2005－DBS－135　(13)
2005／1／21

経歴・オフセット法による関係テーブルの実装方式

東　直樹[†]　　黒田　雅之[†]　　Azharul Hasan[†]
都司　達夫[††]　　樋口　健[††]

本論文では，新たな関係テーブルの実装方式を提案し，それに基づき構築したプロトタイプシステムを評価する．この方式はMOLAPシステムにおけるように多次元配列として関係テーブルを管理する．多次元配列の使用により，配列要素の添字の組から，高速に要素にランダムアクセスするためのアドレス関数が使用できる．ところが，一般に，この多次元配列は(1) 疎配列となる，(2)アドレス関数が構成できるためには配列の各次元サイズが固定であることが必要である．次元に対応するカラムに新たな値が追加されたときには，その次元のサイズを拡張する必要があり，これは多次元配列全体を再配置する必要があることを意味する．ここでは多次元配列について，(1)(2)の問題点を解決して，記憶コスト，処理コスト共に効率よく拡張可能性を実現している．評価実験として既存のDBMSであるPostgreSQLとの比較評価を行う．

An Implementation Scheme for Relational Tables Using History-Offset Method

Naoki AZUMA[†]　Masayuki KURODA[†]　Azharul HASAN[†]
Tatsuo TSUJI[††]　Ken HIGUCHI[††]

In this paper, a new implementation scheme for relational tables is proposed, and a prototype system based on the scheme is evaluated. The scheme implements a relational table by employing a multidimensional array like in MOLAP systems. By using multidimensional arrays, fast random addressing functions for element access can be invoked by knowing a tuple of subscripts of an array element. However these kinds of multidimensional arrays suffer from the following two problems, (1) in general, they are sparse,(2) to be able to construct fast addressing functions, their sizes are fixed in every dimension; when a new column value is added, array size extension along the corresponding dimension is necessary and this implies total reorganization of the array. In our scheme, the above problems are resolved and extendibility is acquired with low space and time costs. Our scheme is evaluated compared with PostgreSQL.

1．はじめに

本論文では，新たな関係テーブルの実装方式を提案する．この方式はMOLAPシステムにおけるように多次元配列として関係テーブルを実装し，管理する[1][2]．多次元配列の使用により，配列要素の添字の組から，高速に要素にランダムアクセスするためのアドレス関数が使用できる．多次元配列の二次記憶上の実装方式について多くの研究が行われている[3][4][5]．ところが，一般に，この多次元配列は(1) 疎配列となる，(2)アドレス関数が構成できるためには配列の各次元サイズが固定であることが必要である．次元に対応するカラムに新たな値が追加されたときには，その次元のサイズを拡張する必要があり，これは多次元配列全体を再配置する必要があることを意味する．この(2)の問題点を拡張可能配列をベースとして解決する．拡張可能配列[6]~[10]では，拡張部分のみが部分配列として動的に割付けられ，拡張前の配列要素のデータは再配置する必要はない．

Otoo等によるインデックス配列による拡張可能配列の実現モデル[8]はわずかな記憶域を付加するのみで，高速に配列要素を参照することができ，有力な技法であると考えられる．しかし，同時に(1)の問題点を解決するには，不十分である．すなわち，アドレス関数を使用して配列要素にランダムアクセスできるためには，関係テーブルに存在しているレコードに対応する配列要素以外の要素についても記憶域を確保する必要がある．したがって，適切なデータ圧縮を行う必要があるが，この時，配列の高速ランダムアクセス機能を阻害してはならない．固定サイズの多次元配列を扱うMOLAPの視点からは chunk-offset 圧縮[3]と呼ぶ方法を用いてchunk 単位でデータ圧縮を行う方法が多く採用されているが，chunk の中ではバイナリサーチを行うなど検索速度に問題がある．本論文では，システム内部で関係テーブルを取り扱うための記憶効率のよいレコードのエンコーディングを提案する．

また，レコードとして存在している配列要素についてのみ，記憶域を確保することにより，効率よい圧縮を行いながら検索性能を従来の関係テーブルの実装より大

† 福井大学工学研究科
Graduate School of Engineering., Fukui University
†† 福井大学工学部
Faculty of Engineering., Fukui University

幅に向上させることが可能である．しかし，次元数が大きくなり，部分配列のオフセット空間がオーバーフローすることがある．この問題に対して，検索効率を落とさずにテーブルを垂直に分割することにより対処する．これにより，次元数とレコード数が一般に非常に大きい大規模なテーブルの効率よい取り扱いが要求されるMOLAP をはじめ種々のアプリケーションにおいて，本実装方式が実用的に使用できる．

2．拡張可能配列を用いた関係テーブルの表現方式

以下では，多次元配列を用いた関係テーブルの表現方式とその問題点，またその問題の対策として拡張可能配列[2][3][4]を用いた実現モデルを説明する．

2.1 多次元配列を用いた関係テーブルの表現方式

多次元配列を用いた表現では，関係テーブルの各カラムを配列の各次元に置き換え，カラムにおける各カラム値を対応次元の配列添字と対応付けることで，関係テーブルに挿入されるレコードを配列の一要素として扱うことが出来る．この方式では，同一カラム内において同じカラム値を持つレコードが複数あろうとも，そのカラム値そのものは只一つを保持するだけでよいため，カラム値を保持するコストを削減することが出来る．また，検索や追加，削除などのレコードに対するアクセスは，その多次元配列のアドレス関数を用いることで高速に行うことが出来る．しかし，このような固定配列を用いた関係テーブルの表現において新たなカラム値を含むレコードの追加を行う場合，拡張が必要になった次元のサイズを一つ増加させた多次元配列の領域を新たに確保し，そこに元の領域の情報を適宜コピーするといった高コストな処理が必要になる．また，多次元配列が大きなものになるほどこのコストは増大する．

2.2 拡張可能配列を用いた実現モデル

拡張可能配列とは，配列の拡張が必要になった時に拡張差分の領域のみを確保し，現在の領域はそのまま使用することを可能としたデータ構造である．その例を図 1 に示す．拡張可能配列は，拡張のたびにその拡張差分の領域を部分配列として確保し，その拡張順を表す拡張経歴値と，部分配列の先頭アドレスを経歴値テーブルと呼ばれる補助テーブルに記録する．そのため，配列要素のアドレス配置が従来の配列とは異なっており，従来とは異なるアドレス計算が必要となる．配列要素のアドレスを得るためにはまず，指定された各次元の添字に属する部分配列のうち，その経歴値が最大であるものを選択する．これは配列要素が，常にその各次元の添字に属する部分配列のうち，最大の経歴値を持つ部分配列内に存在するという特性を利用している．次に，指定要素の部分配列内オフセットを求める．拡張可能配列の次元数を n とすると，その部分配列は n-1 次元である．ここで，部分配列の各次元サイズが $(m_1, m_2, \cdots, m_{n-1})$ であるとすると，部分配列内における指定要素 $(x_1, x_2, \cdots, x_{n-1})$ のオフセットは，次式

$$m_1 m_2 \cdots m_{n-2} x_{n-1} + m_1 m_2 \cdots m_{n-3} x_{n-2} + \cdots + m_1 m_2 x_3 + m_1 x_2 + x_1$$

で求めることが出来る．さらに，選択した部分配列の先頭アドレスに, 求めた部分配列内オフセットを加えることにより，指定要素のアドレスを得ることが出来る．また, 要素へのアクセスのたびに上式を計算することは高いコストを伴うので，上式のうち

$$(m_1 m_2 \cdots m_{n-2}, m_1 m_2 \cdots m_{n-3}, \cdots, m_1 m_2, m_1)$$

の n-2 個の静的な値を係数ベクトルと呼び, 経歴値テーブルに記録しておくことによりオフセット計算のコスト低下を図る．例として，図 1 において(3, 3)の要素に対するアクセスを説明する．まず，各次元の指定添字のうち, 最大の経歴値を持つ 1 次元目の添字 3 を選択し，その経歴値 6 に対応する部分配列を選び出す．次に，部分配列が属している次元以外の添字から, 部分配列内オフセットを計算する．そして，選択された部分配列の先頭アドレス 70 に，計算により求められたオフセット 3 を足し，アドレス 73 を得ることが出来る．

この拡張可能配列を用いて関係テーブルを表現することにより，レコードの追加による配列の拡張を，低コストで処理することが出来る．

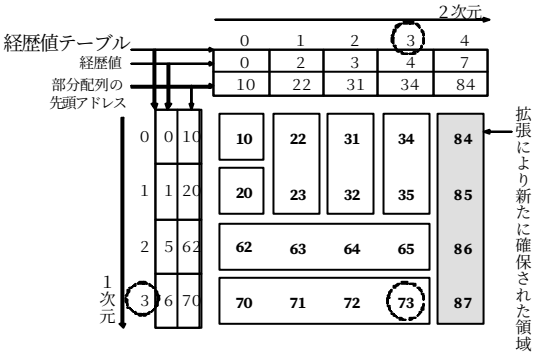

図1 ：拡張可能配列

3．提案する実現モデル

本論文で提案する経歴・オフセット法を用いた関係テーブルの表現方式の実現モデルを説明し, その概要を図 2 に示す．

3.1 経歴・オフセット法

前節において説明した，多次元固定配列および多次元拡張可能配列を用いた関係テーブルの表現では，レコー

ドの存在を表現するための多次元配列の領域全体を確保する必要がある．この領域は，表現する関係テーブルが大規模なものになるほど巨大なものとなり，また，一般に関係テーブルのレコードを多次元配列の要素として表現する場合，疎配列となるため記憶領域を浪費する．そこで本方式では，配列実体は持たずに，関係テーブルに存在しているレコードについてのみ，その位置情報を保持する経歴・オフセット法という方式を用いる．従来の固定配列では，配列の一要素の位置を表現するために，その配列の次元数だけの添字の組が必要であった．しかし，拡張可能配列においてはその要素の位置情報を，配列の次元数によらず，所属する部分配列の経歴値と部分配列内オフセットの2つの値のみで表現することが出来る．そこで配列実体を確保せず，関係テーブルのレコードを表す配列内の有効要素についてのみ，その位置情報を表すこの2つの値の組をRDT(Real Data Tree)とよぶB^+木にキーとして挿入する．これにより，レコードの存在情報を記録する領域を最小限に抑えることが出来る．以後，実体を持たないこの拡張可能配列を論理拡張可能配列と呼び，RDT 内に挿入される経歴値とオフセットの組を，（経歴値，オフセット）対で表す．この対は図 2 に示すように，経歴値が上位バイトにオフセットは下位バイトに配置される．論理拡張可能配列の部分配列はその経歴値で指定されるので，部分配列の各要素はRDT のシーケンスセット上で連続に，さらにオフセットについて昇順に配置される．

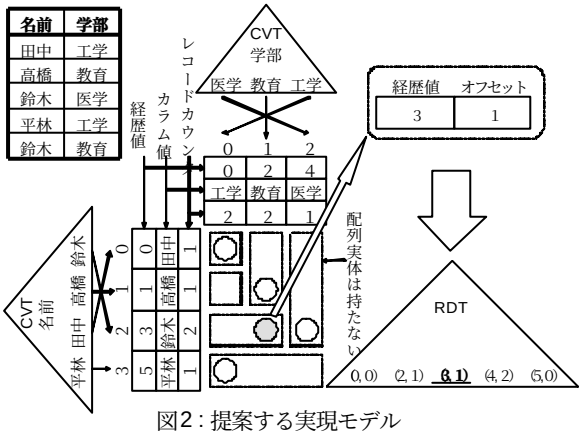

図2：提案する実現モデル

3.2 カラム値から配列添字への変換

関係テーブルを，多次元配列を用いて表現する場合，カラム値から配列添字への変換，および配列添字からカラム値への逆変換が必要となる．我々は，大規模な関係テーブルをより高速に扱い，さらにはカラム値の範囲を指定した範囲検索にも対応することを目的としているため，この変換を高速に行うことが求められる．そこで関係テーブルの各カラムに CVT（key subscript ConVersion Tree）と呼ぶ添字変換用B^+木を配置し，カラム値をキーとして，配列添字をキーに対応するデータとして CVT に挿入する．逆変換については，経歴値テーブルの各スロットに，対応するカラム値そのもの，もしくは CVT 内のカラム値が格納されている領域へのポインタを記録しておく．

3.3 レコードの追加と削除

本方式における関係テーブルに対するレコードの追加は，論理拡張可能配列の一要素に存在情報を格納する操作，つまり，未登録のカラム値があればそれを対応する次元の CVT に登録することおよび，経歴・オフセット法により計算された経歴値とオフセットの組を 1 つのキーとして RDT に挿入することと等価である．レコードの追加処理ではまず，挿入対象のレコードの各カラム値が対応次元の CVT にそれぞれ存在しているかを確認する．もし存在しなければ，論理拡張可能配列をその次元方向に 1 つ拡張し，CVT にそのカラム値を挿入して拡張部分に対応付ける．次に，追加対象のレコードを各次元の添字の集合に変換し，さらに経歴・オフセット法により経歴値とオフセットの組に変換，それを RDT にキーとして挿入する．

また，レコードの削除処理は論理拡張可能配列の一要素の存在情報を削除すること，つまり，RDT から（経歴値，オフセット）対を，また，必要ならばレコードの各カラムに対応する CVT からカラム値を除去することと等価である．まず，削除対象レコードの経歴値とオフセットの組を計算し，RDT からそれを削除する．次に，各 CVT の検索結果であるそのレコードに対応する論理拡張可能配列の各添字について，その添字を持つレコードが他に存在しないならばその次元に対応する CVT からその添字に対応するカラム値を削除する．ただし，存在しないことを確認するには，拡張可能配列について，対象次元をその添字に固定したスライス検索を行う必要があり，この処理をレコードの削除が起こるたびに毎回行っていたのでは大きなコストがかかってしまう．そこで，各次元の経歴値テーブルの各スロットに，そのカラム値を持つ全てのレコード数を記録するカウンタをレコードカウンタとして設け，追加や削除の際に適宜更新する．これにより，その添字に関連付けられたカラム値を持つレコードが存在するかどうかを，このカウンタが 1 以上であるかどうかを見るだけで判断できる．

また，CVT からカラム値が削除される場合，論理拡張可能配列の経歴値テーブルに使用しない領域，すなわち空きスロットができてしまう．新たなカラム値の追加が起こった場合，この空きスロットの再利用を効率的に行うために上述のレコードカウンタを利用した空きスロットリストの管理を行っているが詳細は省略する．

3.4 経歴値・オフセットから配列添字への逆変換

経歴・オフセット法によって RDT 内に挿入されている

経歴値とオフセットの組を，検索などの処理において取り出した場合，元のレコードを復元，つまりカラム値の集合に戻すためには経歴値とオフセットの組を論理拡張可能配列の各次元の添字に変換する処理が必要となる．例として，3 次元で各次元のサイズが(m_1, m_2, m_3)の部分配列内の要素(x_1, x_2, x_3)の部分配列内オフセットは $m_1m_2x_3 + m_1x_2 + x_1$である．ここで，オフセット値をm_1m_2で割ったときの整数商をx_3，その余りをm_1で割った時の整数商をx_2，さらにその余りをx_1とすることにより，部分配列の次元数-1 回の除算で，部分配列内における指定要素の各次元添字を得ることが出来る．これに，その部分配列が属する添字を組み合わせて，論理拡張可能配列内における各次元の添字を得ることが出来る．

3.5 レコードの検索

以下では，提案する実現モデルにおける関係テーブルの検索処理について説明する．

3.5.1 検索の概要

本システムにおける検索は，RDT 内に格納されている経歴値とオフセットの組に対して行われる．検索には，全カラムのカラム値が指定される場合と，いくつかのカラム値が指定されない場合の 2 種類がある．

全カラムが指定された場合の検索は，まず，指定されたカラム値の集合から各次元の CVT を通して論理拡張可能配列の配列添字の組を得る．この時点で CVT 内に指定されたカラム値が1つでも存在しなかった場合，その検索で得られるレコードは存在しないことになるので，検索処理を終了しユーザにレコードが存在しないという情報を返す．次に，経歴・オフセット法によりその配列添字の組に対応する経歴値とオフセットの組を得る．得た経歴値とオフセットの組で RDT を検索し，RDT 内に存在すればそのレコードはシステムが扱う関係テーブル内に存在するということであり，その場合，ユーザにレコードが存在する旨を伝える．レコードの各カラム値は全て，ユーザによって指定されているため，ここでは逆変換操作などは必要ない．また，もし RDT 内に存在しなければ，テーブル内にそのレコードは存在しないということであり，存在しないという情報をユーザに返す．

また，いくつかのカラム値が指定されない場合の検索では，まず，カラム値が指定された全ての次元に対し CVT を用いて，カラム値から論理拡張可能配列の添字に変換する．CVT 内に指定されたカラム値が 1 つでも存在しなかった場合，上記の検索と同様に処理を終了する．次に，指定されたすべてのカラムに対応する次元それぞれについて，その添字が保持する論理部分配列の拡張経歴値のうち，最大の拡張経歴値を得る．そして，その経歴値を持つ部分配列と，その経歴値より大きく，且つカラム値が指定されていない次元に属する論理部分配列全てにおいて各指定添字を持つレコードが存在するかどうかを検索する．これは拡張可能配列において，指定添字を持つ要素は，その添字に対応する部分配列の拡張経歴値を持つ部分配列内および，それより大きな経歴値を持ち，且つ指定次元に属していない部分配列内にのみ存在するという特性を利用している．

3.5.2 部分配列内の検索

本方式において大規模な関係テーブルを表現する際，RDT のサイズが非常に大きなものになると考えられる．そこで本方式では，RDT を主記憶上ではなく二次記憶上に配置することを想定する．二次記憶上に置かれた B^+ 木の検索において，ルートノードからリーフノードに至る処理には，B^+木の高さ分のディスクアクセスが必要になるため，高コストな処理であると考えられる．一方，RDT 内では経歴値順に，さらにオフセット順にシーケンスセット部に部分配列内のレコードが連続的に格納されている．したがって，シーケンスセット部を辿る操作，すなわち現在選択しているレコードの次に格納されているレコードを取り出す処理は，シーケンスセット部のノードにおいて先頭の要素であれば 1 回のディスクアクセス，それ以外ならばディスクアクセスを必要としないので，低コストである．

本システムにおいて高速な検索を実現するには，ルートノードから辿る操作とシーケンスセット部を辿る操作を組み合わせた効率のよいアルゴリズムが必要であり，ここでは，二つの方式を考案した．論理拡張可能配列の有効要素の疎密に応じて使い分ける．

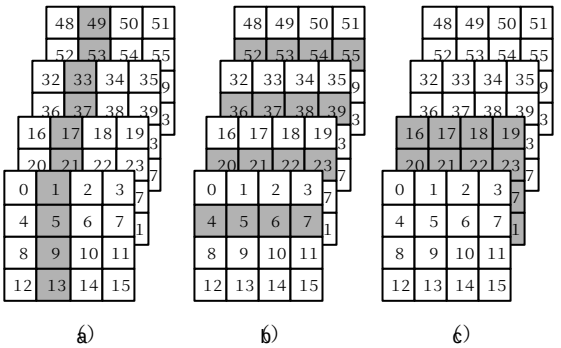

図3 ：検索範囲

3.5.3 範囲指定サーチ

有効要素の密度が高い場合に有効な方式である．この方式は，部分配列内において，検索対象のレコードが RDT 内で連続に格納されている範囲が存在するということに着目し，連続している範囲に対しては高速に辿ることの出来るシーケンスセット部を辿り，それ以外では，ル

ートノードからの探索を行う方式である．例として図 3 に，3 次元で各次元のサイズが 4 である部分配列における 1 つの次元が指定された際の検索範囲を示す．この方式では，図 3(b)のような場合には，まずオフセット 4 に対してルートノードからの探索を行い，そこからオフセット 7 を超えるレコードが現れるまでシーケンスセット部を辿り，7 を超えない間はそのレコードを検索対象として得る．次の範囲も同様に処理し，部分配列全体の検索を行う．ここで図 3(b)の場合は，ルートノードからの探索は 4 回，図 3(c)の場合は，ルートノードからの探索は 1 回，図 3(a)の場合は，ルートノードからの探索は 16 回である．このように，指定された次元によって検索コストに格差が生まれてしまい，これを検索の次元依存性と呼ぶ．

3.5.4 シーケンシャルサーチ

もう 1 つの方式として，シーケンシャルサーチを考案した．この方式は，関係テーブルを表現している論理拡張可能配列が疎である場合に有効であり，その部分配列内に存在する全てのレコードに対して，検索条件に合致するかどうかを，経歴・オフセット法の逆変換を用いて調べる方式である．この方式による検索は，部分配列内の全てのレコードに対して経歴・オフセット法の逆変換処理をして添字を計算する必要があるが，部分配列が疎配列であるために少ないディスクアクセス回数で，しかも高速に辿ることの出来るシーケンスセット部を辿ることで，上記の方式のような次元依存性もなく，高速な検索が可能になると考えられる．

3.6 HORT

本論文で提案する，経歴・オフセット法を用いて関係テーブルを表現するためのデータ構造を HORT(History Offset implementation scheme for Relational Table)と呼ぶこととする．

4．経歴・オフセット空間のオーバーフローとその対策

HORT においてレコードを表現するために，RDT に格納されている経歴値とオフセットの組が表現できる空間は，実装上 2 つの値を変数に格納するため，その変数のとり得る値の上限によって部分配列の論理サイズ（経歴・オフセット空間）には制限がある．これでは，表現する関係テーブルのカラムやそのカラム値の種類が多くなった時，経歴値，もしくはオフセットのどちらかがオーバーフローしてしまう可能性がある．以下では，この経歴・オフセット空間のオーバーフローの対策を説明する．

4.1 一意キーの別管理

経歴・オフセット空間のオーバーフローを加速させる原因の 1 つとして，一意キーの存在がある．一意キーとは，例えば「学籍番号」「社員番号」のように，カラム値の重複が起こり得ないカラムのことである．このようなカラムが存在する場合，レコードが挿入されるたびに必ず一意キーに対応する次元方向に論理拡張可能配列の拡張が起こるため経歴値が増え，さらには論理部分配列のサイズが大きくなるため，経歴・オフセット空間のオーバーフローを早めてしまう．そこで，一意キーを他のカラムとは別に管理し，一意キー以外のカラムのみによって論理拡張可能配列を構成することにより，論理拡張可能配列の次元が減り，論理部分配列のサイズが小さくなるため，経歴・オフセット空間のオーバーフローを遅らせることが出来る．一意キーの別管理を行った例を図 4 に示す．

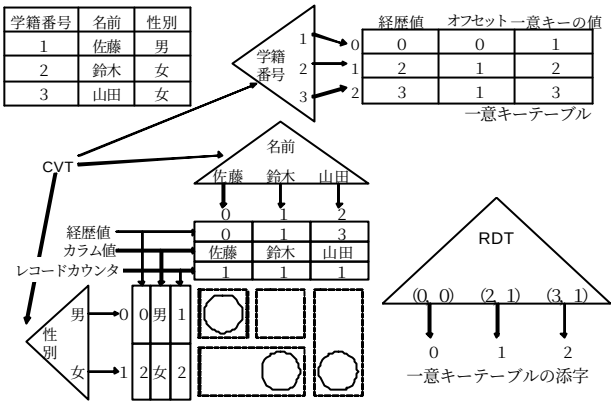

図4 ：一意キーの別管理

まず，一意キーではないカラムによって通常の HORT を構成し，論理拡張可能配列内での対象レコードの位置情報である経歴値とオフセットの組を得る．次に，一意キーについては従来の HORT とは別に，一意キーテーブルというテーブルを構成し，その 1 スロットに他のカラム値から得られた経歴値とオフセットの組と，一意キーのカラム値そのものを格納する．さらに RDT に，経歴値とオフセットの組をキーとして，格納した一意キーテーブルのスロットの添字をキーに対応するデータとして挿入する．また，一意キーに対応する CVT を設け，キーとしてカラム値を，キーに対応するデータとして格納した一意キーテーブルのスロットの添字を挿入する．

このような構成にすることにより，一意キーのカラム値が指定された場合には，一意キーテーブルの該当スロットに格納されている経歴値とオフセットの組から，経歴・オフセット法の逆変換により一意キー以外のカラムのカラム値を得ることができ，逆に一意キー以外が指定された場合にも，従来の検索によって経歴値とオフセットの組に対応するデータとして一意キーテーブルの添字を得ることが出来るので，一意キーテーブルから，対応する一意キーのカラム値を得ることが出来る．

また，一意キーに対する経歴値テーブルが不要になる．経歴値テーブルは，拡張経歴値やレコード数のカウンタ，

係数ベクトルなどを持っていたため，一意キーテーブルと同等もしくはそれ以上の大きさの領域を要するテーブルであった．そのため，上記の別管理方式を用いることで，経歴･オフセット空間のオーバーフローを遅らせるとともに，空間的コストも抑えることが出来る．また，一意キーが複数存在する場合でも，一意キーテーブルのスロットの最後に複数の一意キーのカラム値を格納し，それぞれのCVTによって一意キーのカラム値と一意キーテーブルのスロットの添字を関連付けることによって対応することができる．

4.2 関係テーブルの分割管理

前節で述べたように，一意キーを別に管理することで論理拡張可能配列の次元数やサイズを減少させることができるが，これはあくまでも経歴・オフセット空間のオーバーフローを遅らせるための措置であるため，新たなカラム値を持つレコードの追加によって経歴・オフセット空間がオーバーフローしてしまうことを回避できない．そこで，経歴値やオフセットがオーバーフローを起こした時の処置として，表現対象の関係テーブルを分割管理することを提案する．

表現対象の関係テーブルに新たなレコードが挿入された際に論理拡張可能配列の拡張が行われ，経歴・オフセット空間のオーバーフローが起きた場合，現在扱っている関係テーブルを 2 つのカラム集合の組に分割する(図5)．そして，分割したそれぞれの関係テーブルについて従来通り論理拡張可能配列を構成し，経歴値とオフセットの組をそれぞれの RDT に格納する．この際，関係テーブルをただ単に 2 つに分割し，それぞれを別に管理するだけでは，2つに分割された関係テーブル間の関係が失われてしまう．そこで，2つの分割テーブルの関係を維持するために，前節で提案した一意キーテーブルを利用する．一意キーテーブルには，1個以上の一意キーの値と，経歴値とオフセットの組を 1 つ格納していたが，この方式では，元の関係テーブルが持つ一意キーの値と，分割された論理拡張可能配列がそれぞれに持つ RDT に格納されている経歴値とオフセットの組を格納する．また，一意キーの別管理と同様に，RDTのキーに対応するデータとして一意キーのスロットの添字を格納する．これにより，一意キーの値が1つわかれば，一意キーテーブルを参照することにより，各分割テーブルに対応する論理拡張可能配列の経歴値とオフセットの組と，他の一意キーの値が得られる．また，1つの分割テーブル内のRDTに格納されている経歴値とオフセットの組がわかれば，他の分割テーブルにおける経歴値とオフセットの組と，すべての一意キーの値を知ることが出来る．

元の関係テーブルに一意キーが存在せず，一意キーテーブルが存在しない場合にも，分割時に一意キーテーブルを作成し，分割された各論理拡張可能配列のRDT に格納されている経歴値とオフセットの組を格納する．

この方式では，分割する際に，RDT，CVT 及び一意キーテーブル，これら全てを再構成するため，分割時に大きなコストがかかってしまう．さらに，RDTは，分割前のものと同じサイズのものが2つ出来るため,空間的コストもかかることになる．しかし，論理拡張可能配列の次元数が約半分になるため，経歴･オフセット空間のオーバーフローを格段に遅らせることが出来る．さらに，各分割テーブルにおいてもオーバーフロー時に同様に分割処理を行うことが出来るため,ほぼ無限にレコードを挿入することが出来る．ただし，分割時にただ2つに分割しただけでは，2つの分割テーブルにおける次の分割タイミングに大きな差が出来てしまう可能性があり，この差を出来るだけ等しくなるように分割することにより,時間的コストの大きな分割処理のタイミングを遅らせることが出来る.この分割タイミングを出来るだけ等しくするためには,論理拡張可能配列の各次元のサイズを調べ，2 つの分割テーブルにその積ができるだけ平等になるように振り分けることが必要である．

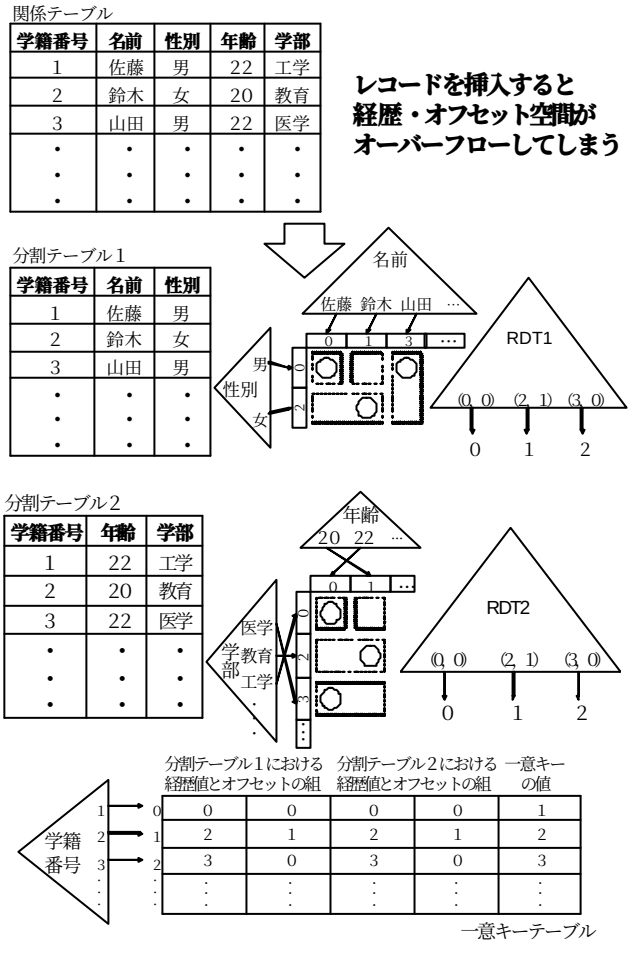

図5 ：関係テーブルの分割管理

5．評価実験

既存の DBMS である PostgreSQL [11](version7.4.3) と HORT との比較実験を行う事により，本方式の有用性について検証を行う．

5.1 実験環境

実験に使用した計算機(64bitsUNIX 計算サーバ)の仕様を表 1 に示す．また，HORT において RDT に格納する値について，経歴値には非負整数型 unsigned int(32bits)，オフセットには非負整数型 unsigned long(64bits)，キーに対するデータ値には，整数型 long(64bits)をそれぞれ割り当てる．また，HORT の検索方式としては，3.5.4 において説明したシーケンシャルサーチを用いる．

表 1：64bitsUNIX 計算サーバ

計算機	SUN Enterprise 5500
CPU	UltraSPARC-II × 14
メモリ容量	5GB
OS	Solaris8

5.2 実験 1

HORT と PostgreSQL における検索速度，及び二次記憶上におけるサイズを比較する．

5.2.1 実験内容

5 つのカラムからなる関係テーブル(一意キーなし)を HORT，PostgreSQL によって表現し，その検索速度及び二次記憶サイズの比較を行う．レコード数は 100 万件とし，カラム値の型は全カラム，整数型(int:32bits)及び長さ 20 の文字列型(160bits)の 2 通りにおいてそれぞれ実験を行う．さらに，全てのカラムにおける重複度(全レコード数/カラム値の種類)についても全て同じとし，重複度 50，100，200 の 3 通りにおいてそれぞれ行う．

5.2.2 実験結果

実験結果を図 6，図 7 に示す．図 6 より，検索において HORT の方が整数型では約 2～3 倍，長さ 20 の文字列型では約 3～4 倍の速度で処理出来ることがわかる．これは HORT では全レコードの内，検索する範囲を限定することが出来，さらにレコードカウンタを保持しているため，検索処理の終了判定を適正に行う事が出来ることによるものと考えられる．また，HORT において重複度が高くなるにつれ，検索処理時間が減少している．これは同じレコード総数に対して，重複度が高くなるにつれ，HORT が管理する論理拡張可能配列のサイズが小さくなるため，検索範囲が狭まることによるものであると考えられ，HORT は重複度が高い関係テーブルにおいて有効であると言える．また，図 7 より，HORT の二次記憶サイズは PostgreSQL におけるそれの，整数型においては約 2 分の 1，長さ 20 の文字列型においては約 4 分の 1 であることがわかる．これらの結果より，HORT は検索速度，二次記憶サイズ共に PostgreSQL に比べ優れている．

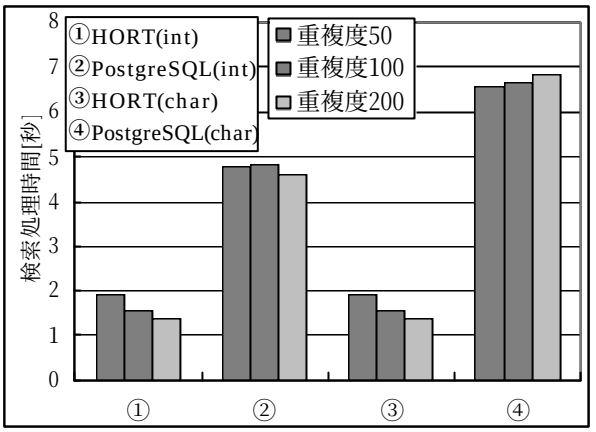

図 6：検索処理時間の比較

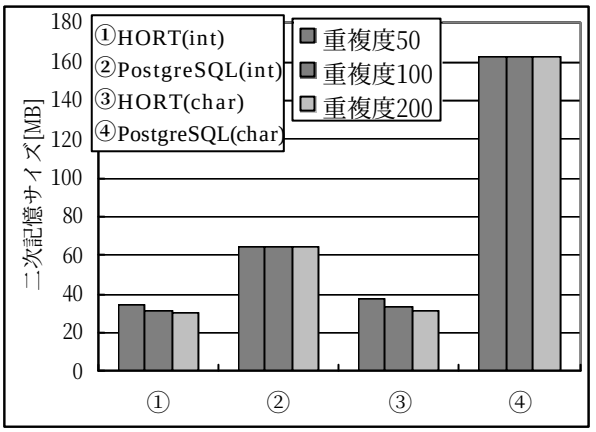

図 7：二次記憶サイズの比較

5.3 実験 2

HORT における分割管理による検索速度及び二次記憶サイズの変化について考察を行うために，経歴・オフセット空間がオーバーフローする状態近くまで拡張を行ったものと，それを分割管理したものについて，検索速度及び二次記憶サイズの比較を行う．また PostgreSQL においても同様の比較を行う．

5.3.1 実験内容

10 個のカラムからなる関係テーブル(一意キーを 1 つ含む)を HORT，PostgreSQL において表現し，それぞれ，及び HORT において関係テーブルを分割管理したものについて，検索速度及び二次記憶サイズの比較を行う．レコード数は 100 万件とし，一意キーの型は整数型(int:32bits)とする．また，一意キーを除く各カラム値の型は全カラム共通とし，整数型(int:32bits)及び，長さ 20 の文字列型(160bits)の 2 通りにおいてそれぞれ実

験を行う．また，一意キー以外のカラムの重複度は 10000 とする．

5.3.2 実験結果

実験結果を図 8，図 9 に示す．図 8 より，関係テーブルの分割管理を行った HORT の方が元の HORT より検索速度が少し速くなっていることがわかる．これは，分割管理することにより HORT 内の，検索対象のカラムを含む論理拡張可能配列のサイズが小さくなることにより，検索を行う範囲が狭まったことによるものと考えられる．また図 9 より，HORT において関係テーブルの分割管理をすることにより，二次記憶サイズが増加することがわかる．これは，一意キーテーブルの 1 スロットのサイズが，経歴値とオフセットの組をもう一組格納するために増加したためである．また，整数型を表現した HORT においては，分割管理をすることにより，PostgreSQL における 2 次記憶サイズを少し超えてしまっている．これは，表現する関係テーブルにおける 1 レコードのサイズが小さいために，HORT において 1 レコードを表現するために必要なサイズを下回っているからである．しかし検索速度においては PostgreSQL を上回っており，この方式は有効であると言える．

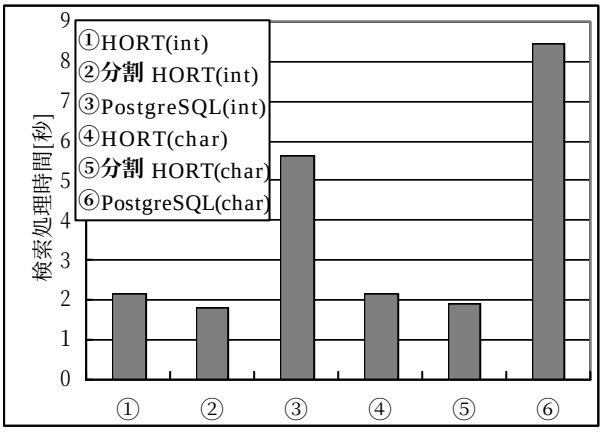

図 8：検索処理時間の比較(2)

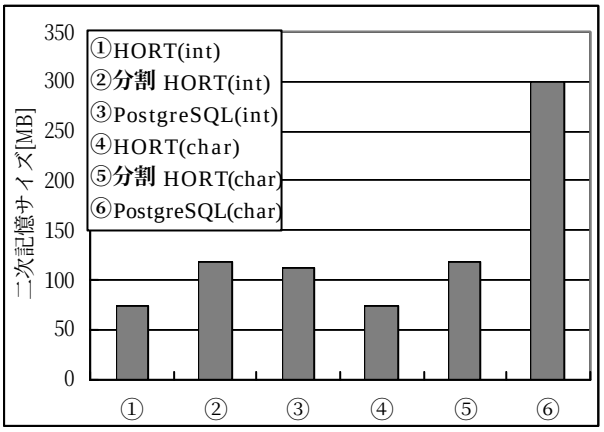

図 9：二次記憶サイズの比較(2)

6．まとめ

関係テーブルを効率よく表現し，高速な検索を可能とするデータ構造の提案を行った．本方式は，関係テーブルのレコードが各カラムのカラム値の集合であることに注目し，また，カラム値の追加による拡張に低コストで対応するために各カラムを各次元に対応付けた多次元拡張可能配列を用いて関係テーブルを表現するものである．さらに，拡張可能配列の概念である経歴・オフセット法を用いて，配列要素の位置情報を圧縮し，拡張可能配列内の有効要素のみ RDT に記録することで，疎配列問題を解消すると共に RDT を使用した高速検索アルゴリズムを提案した．本方式の欠点は関係テーブルのサイズが増大するにつれ，経歴･オフセット空間が飽和し，新たなカラム値を持つレコードの追加が不可能になることであるが，この点に関する解決策を提案した．また，既存の DBMS である PostgreSQL との比較評価を行い，本方式が有効であることを示した．

参考文献

[1] Heum-Geun Kang, Chin-Wan Chung: Exploiting Versions for On-line Data Warehouse Maintenance in MOLAP Servers, Proc. of VLDB 2002, pp.742-753(2002).

[2] T.Tsuji, A.Isshiki, T.Hochin, K.Higuchi: An Implementation Scheme of Multidimensional Arrays for MOLAP, Proc. of the 13th International Workshop on Database and Expert Systems Applications, pp.773-778(2002)

[3] S. Sarawagi, M. Stonebraker: Efficient Organization of Large Multidimensional Arrays, Proc. of ICDE, pp.328-336, 1994.

[4] K. E. Seamons, M. Winslett: Physical Schemas for Large Multidimensional Arrays in Scientific Computing Applications, Proc. of Scientific and StatisticalDatabase Management, pp.218-227(1994)

[5] P.M.Deshpande, K.Ramasamy, AShukla, and J.F.Naughton: Caching multidimensional queries using chunks, Proc. of SIGMOD, pp.259-270(1998).

[6] A.L.Rosenberg : Allocating Storage for Extendible Arrays, JACM, Vol. 21, pp.652-670(1974)

[7] A.L.Rosenberg, L.J.Stockmeyer : Hashing Schemes for Extendible Arrays, JACM, Vol.24, pp.199-221(1977)

[8] E.J.Otoo, T.H.Merrett : A Storage Scheme for Extendible Arrays, Computing, Vol.31, pp.1-9(1983)

[9]A.Novacek: Using Time Stamps for Storing and Addressing Extendible Arrays, Computing, Vol.37, pp.303-313(1986)

[10] 水野 剛, 都司 達夫, 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)

[11] PostgreSQL グローバル開発グループ: PostgreSQL 7.4.3 文書, http://www.postgresql.jp/document/pg743doc/html/