブルについてバイナリサーチを行い，求めるデータ値を決定する．

ただし，シーケンスセットノードの分割と再配分により，配列本体要素の移動のオーバヘッドが引き起こされる．このオーバヘッドの軽減については次節で述べる．

4.　入力バッファリング

前節において，新たなカラム値の登録による B^+ 木のシーケンスセットノードの分割と再配分に同期して，拡張可能配列本体を拡張して，関連するチャンクオフセット集合のみを局所的に移動する方式を提案した．通常の固定配列を使用したときのようにより大きな配列を生成して，そこへ元の配列全体を再配置・移動する必要はない．しかし，局所的であるとは言え，基幹系システムの関係テーブルの内容と多次元拡張可能配列との間に実時間的な一致が要求される場合には，非常に大きなオーバヘッドが必要とされる．新たなカラム値の追加を直ちに提案データ構造に反映する必要があるので，ノードの分割や再配分が引き起こされるたびに拡張可能配列本体の関連するチャンクオフセット集合を移動する必要があるからである．

しかし，通常 MOLAP では，ある一定期間毎に基幹系システムの関係データベーステーブルがダンプされ，それが分析に使用されるので，このような実時間性は必要とされない．そこで，ここでは，基幹系システムと本 MOLAP 用システムとの間

の連携による次のような入力データのバッファリング機能を用いて，このオーバヘッドを軽減させることとする．これはバッチ的な登録処理が可能であるために，(カラム値，添字値) 対の B^+ 木への追加登録とチャンクオフセットデータの追加登録が独立して行えることに注目している．

基幹系システムでは，前回ダンプ時以降に新しく追加されたレコードを関係テーブルに追加すると共に自システム内のバッファ**B** に別に登録しておく．本 MOLAP 用システムでは次回のダンプ時には，このバッファ内のレコードをスキャンして，次の 2 段階により，前節で提案したデータ構造に登録する．

(1) B^+ 木へのカラム値の追加登録

バッファ**B** 中の各レコードのカラムについて，対応する次元の B^+ 木にカラム値が既登録でなければ，その添字値と共に新たに登録する．この時，3.4 節で述べたカラム値の追加アルゴリズムの内，シーケンスセットノードへの (カラム値，添字値) 対の追加・更新処理が行われる．レコードの追加により，オーバフローを起こしたノードについて，対応するチャンクサブ配列 S_c を分割・再配分する．このとき，拡張可能配列本体が拡張されるのにともなって，S_c のチャンクデータが拡張分のチャンクに移動する．ただし，この分割と移動は前回ダンプ時のチャンクデータに限定される．

(2) チャンクオフセットデータの追加登録

バッファ**B** 中の各レコードを再度，最初からスキャンする．レコードの各カラム値について (1) で追加登録した各次元の B^+ 木を検索して，添字値の組を求めて，3.4 節で述べた配列要素のアドレス計算アルゴリズムを実行して，追加するべきチャンクとチャンク内オフセットを決定し，その要素のチャンクデータをチャンクに挿入する．必要とあればその要素のファクトデータを更新する．

集約値の計算については，上記 (2) の段階において，バッファ**B** 中のレコードについてのみ計算し，その結果を前回ダンプ時の集約値と併せて，最新の集約値に更新可能である．以上の方式により，シーケンスセットノードの分割・再配分に伴う拡張可能配列本体のチャンク集合に対するチャンクオフセット集合の再配分・移動の回数は最小限に抑制される．前回ダンプ時のシーケンスセットノードに関連する変更に対してのみ，対応するチャンクオフセット集合を再配分・移動すればよい．新たに追加されたノードに関連するチャンクについては，(1) の段階で得られた B^+ 木によるマッピングに従って，当該チャンクにチャンクオフセットを登録すればよい．基幹系システムの関係テーブルとの実時間での一致を考慮するときに必要なチャンクオフセット集合の再配分・移動を繰り返す必要がない．

5. 検　　索

この節では，前節までに述べた n 次元拡張可能配列システムを用いて多次元拡張可能配列にダンプされたデータに対する検索の方法について述べる．MOLAP では，多次元配列に対し，いくつかの次元について検索条件を与えることで部分配列を指定し，その配列中のファクトデータに種々の統計処理を施す．n 次元の多次元配列において，$n-m$ 個の次元にそれぞれ 1 個のカラム値を条件として与え，m 次元の部分配列を取り出す操作は m 次元スライスと呼ばれる．また，検索条件として特定の次元がカラム値の幅を持つような範囲検索も統計処理には重要である．本システムは，任意の次元に対応するカラムについて，任意のカラム値の幅を持つ範囲検索を効率よく行う検索処理システムを提供する．

5.1 範囲検索処理

検索条件は各カラムごとの検索の始点および終点となるカラム値の対の集合で表現される．m 次元スライス検索を行う時は $n-m$ 個の次元のカラム値を指定して，他の m 個の次元には"*"を指定する．この時，始点と終点のカラム値は対応する次元の B^+ 木のシーケンスセットの先頭および最後尾のカラム値となる．

ある次元についての範囲検索は，まず対応する B^+ 木を用いて，範囲検索の始点と終点のカラム値が含まれているシーケンスセットノードおよびそれぞれに対応する添字に変換する．そして，始点から終点の範囲でシーケンスセットをたどりながら，他の次元のカラム値に対する添字も含めて，n 次元の添字の組から，その要素が含まれるチャンクとそのチャンク内オフセットを順次求める．オフセットを求めるアルゴリズムは 3.4 節の要素アドレス計算アルゴリズムと同様である．チャンクに格納されているオフセットテーブルを主記憶上にロードし，求めたオフセットで表される要素が登録されているかどうか調べる．登録されており，したがって有効要素であった場合はこの要素を検索結果に加える．検索範囲内の全てのカラム値の組合せについて調べ終った時点で検索処理は終了する．

ここで，添字の値を変化させる方法に注意が必要である．範囲検索の過程において調べる対象となるチャンクが変わった際には，チャンクに格納されているオフセットテーブルを二次記憶から新たに読み出さなければならず，検索の所要時間に大きな影響を及ぼす．そのため，チャンク間の移動の回数を減らし，二次記憶からオフセットテーブルを読み出す回数をできるだけ少なくする必要がある．

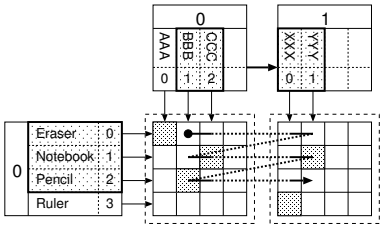

図 6　非効率的な要素探索の方式

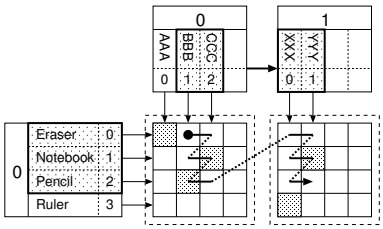

図 7　チャンクに特化した要素探索の方式

図 6 及び図 7 は，(“Eraser”～“Pencil”,“BBB”～“YYY”) という同一の検索条件に対して要素を探索する順番を表す．この時，図 6 のように始点から終点まで連続して探索する方式をとると，チャンク間の移動回数が増え，何度も同じチャンクをロードする必要があるので効率が悪い．そこで，図 7 のように 1 つのチャンクに対する探索をまとめて行うことで，チャンク間の移動回数を抑え，検索所要時間を短縮させることができる．探索をまとめて行うことができるのは，3.4 節の冒頭で述べた不利な点を解消し，1 つのノードに含まれるカラム値に対する添字値集合は複数のチャンクにまたがることなく，同一のチャンクに限定させることできるからである．

5.2 範囲検索アルゴリズム

5.1 節で示した検索方法を実現するアルゴリズムを図 8 および図 9 に示す．ただし，<>で囲った関数の引数は出力引数を表す．

```
Dim：配列の次元数
KeyBegin[]   ：検索範囲の始点のカラム値集合
KeyEnd[]     ：検索範囲の終点のカラム値集合
NodeBegin[]  ：検索の始点となるノードへのポインタ集合
NodeEnd[]    ：検索の終点となるノードへのポインタ集合
IndexBegin[]：検索の始点となるカラム値のノード内の
               位置集合
IndexEnd[]   ：検索の終点となるカラム値のノード内の
               位置集合
Node[]       ：探索の対象となるチャンクに対応する
               ノードへのポインタ集合
Result[]     ：検索結果を受け取る配列

function Search(KeyBegin[], KeyEnd[], <Result[]>){
  GetNode(KeyBegin, KeyEnd, <NodeBegin>, <NodeEnd>);
  GetIndex(KeyBegin, KeyEnd, <IndexBegin>, <IndexEnd>);

  SpecifyChunk(0);
}

function GetNode
(KeyBegin[], KeyEnd[], <NodeBegin[]>, <NodeEnd[]>) {
  for(i = 0; i < Dim; i++) {
    if(i 次元の検索範囲が全範囲ではない) {
      NodeBegin[i] = KeyBegin[i] の含まれるノード;
      NodeEnd[i] = KeyEnd[i] の含まれるノード;
    } else {
      NodeBegin[i] = i 次元の B+木の始点のノード;
      NodeEnd[i] = i 次元の B+木の終点のノード;
    }
  }
}

function GetIndex
(KeyBegin[], KeyEnd[], <IndexBegin[]>, <IndexEnd[]>) {
  for(i = 0; i < Dim; i++) {
    if(i 次元の検索範囲が全範囲ではない) {
      IndexBegin[i] = KeyBegin[i] のノード内位置;
      IndexEnd[i] = KeyEnd[i] のノード内位置;
    } else {
      IndexBegin[i] = 0;
      IndexEnd[i] = NodeEnd[i] 内のカラム値個数-1;
    }
  }
}
```

図 8　検索条件から検索範囲を求めるアルゴリズム

受け取った検索条件から各次元ごとの検索範囲の始点・終点を求めるアルゴリズムを図 8 に示す．全ての次元に対して，検索の始点・終点のカラム値をキーとして B^+ 木を探索し，それぞれのキーの存在するノードへのポインタとノード内での添字を各配列に格納する (GetNode,GetIndex 両関数)．検索条件が指定されていない場合は B^+ 木のシーケンスセットの始点と終点のノードを保存し，IndexBegin[] には 0，IndexEnd[] には終点のノードに含まれるカラム値の個数-1 を格納する．

検索条件から検索範囲を求めたならば，次に検索の対象となるチャンクとチャンク中のオフセットを順次求めていく．図 8 中の SpecifyChunk 関数の呼び出しがこれにあたる．アルゴリズムを図 9 に示す．

```
ChunkID  ：チャンクの ID
Offset[]：ChunkID に対して有効要素であるかどうか
          調べるオフセット値の集合
Index[]  ：オフセット値を調べる添字の集合
IndexBegin_Chunk[]
  ：チャンク内での検索範囲の始点となる添字の集合
IndexEnd_Chunk[]
  ：チャンク内での検索範囲の終点となる添字の集合

function SpecifyChunk(D) {
  if(D == Dim) {
    foreach(node in NodeBegin[D]..NodeEnd[D]) {
      Node[D] = node;
      ChunkID = GetChunkID(Node);
      GetIndex_Chunk(Node, NodeBegin, NodeEnd,
                IndexBegin, IndexEnd,
                <IndexBegin_Chunk>, <IndexEnd_Chunk>);
      SearchChunk(0);
      GetValue(ChunkID, Offset, <Result>);
    }
  } else {
    foreach(node in NodeBegin[D]..NodeEnd[D]) {
      Node[D] = node;
      SpecifyChunk(D + 1);
  }
}

function SearchChunk(D) {
  if(D == Dim) {
    foreach
    (index in IndexBegin_Chunk[D]..IndexEnd_Chunk[D]) {
      Index[D] = index;
      Index[] からオフセットを求め，Offset[] に追加;
    }
  } else {
    foreach
    (index in IndexBegin_Chunk[D]..IndexEnd_Chunk[D]) {
      Index[D] = index;
      SearchChunk(D + 1);
    }
  }
}
```

図 9　検索対象チャンクとオフセットを求めるアルゴリズム

まず再帰を用いて 1 次元から順に検索の対象となるノードを求めていき，すべてのノードが揃った時点でこのノード群によって表されるチャンクを求める．チャンクにはそれぞれ ChunkID と呼ばれる番号が付加されており，この ID でチャンクと対応する二次記憶上のチャンクサブ配列を管理している．

次に検索範囲の始点・終点のノードと現在対象としているノードとの位置関係を用いて，このチャンク内における検索範囲を求める (GetIndex_Chunk 関数)．

対象とするチャンク内での検索範囲が求まったら，再帰を用いてチャンク内の検索範囲に含まれる要素のオフセット値をアドレス計算により順次求めていく．そして，求めたオフセット集合とチャンクサブ配列中のオフセット集合を照らし合わせる．両方の集合に共通するオフセット値が存在した時は，そのオフ

セット値に結びつけられた値を有効要素として Result[] に格納していく (GetValue 関数).

以上の処理を SpecifyChunk 関数の再帰処理が終了するまで行い，得られた Result[] を検索結果として返して検索処理は終了する.

6. 検　　証

これまで述べてきたシステムの性能を確認するため，検証を行った．検証は SunBlade1000 ワークステーション (CPU UltraSPARC-III 150MHz, 主記憶 512M バイト）上でプロトタイプシステムを作成して行った.

6.1 バッファリング機能

4. 節で提案したバッファリング機能の性能を検証するため，5 カラム，300,000 レコードのテーブルデータを用意した．5 カラムそれぞれに 100 種類のカラム値を持っている.

(1)　全レコードのカラム値を入力した後にデータ値を入力する.

(2)　100000 レコードごとにカラム値入力とデータ値入力を行う.

(3)　10000 レコードごとにカラム値入力とデータ値入力を行う.

(4)　1000 レコードごとにカラム値入力とデータ値入力を行う.

(5)　100 レコードごとにカラム値入力とデータ値入力を行う.

(6)　1 レコードごとにカラム値入力とデータ値入力を行う.

の 6 つの方法で，全レコードを入力するのに要した時間をカラム値入力とデータ値入力に分けて計測した．(2) から (5) が本システムにおける定期的ダンプに値し，(6) が実時間性を重視した入力となる.

結果を表 1 に示す.

表 1　データ配列構築の所要時間

方法	カラム値 入力時間 (sec)	データ値 入力時間 (sec)	合計 所要時間 (sec)
(1)	4.60	16.81	21.41
(2)	4.58	17.20	21.78
(3)	4.26	28.14	32.40
(4)	4.11	38.32	42.43
(5)	4.41	39.92	44.33
(6)	9.61	64.51	74.12

このように，レコード数の分割が多ければ多いほど，拡張可能配列の構築と入力に時間を要する結果となった．しかしカラム値の入力に要する時間は (6) を除いてほぼ同じであり，データ値の入力がデータ配列の構築時間に大きな影響を及ぼしていることがわかる.

(6) のカラム値入力に要する時間が長いのは，B^+ 木のノード分割によって二次記憶上のデータの再配置が頻繁に起こるためである．(6) の場合にはノード分割を繰り返すことにより，同じカラム値のデータが複数回移動することがあるのに対して，(2) から (5) の場合は，入力データのバッファリングにより，レコード数の分割毎に 1 度で済ますことができる．(1) の場合は先にカラム値を全て入力するため，ノードが分割してもデータの再配分と移動は発生しない.

6.2 検　　索

6.1 節で構築したデータ配列に対し，4 次元から 1 次元のスライス検索を行い，検索に要する時間を計測した．結果を表 2 に示す.

表 2　n 次元スライス検索の所要時間

スライス次元	検索所要時間 (sec)
4 次元	264.43
3 次元	2.22
2 次元	0.02
1 次元	≒ 0

このように，スライス次元が 1 増えるごとに所要時間がほぼ 100 倍になった．これは，各次元のカラム値の種類が 100 件であることから，スライス次元を 1 次元増やすことで検索の対象となる要素の数が 100 倍になるためである.

また，3 次元スライスにおいて条件を与える 2 つの次元のうち，片方に検索範囲の幅を与え，範囲検索を行った結果を表 3 に示す.

表 3　検索範囲の幅に対する範囲検索の所要時間

検索範囲の幅	検索所要時間 (sec)
1	2.22
2	4.58
5	11.93
10	25.10
20	47.63
50	128.85
100	264.43

このように，検索範囲の幅と検索所要時間がほぼ比例する結果が得られている.

7. む　す　び

拡張可能な多次元配列に基づいて MOLAP 用の基盤データ構造を提案した．これにより，基幹系システムの関係テーブルからのインクリメンタルなダンプおよびそれに伴うインクリメンタルな集約演算が効率よく実行可能である.

文　　献

[1] Y.Zhao, P.M.Deshpande, J.F.Naughton, “An Array-Based Algolithm for Simultaneous Multidimensional Aggregates”, Proc. of SIGMOD’97, pp.159-169, 1997.

[2] H.G.Kang and C.W.Chung, “Exploiting Versions for On-line Data Warehouse Maintenance in MOLAP Servers”, Proc. of the 28th VLDB Conference, pp.742-753, 2002.

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

[4] A.L.Rosenberg and L.J.Stockmeyer, “Hashing Schemes for Extendible Arrays”, JACM, vol.24, 199-221, pp.1977.

[5] E.J.Otoo and T.H.Merrett, “A Storage Scheme for Extendible Arrays”, Computing, vol.31, pp.1-9, 1983.

[6] 都司達夫，水野剛，宝珍輝尚，樋口健, “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.