チャンク化拡張可能配列による関係テーブルの実装と評価

相羽 洋平　黒田 雅之　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

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

あらまし　本論文では、チャンク単位で拡張を行う拡張可能配列による関係テーブルの一実装方式を提案、実装し、空間的コストと検索時の時間的コストについて、従来の方式との比較評価を行う。我々は HORT （History Offset implementation scheme for Relational Tables）と呼ぶ時間的・空間的コストの優れた関係テーブルの実装方式を提案している。HORT における問題点のひとつとして、経歴・オフセット空間のオーバフローの問題がある。ここでは、経歴・オフセット空間を有効に使用するために、拡張可能配列の拡張を、配列要素単位で行うのではなく、配列要素の正方部分配列からなるチャンクを単位として行うことによりこの問題を軽減する。 評価実験として、拡張可能配列の拡張が配列要素単位である HORT と、チャンク化 HORT による空間的コストと検索の時間的コストを実測し、比較評価を行う。

キーワード　関係データベース　拡張可能配列　チャンク

An Implementation Scheme of Relational Tables by Extendible Chunk-Array

Yohei AIBA, Masayuki KURODA, Tatsuo TSUJI, and Ken HIGUCHI

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

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

Abstract　In this paper, an implementation scheme for relational tables is proposed. Our scheme employs extendible chunked arrays. We are proposing an implementation scheme for relational tables named HORT(History Offset implementation scheme for Relational Tables) which exhibits good performance in space and time costs compared with conventional implementation. However, the problems of HORT include that its history-offset space will overflow when a large scale relational table is stored. While a subarray of elements is dynamically allocated in the usual HORT, in our scheme proposed, a subarray of chunks is allocated in order to utilize the history-offset space efficiently, Here, a chunk means a hyper-cube shaped subset of a extendible array. Using a constructed prototype system, the space and time performances of the usual HORT and the proposed scheme will be measured and compared.

Key words　Relational Database, Extendible array, Chunk

1.まえがき

近年の高度な情報化社会のなか、企業や組織が運用を通じて蓄積されたデータや顧客情報などのデータを分析し、意思決定や戦略思案の支援にデータベースを活用することが多くなっている。そのため、大量のデータをユーザーの問い合わせに応じて高速に検索を行うデータベースは必要不可欠なものになっている。そこで我々は、関係テーブルが各カラムのカラム値の集合であることに着目し、関係テーブルの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し、高速な検索を可能にする関係テーブルの実装方式である HORT(History Offset implementation scheme for Relational Table)[1][2]を提案した。HORT では、新たなカラム値を持つレコードの追加による関係テーブルの拡張に対して、低コストで対応するために、従来の固定サイズの配列ではなく、拡張可能配列[3]-[7]の概念をベースとして用いる。従来の拡張可能配列の実装に対して、HORT では経歴・オフセット法[1] [2] と呼ぶ手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について B^+-tree に格納することで疎配列問題を解消している。しかし、この方式に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という欠点がある。本論文では、n 次元拡張可能配列をチャンクと呼ばれる各次元等サイズの n 次元部分配列を単位として拡張することを提案し、この経歴・オフセット空間の狭小の問題の解決を減少させる。また HORT では、拡張可能配列内の有効要素つまり、関係テーブルの全てのレコードの情報を 1 つの

B+-tree に格納しているため、挿入レコードが増えるにつれてこの B+-tree の高さが高くなり、それにつれ挿入や検索の時間的コストが大きくなってしまう。そこでチャンク単位で拡張を行う拡張可能配列が、チャンク毎の独立性が高いことに注目して、チャンク毎に B+-tree を分割して持たせることとする。このように B+-tree を分割することで、1 つの B+-tree に格納されるレコードの数が減るため、挿入時間や検索時間のコストを抑えることができる。以上の方式を実装し、既存の DBMS である postgreSQL と配列要素単位で拡張を行う HORT 、チャンク単位で拡張を行う本方式について単一値指定検索における時間的コストを実測し、比較評価を行う。

2. HORT の概要

ここでは、我々が提案している HORT(History Offset implementation scheme for Relational Tables) と呼ばれる拡張可能配列の概念を用いた関係テーブルの実装方式について説明する。

2. 1. 多次元配列を用いた関係テーブルの実装方式

従来の関係テーブルの実装方式においては、レコードは挿入順に逐次 2 次記憶上に配置されるため、次のような問題点がある。まず、レコードはカラム値の集合として 2 次記憶上に配置されるため、年齢や性別といったカラムにおいては同じカラム値が重複して数多く 2 次記憶上に格納される。また、あるカラム値を持つレコードの検索を行うには、全てのレコードを主記憶上に読み込み、カラム値をチェックする必要がある。そこで、これらの問題の対策として、多次元配列を用いて関係テーブルを実装することが考えられる。

図 1 に示すように、関係テーブルの各カラムを多次元配列の各次元に割り当て、カラム値を対応次元の配列添字と対応付けることにより、関係テーブルの各レコードを配列の 1 要素として扱うことができる。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、カラム値そのものはただ 1 つ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できる。また、レコードの検索や挿入、削除におけるレコードへのアクセスは、その多次元配列のアドレス関数を用いることにより、高速に行うことができる。しかし、このような多次元固定配列を用いて実装された関係テーブルに新たなカラム値を含むレコードを挿入するには、そのカラムが割り当てられている次元のサイズを 1 つ増加させた多次元配列の領域を確保し、全配列要素を再配置するといった高コストな処理が必要となる。また、多次元配列が大きなものとなるほどこのコストは増大する。そこで、任意次元方向に低コストで拡張を行うことができる拡張可能配列を用いることとする。

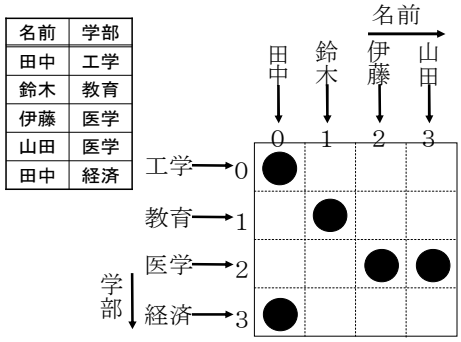

図 1：多次元配列を用いた関係テーブルの実装例

2.2. 拡張可能配列を用いた関係テーブルの実装方式

拡張可能配列とは、配列の拡張が必要となった時に拡張差分の領域のみを確保し、現在確保している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。n 次元拡張可能配は、図 2 に示すように経歴値カウンタと各次元に経歴値テーブル、アドレステーブル、次元数 が 3 以上であれば係数テーブルの 2 または 3 種類の補助テーブルを持つ。配列拡張が行われるたびに現在の経歴値カウンタが 1 つインクリメントされ、その値が経歴値テーブルに順次記録される。ある次元方向への配列拡張は、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確保し拡張可能配列に追加することによって行われる。この n －1 次元の連続記憶領域は通常の固定配列であり、部分配列という。例えば、現在のサイズが[s_1, s_2, s_3, s_4] の拡張可能配列において、次元 2 方向に 1 つ配列拡張を行う場合、サイズ[s_1, s_3, s_4] の 3 次元部分配列が動的に確保され、この部分配列の先頭アドレスをアドレステーブルの該当スロットに記録する。拡張可能配列が 3 次元以上の場合には、部分配列内の要素のアドレスを計算するための 1 次関数の n －2 個の係数からなる係数ベクトルを部分配列毎に係数テーブルに記録する。例えば、上記の部分配列内の要素(i_1, i_3, i_4) のアドレスは 1 次関数 $s_1s_3i_4 + s_1i_3 + i_1$ で求められる。この(s_1s_3, s_1) を係数ベクトルと呼ぶ。例えば図 2 において、配列要素(3,4) のアドレスの計算は次にように行われる。まず、第 1 次元の添え字が 3 である部分配列の経歴値 6 と、第 2 次元の添え字が 4 である部分配列の経歴値 8 を比べ、6 ＜ 8 であるので、要素(3,4)は、経歴値 8 の部分配列に含まれる。また、この部分配列の先頭アドレスは 63 であり、要素(3,4)は部分配列内では要素(3)であるため、求めるアドレスは 66 となる。この拡張可能配列を用いて関係テーブルを実装することにより、新たなカラム値を含む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できる。

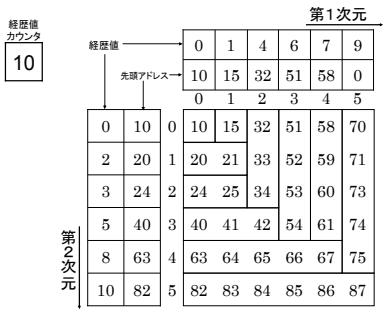

図 2：拡張可能配列の構造

2.3. 経歴･オフセット法

多次元固定配列ならびに多次元拡張可能配列を用いた関係テーブルの実装方式では、レコードの存在の有無を表現するために、配列領域全体を確保する必要がある。この領域は、実装する関係テーブルのカラム数や､カラム値の種類が多くなるほど巨大なものとなる。さらに、一般に関係テーブルを多次元配列を用いて実装すると、疎配列となるため記憶領域を浪費する。そこで、関係テーブルに存在しているレコードについてのみ配列上での位置情報を保持する。一般に、多次元配列内の要素の位置を示すには次元数だけの配列添字を必要とするが、HORT では拡張可能配列の次元数に依らず、要素が含まれる部分配列の経歴値と部分配列内オフセットの２つの値のみを用いて要素の位置を示す経歴・オフセット法を用いる。例えば、図 2 の配列要素(3,4) の位置を経歴・オフセット法を用いて表現すると、要素(3,4) が含まれる部分配列の経歴値は８，要素(3,4)の部分配列内での位置は(3)であるため､経歴値が８，オフセットが３となる。また、経歴値とオフセットの組から配列要素の組を求めるには、経歴値から要素が含まれる部分配列を特定し、オフセットをその部分配列の係数ベクトルで順次除算する。HORT では関係テーブルのレコードを表す配列の有効要素についてのみ、レコードの位置情報を表すこの２つの値の組を、2 次記憶上に配置された RDT(Real Data Tree) と呼ぶ B⁺-tree にキーとして挿入する。これにより、配列実体を確保する必要がなくなり、レコードの存在情報を記録する領域を抑えることができる。以後、実体を持たないこの拡張可能配列のことを論理拡張可能配列と呼ぶ。また、RDT 内では、キーの上位バイトを経歴値､下位バイトをオフセットとすることにより、経歴値、次いでオフセットの順でソートされる。したがって、同じ経歴値を持つキーは RDT のシーケンス・セット上に連続して配置される。

2.4. カラム値から配列添え字への変換

関係テーブルを多次元配列で実装する場合、カラム値から配列添字への変換ならびに配列添字からカラム値への逆変換が必要となる。カラム値から配列添字への変換を高速に行うため、関係テーブルの各カラムに CVT(key-subscript ConVersion Tree)と呼ぶ B⁺-tree を配置し、カラム値をキーとして、対応する配列添字をデータとして格納することにより高速に変換を行う。また、配列添字からカラム値への逆変換は、各次元の補助テーブルとしてカラム値テーブルを設け、対応するカラム値を記録することにより行う。以後、各次元の補助テーブル群のことを､HORT テーブルと呼ぶこととする。図 3 に HORT の構造の例を示す。

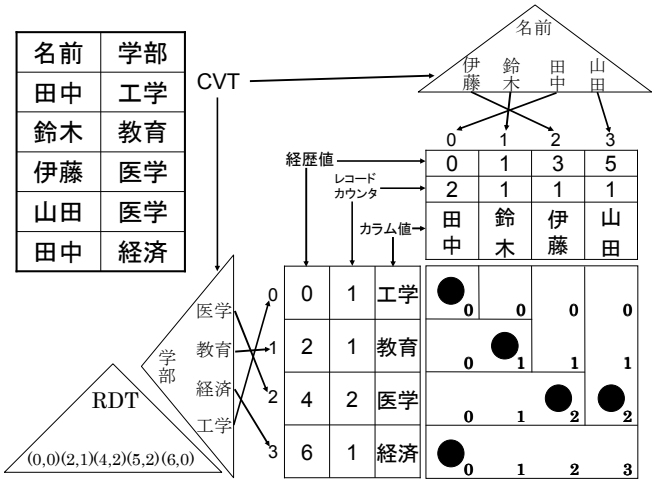

図 3:HORT の構造

3. チャンク単位で拡張を行う HORT の概要

以下では、HORT における問題点の提起と、その問題点を解決するために提案するチャンク単位で拡張を行う HORT について説明する。

3.1. 経歴･オフセット法の問題点

HORT では､関係テーブルのレコードを示す論理拡張可能配列上での有効要素の位置を、その要素が属する部分配列の経歴値と、その部分配列内オフセットの２つの値の組で表現する経歴・オフセット法を用いている。しかしこの表現方法では、部分配列内オフセットに割り当てられている記憶領域を有効に使用できない。例えば､経歴値が 0 や 1 の部分配列のサイズは 1 であるなど、経歴値の小さな部分配列においては、オフセットを格納するための変数の取り得る値の領域をほとんど使用していない。またもうひとつの問題として､HORT で実装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ると、経歴値またはオフセットの値が格納する変数の取り得る値の上限を超えてオーバーフローが起こってしまう。経歴値に 32bit、オフセットに 64bit を割り当て、n 次元論理拡張可能配列をなるべく各次元のサイズを均等に最大まで拡張した時の 1 辺のサイズ､つまり 1 カラム当たりの挿入可能なカラム値の種類と、本来、経歴値とオフセットの組で表現できるアドレス空間、この例では 2^{96} の使用率

を計算すると、最もアドレス空間の使用率が高くなる3次元の時でも、全体の3.07%しか使用できていない。この時の、1カラム当たりの挿入可能なカラム値の種類は1,431,655,766種類である。これ以降、次元が増えるにつれアドレス空間の使用率は下がり、挿入可能なカラム値の種類も少なくなる。例えば10次元の時の空間使用率は$3.00*10^{-6}$%で、1カラム当たりの挿入可能なカラム値の種類は138種類となってしまう。

この問題の解決策の1つとして、論理拡張可能配列を要素単位で拡張を行うのではなく、ある一定要素の塊であるチャンク単位で拡張を行うことにより、経歴値とオフセットの組で表現できるアドレス空間をより有効に使えるようにする。

3.2. チャンク単位で拡張を行う拡張可能配列

ある一定要素の塊であるチャンクを単位として拡張を行う拡張可能配列では、チャンク毎に拡張された順番を表すチャンク番号を振っていく。また、拡張に際しては2.2における要素単位の部分配列に対して、チャンク単位の部分配列が確保される。以降この部分配列をチャンクサブ配列と呼ぶ。この時、拡張の順番を示す経歴値とそのチャンク部分配列内の先頭チャンクの番号をHORTテーブルに持たせる。図4に、関係テーブルをチャンク単位で拡張を行う拡張可能配列で表現した例を示す。

関係テーブルをチャンク単位で拡張を行う拡張可能配列で表現すると、拡張可能配列内でのレコードの位置情報を、各チャンクに一意にふられたチャンク番号と、チャンク内オフセットの2つの値を用いて表現する。チャンクのサイズを、チャンク内オフセットに割り当てられた変数が取り得る最大値に近づけることで、経歴・オフセット空間の使用率を上げることができる。要素単位で拡張を行った時と同様に、チャンク番号に32bit, チャンク内オフセットに64bitを割り当て、n次元論理拡張可能配列をなるべく各次元のサイズを均等に最大まで拡張した時の1辺のサイズと、本来、チャンク番号とチャンク内オフセットの組が表現できる空間の使用率を計算する。3次元の時は、アドレス空間の使用率は99.9%となり、1カラム当たりの挿入可能なカラム値の種類は4,294,529,957種類、10次元の時は、アドレス空間の使用率は89.7%となり、1カラム当たりの挿入可能なカラム値の種類は768種類まで増える。この様に、チャンク単位で論理拡張可能配列を拡張しチャンク番号とチャンク内オフセットの2つの値を用いてレコードを表現することにより、経歴・オフセット空間の使用率が低いという問題を解決することができる。経歴・オフセット空間の使用率を上げることにより、従来の要素単位で拡張を行った時よりも1カラムあたりのカラム値の種類を増やすことができる。従って、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できる。図5にチャンク単位で拡張を行うHORTの構造の例を示し，以下でその概要と構造について説明する。以後，チャンク単位で拡張を行うHORTのことをC-HORTと呼ぶこととする。

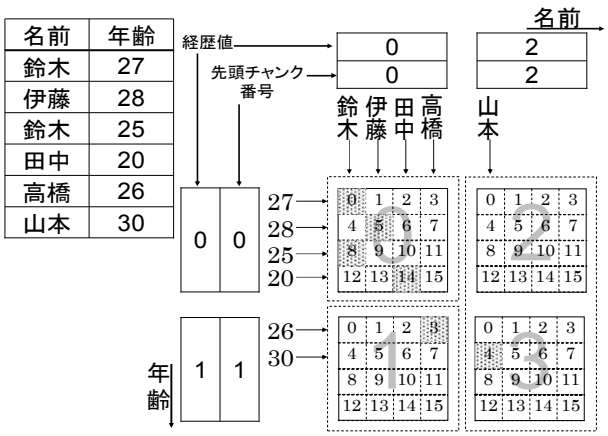

図4:チャンク単位で拡張を行う拡張可能配列の実装例

3.3. C-HORTの構造

C-HORTにおける各次元のHORTテーブルは、チャンク部分配列の情報を記録するチャンク情報テーブルと、カラム値の情報を記録するためのカラム値情報テーブルの2つに分けられる。チャンク情報テーブルはチャンク部分配列ごとに作成され、経歴値やチャンク部分配列内の先頭チャンク番号とチャンク数、係数ベクトルが格納される。また、カラム値情報テーブルはカラム値ごとに作成され、カラム値とレコードカウンタが格納される。なお、全てのチャンクのサイズを同一とすることにより、チャンク内オフセットを求めるための係数ベクトルはC-HORTに唯一保持するだけでよい。また、HORTではカラム値毎に保持していた部分配列の経歴値と係数ベクトルを、C-HORTではチャンク部分配列毎に保持するだけでよくなるため、空間的コストを抑えることができる。

3.4. 配列添字の組からチャンク番号とチャンク内オフセットへの変換

チャンク単位で拡張されるn次元論理拡張可能配列のある要素の添字が$(i_1, i_2, \cdots, i_n)$、チャンクの各辺のサイズが$(c_1, c_2, \cdots, c_n)$であったとすると、各次元のチャンク情報テーブルの添字は$(\lfloor i_1/c_1 \rfloor, \lfloor i_2/c_2 \rfloor, \cdots, \lfloor i_n/c_n \rfloor)$で求められる。最も大きな経歴値を持つチャンク部分配列内に目的の要素が含まれているので、この添字を基に各次元のチャンク情報テーブルにアクセスし、経歴値を比較することで、的の要素が含まれるチャンク部分配列を特定することができる。次に、そのチャンク部分配列の係数ベクトルと、各次元のチャンク情報テーブルの添字とチャンク

部分配列内の先頭チャンク番号を用いて目的の要素が含まれているチャンクの番号を計算することができる。また、このチャンクにおける目的の要素の添字は（$i_1\%c_1, i_2\%c_2, \cdots, i_n\%c_n$）で求められるため、チャンクの係数ベクトルを用いてチャンク内オフセットを求めることができる。また、チャンク番号とチャンク内オフセットの組をカラム値の組に逆変換する際には、まず、そのチャンクが含まれているチャンク部分配列の係数ベクトルを用いてチャンクのチャンク部分配列内での添字、すなわち各次元のチャンク情報テーブルの添字を求める。次に、チャンクの係数ベクトルを用いてチャンク内オフセットからチャンク内での要素の添字を求め、これらを基に各次元のカラム値情報テーブルにアクセスすることによりカラム値を求める。このように、配列添字の組からチャンク番号とオフセットの組への変換ならびに逆変換時の計算コストは、HORT において配列添字の組から経歴値とオフセットの組への変換ならびに逆変換時の計算コストよりも大きくなる。

名前	年齢
鈴木	27
伊藤	28
鈴木	25
田中	20
高橋	26
山本	30

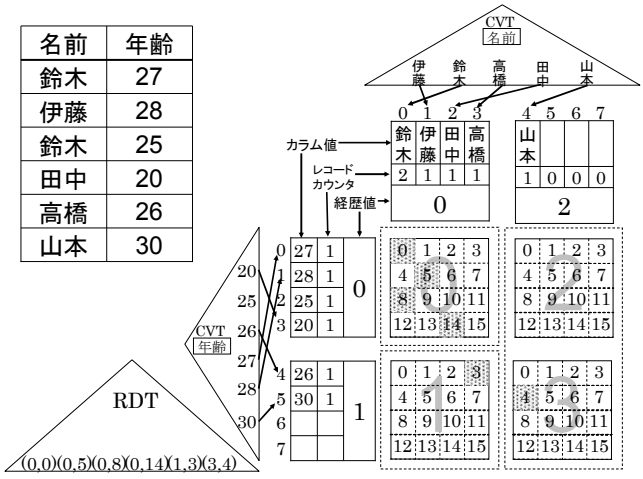

図 5:C-HORT の構造

3.4. C-HORT におけるレコードの挿入と削除

レコードの挿入を行う際には、まず、挿入対象のレコードの各カラム値が論理拡張可能配列の対応次元の添字と対応付けられているかどうかを、CVT 内を探索することによって調べる。もし、カラム値が配列添字と対応付けられていなければ、現在使用されていない配列添字と対応付ける。この時、使用されていない配列添字が無ければ、論理拡張可能配列を対応次元方向にチャンク単位で 1 つ拡張する。次に、挿入対象のレコードの各カラム値を各次元の配列添字の集合に変換し、チャンク番号とチャンク内オフセットの組に変換、それを RDT に格納する。また、各カラム値のレコードカウンタを 1 つインクリメントする。また、レコードの削除を行う際には、HORT におけるレコードの削除処理と同様に、まず、削除対象のレコードの各カラム値を CVT を用いて配列添字に変換し、その組をチャンク番号とチャンク内オフセットの組に変換、その組を RDT から削除する。次いで、削除したレコードの各カラム値のレコードカウンタの値を 1 つデクリメントし、0 になれば、CVT からこのカラム値を削除する。

3.5. C-HORT におけるレコードの検索

C-HORT におけるレコードの検索は、RDT に格納されているチャンク番号とオフセットの組に対して行われる。HORT におけるレコードの検索と同様に、検索条件として、1 つのカラムにおいてカラム値が 1 つ指定された場合の単一値指定検索における検索手法を説明する。まず、カラム値を指定されたカラムの CVT を探索して論理拡張可能配列の添字に変換し、さらにカラム値をチャンク情報テーブルの添字とカラム値情報テーブルの添字に変換する。ここで、求まった添え字に対応するチャンク部分配列を基チャンク部分配列と呼ぶ。この時、カラム値が CVT 内に存在しなければ、指定されたカラム値を持つレコードは存在しない。次に、それらの添字を基に RDT 内の検索を行うが、その検索方法として以下の 3 種類の方法を考える。なお、検索条件として指定されたカラム値のレコードカウンタの値だけの検索対象レコードが見つかった時点で検索を終了する。

3.5.1. 検索方法 1

まず、基チャンク部分配列の先頭チャンク番号とチャンク内オフセット 0 をキーとして RDT をルートノードから探索し、基チャンク部分配列内の先頭レコードを取り出す。その後、RDT のシーケンス・セット部を終端まで辿ることにより、基チャンク部分配列以降に拡張されたチャンク部分配列内のレコードを全て取り出す。取り出したレコードであるチャンク番号とチャンク内オフセットの組から、3.4 で説明したチャンク番号とチャンク内オフセットの逆変換方式を用いて、次元の配列添字を求めることにより、検索対象のレコードであるかどうかを判定する。図 6 に、検索対象が、第 1 次元の添え字が 5 である場合の、論理拡張可能配列の検索範囲と RDT の探索範囲の例を示す。

3.5.2. 検索方法 2

まず、基チャンク部分配列内の先頭チャンク番号とチャンク内オフセット 0 をキーとして、RDT をルートノードから探索し、基チャンク部分配列内の先頭レコードを取り出す。その後、RDT のシーケンス・セット部を辿ることにより、基チャンク部分配列内のレコードを全て取り出す。ただし、基チャンク部分配列内のレコード全てが検索対象のレコードではないため、チャンク番号とオフセットの組から検索対象の次元

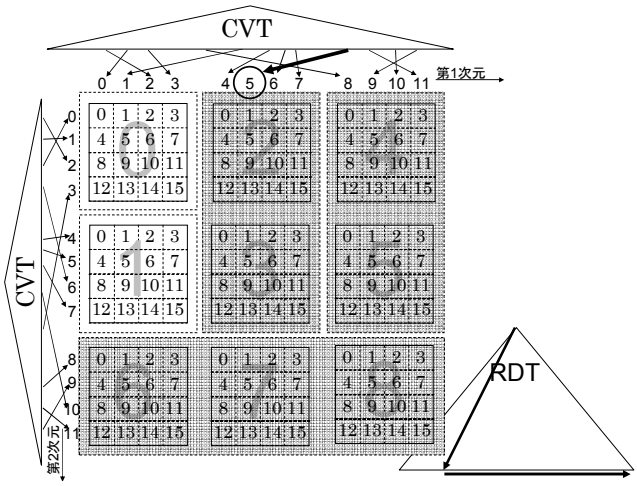

図６：検索方法１の例

の配列添字を求め、検索対象のレコードであるかどうかを判定する必要がある。

続いて、検索対象の次元以外の次元に属するチャンク部分配列のうち、基チャンク部分配列よりも大きな経歴値を持つチャンク部分配列それぞれについて、検索対象レコードが存在しうるチャンク内のレコードを全て取り出し、検索対象の次元の配列添字を求め、検索対象のレコードであるかどうかを判定する。図７に、検索対象が、第１次元の添え字が５である場合の、論理拡張可能配列の検索範囲と RDT の探索範囲の例を示す。

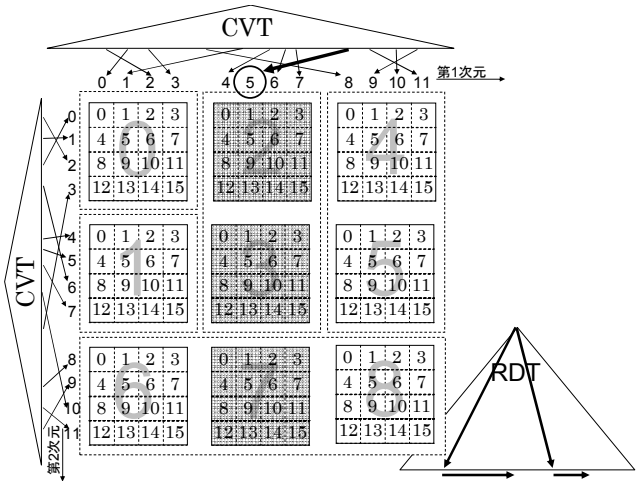

図７：検索方法２の例

3.5.3. 検索方法 3

まず、基チャンク部分配列内の検索を行う。基チャンク部分配列内には、非検索対象要素が含まれているため、検索対象要素のオフセットが連続している区間を求め、検索を行う。

次に、検索対象の次元以外の次元に属するチャンク部分配列のうち、基チャンク部分配列よりも大きな経歴値を持つチャンク部分配列それぞれについて検索を行う。C-HORT における検索方法 2 と同様に検索対象レコードが存在しうるチャンクを求め、それぞれのチャンク内において検索対象要素のオフセットが連続している区間を求めることにより検索を行う。なお、全てのチャンクのサイズは同じであるので、索対象要素のオフセットが連続している区間の長さと、チャンク内に存在しているこの区間の数は全て同じである。

この方法を用いることにより、RDT から取り出したレコードが検索対象のレコードであるかどうかを判定する必要はなくなる。図８に、検索対象が、第１次元の添え字が５である場合の、論理拡張可能配列の検索範囲と RDT の探索範囲の例を示す。

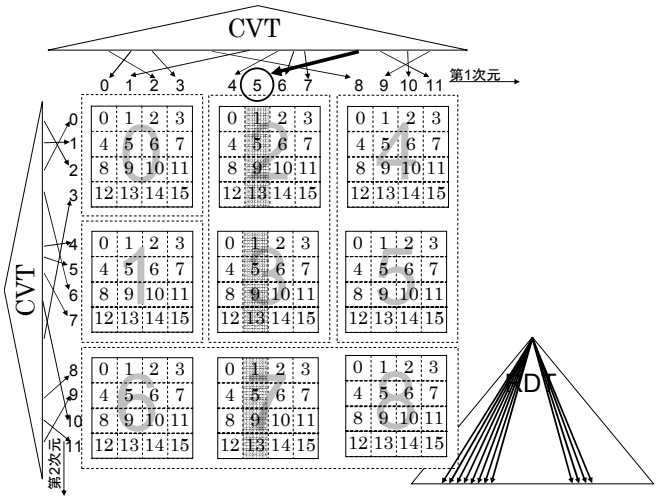

図８：検索方法３の例

3.6. RDT の分割

以下では、C-HORT における問題点の提起し、その対策の説明を行う。

3.6.1. C-HORT の RDT における問題

C-HORT に限らず従来の HORT も同様に、HORT における挿入時間、検索時間において大きなコストを占めているのが、RDT のディスクアクセス時間である。RDT は１つの関係テーブルに１つで、全てのレコードを管理している。RDT は B⁺-tree であるので、レコードが増えるにつれ高さが高くなり、時間的コストが大きくなってしまう。ここで、C-HORT の場合、論理拡張可能配列がチャンク単位で分割されていることに注目して、チャンク毎に RDT を持たせることでこの問題を減少させる。

3.6.2. RDT の分割

図９に RDT をチャンク毎に分割した例を示す。RDT をチャンク毎に分割することにより、1 つの RDT に挿入されるレコードの数が少なくなるため、B⁺-tree の高さが低くなり、挿入時間や検索時間が早くなると考えられる。また、チャンク毎に RDT を持たせることにより、チャンク番号とチャンク内オフセットの２つの値で表されていたレコードの位置情報を、チャンク

内オフセットのみで表すことができるため、RDT の空間的コストも抑えることができる。

また、検索については、3.5 で述べた方法をほぼそのまま使えるが、RDT がチャンク毎に分割されているため、検索方法 1 に相当する検索方法は存在しない。

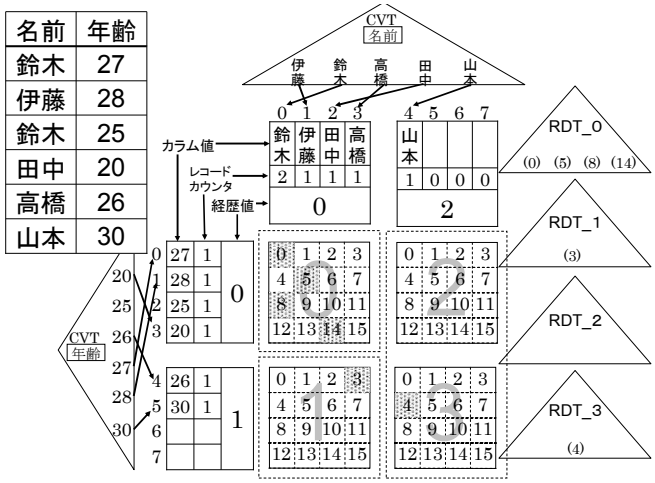

図 9 ：C-HORT において RDT 分割の例

4. 比較評価

以下では、HORT, C-HORT を実装し、HORT と C-HORT において、空間的コストおよび時間的コストの実測値を比較評価する。ただし、RDT は 2 次記憶上に配置し、CVT および HORT テーブルは比較的小さいために実行時に 2 次記憶上の原本をメモリ上にロードして使用している。

実行環境は、Sun Fire E4900 (CPU : UltraSPARC IV , クロック速度:1050MHz, メモリ容量:48 GB, ディスク容量 : RAID1 (73.4 GB×2) RAID5 (73.4 GB×12), OS : Solaris 9)を使用した。

4.1. PostgreSQL、HORT、C-HORT との比較

int 型のカラム, カラム数 5 , 各カラムのカラム値の種類 25000 である関係テーブルを、postgreSQL 8.1.2、HORT および C-HORT で表現し、レコードを 100 万件と 500 万件挿入した時それぞれの、単一値指定検索において検索一件当たりの平均検索時間の比較を図 10 に示す。また、C-HORT の検索は検索方法 2 を使用する。ここで、C-HORT のチャンクの一辺のサイズは 7131 である。

既存の DBMS である postgreSQL と比べ、HORT の検索時間は約半分になっていることがわかる。さらに、C-HORT の検索コストは HORT と比べ約 5 分の 1 となった。これは、HORT の拡張可能配列が要素毎に拡張を行うため、大量のデータが挿入されると部分配列の数が非常に多くなるため、それに比例して検索時に RDT をルートノードから辿る回数が多くなるからである。それに対して、C-HORT の場合、検索時に RDT をルートノードから辿る回数は、最大 n-1 次元スライスの断面に存在するチャンク個数である。例えば、該当の関係テーブルにおいては、HORT の場合カラム値毎に部分配列を持っているので、124,996 個の部分配列が存在する。RDT をルートノードから辿る最大回数は 124,996 回となる。それに対して、C-HORT の場合は、各次元の一辺のチャンクの数は 4 個であり、n-1 次元スライスの断面に存在するチャンク数は 4^4 で 256 個となるので、RDT をルートノードから辿る最大回数は 256 回となる。シーケンス・セット部を辿るよりも、ルートノードから辿るコストの方が高いので、ルートノードから辿った回数が少なくなる C-HORT の検索のコストが小さくなった。

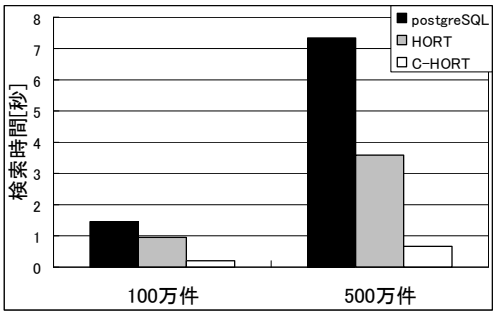

図 10 : 単一値指定検索の比較

4.2. C-HORT において、RDT 分割前後の比較

int 型のカラム, カラム数 5, 各カラムのカラム値の種類 20,000 である関係テーブルを C-HORT で表現し、レコードを 500 万件挿入した時の RDT 分割前後において、挿入時間、RDT サイズ、検索速度の比較を表 1 に示す。また、検索は検索方法 2 を使用する。ここでチャンクの一辺のサイズは 7131 である。

表 1 : RDT 分割前後の比較

	RDT 分割前	RDT 分割後
挿入時間[秒]	1526.54	1113.49
RDT サイズ[MB]	約 150	約 120
検索時間[秒]	0.672	0.631

RDT をチャンク毎に分割すると、挿入のコストは約 3 分の 1 となった。これは 3.6 で説明したように、RDT を分割したことにより、RDT 分割前の時と比べ、それぞれの RDT の高さが低くなったためである。今回の関係テーブルの挿入時において、RDT の高さの平均は、RDT 分割前が 2.9765 だったのに対して、RDT 分割後は 1.9752 となり、挿入時間の比と近い値となった。また、RDT のキー値が、チャンク番号とチャンク内オフセットの 2 つの値からチャンク内オフセットのみになったことにより、RDT の空間的コストも小さくなった。しかし、検索時間は、RDT を分割してもそれほど

減少していない。これは、RDT の高さが低くなることでコストが小さくなるのは、RDT をルートノードから辿るコストだからである。挿入時は、1 つのレコードの挿入の度に RDT をルートノードから辿っているため、高さが低くなることで大幅に時間を短縮できたが、検索においては、それぞれの RDT 毎に、一度ルートノードから辿り、その後はシーケンス・セット部を辿って検索しているため、検索時間はそれほど短縮されなかったと考えられる。

4.3. 検索方法についての比較

RDT 分割前の C-HORT の検索方法 1~3 と RDT 分割後の検索方法 2~3 において、充填率を変化させた時の、単一値指定検索一件当たりの平均検索時間の変化の比較を図 11 に示す。ここで、語頭に "div_" と付いているのが RDT を分割した C-HORT である。

検索条件は、int 型のカラム，カラム数 4，各カラム値の種類 80 である関係テーブルをチャンクの一辺のサイズを 16 とした C-HORT で表現して測定した。

検索方法 1 については、他の検索方法と比べほとんどの場合で検索時間が長い。また、充填率が上がるにつれ検索時間の増大が、他の検索方法より顕著に見られる。検索方法 1 は、基チャンク部分配列以降に拡張されたチャンク部分配列内のレコードを全て取り出して一致するかを判別しているため、かなり広範囲を検索しているためである。検索方法 2 と検索方法 3 の検索時間については、充填率が低い場合では検索方法 2 の方が短いが、充填率 20%付近を境に逆転している。検索方法 2 は検索方法 3 と比べ、論理拡張可能配列内で検索される範囲は広いが、RDT をルートノードから辿る作業を減らし、シーケンス・セット部を多く辿っている。逆に検索方法 3 は、論理拡張配列内の検索される範囲は狭いが、RDT をルートノードから辿る作業が多くなっている。RDT はルートノードから辿る作業の方がシーケンス・セット部を辿る作業よりコストが大きいため、充填率が低い場合は、検索範囲の大きい検索方法 2 が検索時間は短い。しかし、充填率が上がると、検索方法 2 のシーケンス・セット部を辿る回数が増し、検索方法 3 のルートノードから辿るコストを上回るため検索時間は短くなる。

また、RDT 分割前後の検索方法 2 の検索時間がほぼ変わらない理由は 4.2 で説明したが、RDT 分割前後の検索方法 3 では RDT 分割後の方が検索時間は短くなっている。これは、検索方法 3 では RDT をルートノードから辿る回数が多いので、分割により RDT の高さが低くなる恩恵を多く受けたためである。

6. まとめ

本論文では、拡張可能配列を要素単位ではなく、チャンク単位で拡張を行うことにより、HORT における最も大きな問題点である経歴・オフセット空間の狭小の問題を解決した。　さらに、従来の HORT では、拡張可能配列におけるレコードの位置情報をその要素が含まれる部分配列の経歴値と部分配列内オフセットの 2 つの値を用いて表現していたが、提案した方式では、RDT をチャンク毎に持たせることにより、チャンク内オフセットの 1 つのみの値で表現している。本方式の利点は、チャンク内オフセットを用いてレコードを表現するため、より大規模な関係テーブルの実装が可能になったことである。また、以上の方式を実装し、従来の HORT と提案した実装方式の空間的コストと検索時の時間的コストの比較を行い、本方式が有効であることを示した。

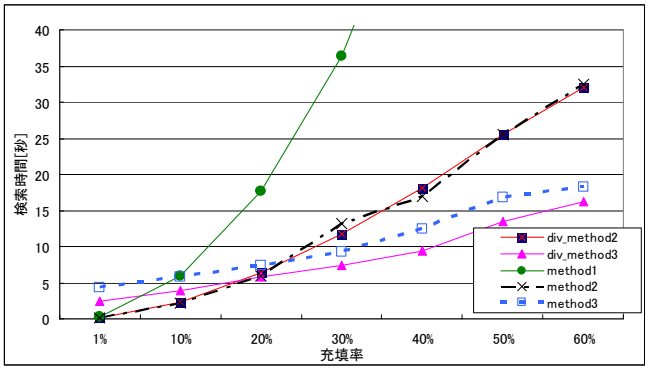

図 11:充填率の変化させた時の、
検索時の時間的コストの変化

文　　献

[1] Masayuki Kuroda, Naoki Azuma, K. M. Azharul Hasan,Tatsuo Tsuji, Ken Higuchi, " An Implementation Scheme of Relational Tables", Proc. of ICDE 2005 Workshop, p. 1244,2005.

[2] K. M. Azharul Hasan, Masayuki Kuroda, Naoki Azuma, Tatsuo Tsuji, Ken Higuchi, "An Extendible Array Based Implementation of Relational Tables for Multi Dimensional Databases", DaWaK 2005, LNCS 3589, pp. 233-242, 2005.

[3] A. L. Rosenberg, "Allocating Storage for Exte -ndible Array' s", JACM, Vol. 21, pp.652-670, 1974.

[4] A. L. Rosenberg, L. J. Stockmeyer, "Hashing Schemes for Extendible Arrays", JACM, Vol.24, pp.199-221, 1977.

[5] E. J. Otoo, T. H. Merrett, "A Storage Scheme for Extendible Array' s", Computing, Vol.31, pp.1-9, 1983.

[6] A. Novacek, "Using Time Stamps for Storing and Addressing Extendible Arrays", Computing, Vol.37 pp.303- 13, 1986.

[7] 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.