多次元データのコンパクトな実装とその性能評価

前田 卓哉† 水野 広治‡ 都司 達夫† 樋口 健†

†福井大学工学研究科 〒910-0011 福井市文京 3-9-1
‡福井大学工学部 〒910-0011 福井市文京 3-9-1
E-mail: †{maeda,tsuji,higuchi}@pear.fuis.u-fukui.ac.jp ‡mizuno@fuis.fuis.u-fukui.ac.jp

あらまし 本稿では動的に増大する多次元データのエンコード方式である経歴・パターン法に基づいた多次元データの実装について述べる．経歴・パターン法は拡張可能配列の柔軟な拡張性に基づいており，我々がこれまで提案してきた経歴･オフセット法によるエンコード方式と長所，短所が相反しており，相補的である．構築したプロトタイプシステムについて，その性能を比較評価した結果，PostgreSQL と比較して記憶コストと検索コストの両面で高い性能を示した．

キーワード 多次元データ，経歴・パターン法，拡張可能配列，PostgreSQL

A Compact Implementation of Multidimensional Datasets and Its Evaluation

Takuya MAEDA† Hiroharu MIZUNO‡ Tatsuo TSUJI† and Ken HIGUCHI†

†Graduate School of Engineering, University of Fukui 3-9-1 Bunkyo, Fukui-shi, Fukui, 910-8507 Japan
‡Faculty of Engineering, University of Fukui 3-9-1 Bunkyo, Fukui-shi, Fukui, 910-8507 Japan
E-mail: †{maeda,tsuji,higuchi}@pear.fuis.u-fukui.ac.jp ‡mizuno@fuis.fuis.u-fukui.ac.jp

1. はじめに

動的に増大する多次元データのエンコード方式として，我々が提案してきた経歴･オフセット法[3][4]は拡張可能配列[1][2]の仕組みを取り入れることにより，多次元データの増大に対して，効率よく対処できる．

挿入したタプル中に新たな属性値が出現した場合，そのタプルを収容する部分配列が動的に確保され，現在の拡張可能配列の当該次元方向に付加される．経歴･オフセット法ではタプルは，拡張可能配列中，それが含まれる部分配列の拡張経歴値とその部分配列におけるオフセットの対にエンコードされる．この方式の基本的な利点は次の3点である．

(1) タプルの次元数nに関わらず，経歴値とオフセットの2つのスカラー値でタプルを表現でき，システム内部でコンパクトな固定長でタプルを扱える．
(2) 単純な四則演算でタプルのエンコード・デコードが可能．
(3) タプルの動的な追加・削除に対して，タプル集合全体を再配置することなく，柔軟に対応し得る．

[5]では，これらの利点を活かしつつ，経歴･オフセット法におけるタプルアクセス速度と記憶効率を向上させるための経歴・パターン法と呼ぶ新たな方式について述べられている．この方式では拡張可能配列のアドレス関数計算を行うことなく，シフトやマスク演算等のレジスタ命令のみでより高速にタプルのエンコードとデコードが可能である．また，必要な記憶量を削減することができる．ただし，経歴･オフセット法に比べて，エンコード結果を収容する論理空間の飽和をより，早めてしまうことが本方式の欠点であるが，経歴・オフセット法と相補的に棲み分けを行うことを前提としている．経歴･パターン法に基づいた多次元データの実装方式として[5]で述べられている経歴・パターンの格納構造としてB+木を使用する方式の他に本稿では従来の関係テーブルの実装と同様に，経歴・パターンを2次記憶上に逐次配置する実装方式について述べ，構築したプロトタイプシステムにおいて，それらの性能を比較評価する．

2. 経歴・パターン法とその実装

2.1. 経歴･パターン法

n 次元データの n 個の属性値 $v_1, v_2, \ldots, v_n$ からなる n 次元タプルは適切なデータ構造により，n 次元空間の 1 点の座標 $P(i_1, i_2, \ldots, i_n)$ に変換される．ここで，述べる経歴パターン法ではこの n 次元空間は n 次元拡張可能配列 A により表され(図 1)，P は A の添字座標である．

A の各次元の添字は 0 から始まるものとし，次元 i の現在の最大添字を s_i とすると，次元 i の実サイズは s_i+1 であり，これは次元 i の属性のカージナリティである．一般に，0 以上の整数 m の表現に要するビット数を $b(m)$とすると，$2^{b(s_i)}$ を次元 i の論理サイズという．

ここで，n 次元拡張可能配列 A の現在の各次元の最大添字を $[s_1, s_2, ..., s_n]$ とする．新たなタプルの挿入により，ある次元 k（$1 \leqq k \leqq n$）の方向に A のサイズが１つ拡張して，$b(s_k) < b(s_k+1)$となったとき（すなわち，次元 k の最大添字が論理サイズを超えたとき）A は次元 k の方向に拡張される．拡張分の部分配列 ΔA の各次元のサイズは元の A と同一であり，拡張後の配列の最大添字は$[s_1, ..., s_{k-1}, s_k+1, s_{k+1}, ..., s_n]$ となり，また，論理サイズは $[2^{b(s_1)}, ..., 2^{b(s_{k-1})}, 2^{b(s_k+1)}, 2^{b(s_{k+1})}, ..., 2^{b(s_n)}]$ となる．このとき，拡張の順番を指定するための経歴値のカウンタ h は 1 インクリメントされ，その値 h+1 が次元 k の経歴値テーブル H_k に格納される．また，次のベクトル V,

$V = <b(s_1), ..., b(s_{k-1}), b(s_k)+1, b(s_{k+1}), ..., b(s_n)>$

は ΔA の境界ベクトルとよばれ，経歴値 h+1 の下に，拡張次元 k とともに境界ベクトルテーブル B に記録される（図 1）．

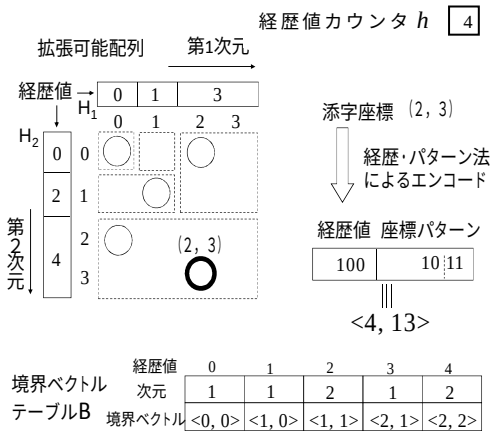


図 1　経歴・パターン法の説明図

配列の要素位置を指定する n 次元座標 $I = (i_1, i_2, ..., i_n)$ は以下のように経歴値 h と座標パターン p の対にエンコードされる．境界ベクトル V はこのエンコードに使用されるとともにエンコードされた対を元の n 次元座標にデコードするのにも使用される．経歴値 h は I の各次元座標のビットパターンサイズ $[b(i_1), b(i_2), ..., b(i_n)]$ について，各次元の経歴値テーブル H_k を参照し，$H_k[b(i_k)]$（$1 \leqq k \leqq n$）の最大値を求めればよい．この h に対して，$B[h]$ の境界ベクトルは各次元の添字のビットパターンのサイズを与える．このサイズにしたがって，座標パターン p は次元の高い添字パターンから座標パターン格納域 (たとえば 1 long 値長 64 ビット) の下位ビットから順に配置して得られる．

例えば，図 1 の拡張可能配列において，配列要素(2,3) をエンコードする場合，(2,3) が属している部分配列の経歴値は 4 であり，$B[4]$を参照して，境界ベクトルは <2,2>であることを知る．これより，座標パターンは最下位ビットから 2 次元目の 3，次の 2 ビットで 1 次元目の 2，の添字パターンの連接となる．したがって，経歴値とこのパターンとを対とした $<4,1011_2>$ にエンコードされる．例えば，この対の格納サイズを 16 ビットの short 型とする場合，経歴値より座標パターンのほうが多数のビットを使うので上位４ビットを経歴値，下位 12 ビットを座標パターンとすることが考えられる．逆に，エンコード結果＜経歴値 h，座標パターン p＞ から元の n 次元座標 $I = (i_1, i_2, ..., i_n)$ へのデコードは，まず，$B[h]$より要素 I が属する部分配列の次元 k を求める．つづいて，$B[h]$の境界ベクトルにしたがって p 中の各次元の座標値 i_k（$1 \leqq k \leqq n$）を分離して取り出せばよい．

2.2. 経歴パターン法の実装 ：HPMD

n 次元 HPMD (History-Pattern implementation of Multidimensional Datasets) は経歴・パターン法の実装方式の一つであり，以下のデータ構造から構成される．

(1) n 個の B+木，CVT_i（1 i n）と 1 個の B+木，RDT．CVT_i は次元 i の属性値をキーとして，対応する拡張可能配列の添字を記録する．また，RDT のキーとして経歴・パターン法によるタプルのエンコード値が格納される．

(2) 各次元毎に経歴値を格納する n 個の経歴値テーブル H_i（1 i n），および，経歴値を添字として，経歴値に対応する拡張次元と境界ベクトルを格納した境界ベクトルテーブル B．

(3) 各次元の添字ごとにその添字に対応する属性値およびその属性値を持つすべてのタプルの数を記録する属性値テーブル C_i（1 i n）．

配列の拡張に対して柔軟に効率よく対応できる拡張可能配列の仕組みは，各次元の添字のビットパターンサイズを一律に固定することなく，各次元の座標値のビットパターンの境界を次元拡張の状況に応じて，柔軟に設定していることに活かされているといえる．ただし，各次元の添字座標値のビットパターンのサイズの総和は定められた固定長(たとえば 1 long 値長 64 ビット)以下である．このような，境界設定の柔軟性を確保した上で，機械語のレジスタ命令のみで，エンコードとデコードが可能である．すなわち，エンコード時の各次元の添字パターンの連接およびデコード時の各次元添字の取り出しは，シフト演算および論理和，論理積演算のレジスタ命令のみの実行により可能であり，固定配列のように乗除算を含むアドレス関数の計算は不要である．

配列の次元数を n とすると，境界ベクトルテーブルの記憶コストは $O(n)$であり，また，各次元の経歴値テーブルサイズの和は各次元長を$[s_1, s_2, ..., s_n]$とする

と，

$$\lceil \log_2 s_1 \rceil + \lceil \log_2 s_2 \rceil + \cdots + \lceil \log_2 s_n \rceil$$

となる．以上より，経歴値テーブルや境界ベクトルテーブルの記憶コストは CVT, RDT および属性値テーブルなどの HPMD の他の構成要素に比べて，ほとんど無視できる．

属性値テーブルの属性値はタプルへのデコードと検索に必要であり，またタプル数は検索の最適化プランを立てるのに必要とされる．RDT は多次元データセットに実際に登録されているタプルのみのエンコード値を格納しており，登録されていないタプルは記憶域を占めない．これにより，一般に多次元データセットを多次元配列に記憶する際の際の問題点である疎配列性の問題を解消している，

図 1 の拡張可能配列の論理サイズは [4,4] であり，また，各次元の実サイズは [3,4] である．経歴・パターン法では，n 次元データセットの場合，実サイズ空間の論理サイズ空間に対する大きさの割合は $1/2^n$ ～1 の範囲である．

2.3. 境界ベクトルの実装

前節で述べた境界ベクトルは経歴・パターン法において，核になるデータ構造の１つである．ここでは，n 次元 HPMD の境界ベクトルの実装を示しておく．境界ベクトルの要素は，次の構造体 struct v_mask で表される．

```
struct   v_mask {
    unsigned char   ppos; // 次元のサイズパターン
                             の開始ビット位置
    unsigned int    mp;    // マスクビットパターン
}
```

struct v_mask の配列 v を境界ベクトルの実装とする．

```
struct   v_mask   v[n];
```

たとえば，6 次元 HPMD において， <6,10,7,20,15,4> を境界ベクトルとすると，配列 v は，

	ppos	mp
0	56,	0x3F
1	46,	0x3FF
2	39,	0x7F
3	19,	0xFFFFF
4	4,	0x7FFF
5	0,	0xF

となる．タプルエンコード後の座標パターンのサイズは境界ベクトルの各包含サイズのビットパターンを高次元のものから順に並べてできるビットパターンのサイズ,すなわち各次元の包含サイズの和であり，これは，v[0].ppos + (v[0].mp のビットパターンサイズ) である．上記の境界ベクトルの例の場合には 56 + |0x3F| = 56 + 6 = 62 である．境界ベクトル v に対して，境界ベクトルテーブル B は

```
struct {
    unsigned char   dim;      // 次元
    struct v_mask   v[ ];     // 境界ベクトル
} B[ ];
```

と定義される．添字座標 $(i_1, i_2, \ldots, i_n)$ から＜経歴値，座標パターン＞へのエンコードおよび逆のデコードはこの境界ベクトルテーブル B を参照することにより高速に行うことができる．たとえば，＜経歴値，座標パターン＞をデコードして各次元 i（$1 \le i \le n$）の添字を求めるには，座標パターン P を v[i-1].ppos ビット分, 右にシフトして v[i-1].mp のマスクビットパターンとの論理積を求めればよい．

2.4. チャンク化 HPMD：C-HPMD

本節では，2.2 節で述べた HPMD について，[4]と同様にチャンク化を行う，チャンク化された HPMD を以後，C-HPMD と呼ぶ．C-HPMD の利点は，次のようにまとめられる．

(1) HPMD における検索範囲の肥大化の問題点を改善することができる．

(2) 拡張可能配列のランダムアクセス性を活かした検索が可能である．

(3) タプルの論理空間を格段に拡げて，より，大規模な多次元データセットを収容することができる．

(4) 並列検索への高い適応性を有する．

n 次元の C-HPMD の拡張可能配列において，チャンクとは 1 辺が 2^r（r 1）の n 次元超立方体であり，r をチャンクのランクという．HPMD は配列要素からなる部分配列を単位として拡張されるのに対して C-HPMD はチャンクからなる部分配列を単位として拡張される．HPMD におけるタプルのエンコードは<経歴値，座標パターン>の対であったのに対して，C-HPMD ではタプルは<チャンク番号，チャンク内座標パターン>の対にエンコードされる．チャンク番号の決定には 2.1 節述べた拡張可能配列の要素アドレスの計算方法が使用される．図 2 にランクが 1 のチャンク化した 2 次元拡張可能配列の例を示す．配列の中の数字はチャンク番号を表す．HPMD の各配列要素をチャンクに対応させると C-HPMD は HPMD とほぼ同等のデータ構造である．また，HPMD に関する 3 節の説明はタプルのエンコード/デコード方式と検索方式以外はほぼそのまま C-HPMD にも適用できる．マシン語長が 64 ビットの場合，C-HPMD では，チャンク内座標パターンとして 64 ビット，チャンク番号として 64 ビットの計 128 ビットの空間を確保でき，元のタプルの論理空間を格段

に拡げることができる．

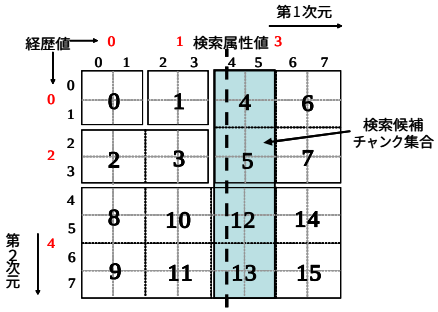


図 2　ランク 1 の C-HPMD における検索範囲

3. 経歴パターン法の実装 ：HPRD

前章で提案した HPMD では RDT のランダムアクセス機能により，＜経歴値，座標パターン＞のキーを高速にサーチすることができる．また，次元依存性や，経歴値依存性が存在するものの，拡張可能配列の特性に基づいて，検索範囲を狭めることが可能である．このような利点は，たとえば多次元配列をベースとする Multidimensional OLAP（MOLAP）の基本的なオペレーションであるスライス・ダイス操作に対しては，有利に働く．また拡張可能配列の柔軟な拡張性に注目して，固定配列では不可能な，効率的な差分処理が可能である．一方，B+木のキーとして＜経歴値，座標パターン＞を高速にサーチするためには，タプルのエンコードである＜経歴値，座標パターン＞は固定サイズキーである必要がある．ここでは，＜経歴値，座標パターン＞は固定サイズではなく，座標パターンのサイズに応じて可変とするような実装方式を提案する．

3.1. 可変サイズ座標パターン

n 次元 HPMD において現在の経歴値を h として，その境界ベクトルの次元 k $(1 \le k \le n)$ の要素を e_k とする．タプルの挿入により次元 k の方向に HPMD を拡張したときには，拡張された部分配列の経歴値は h+1 となり，その境界ベクトルの次元 k の要素は e_k+1 となることから，次の基本的な性質が成り立つことがわかる．

［性質 1］経歴値はそれに対応する境界ベクトルの各次元の要素(包含サイズ)の値の総和であり，したがって，対応する座標パターンのビットサイズを表わす．

表 1 において，経歴値と座標パターンおよび図 5 の境界ベクトル **B** を参照されたい．

この性質により，経歴値は座標パターンのサイズを表わすヘッダとして使用でき，したがって，経歴値に応じた必要十分なサイズの可変長レコードとしてタプルを取り扱うことができる．

3.2. HPRD

上記の［性質 1］に注目して，タプルを可変長サイズで記憶域に格納する HPRD (History Pattern implementation for Relational Data) と呼ぶ実装方式を提案する．この方式は HPMD よりシンプルであり，＜経歴値, 座標パターン＞を B+木に格納するのではなく，通常の関係データベースの場合のように 2 次記憶上のフラットファイルに挿入順に格納する．キー値以外に構造表現に大きい記憶コストを必要とする B+木を使用しないこと，タプルのエンコード結果に対して，上記の［性質 1］より経歴値をヘッダとして必要十分な座標パターンサイズの記憶域を確保すればよいことにより，HPMD の場合に比べて，さらにコンパクトな実装になっている．生成されたタプルは＜経歴値，座標パターン＞にエンコードされた後, 順に RDF(Real Data File)とよぶ逐次ファイルに格納される(図 3). HPRD の基本データ構造は HPMD における RDT が RDF とよぶ逐次ファイルに格納する以外は HPMD と全く同一である．座標パターンサイズに比べて，その経歴値は小さく，例えば，座標パターンが 64 ビットの規模の大きいタプルでも 6 ビットで済む．RDF における経歴値と座標パターンの記憶域はいずれもバイトアラインメントで割当てられる．

HPMD は多次元データベースの実装に用いられるのに対して，HPRD は関係データベースの通常の実装に用いることができる．したがって，HPRD では属性に対して索引付けしない限り，検索は基本的に全タプルを逐次，探索する必要がある．ただし，経歴・パターン法によるタプルエンコーディングにより，関係テーブルの通常の実装におけるタプル格納ファイルに比べて，RDF は十分に小さく，ファイル I/O のコストを抑制できる. PostgreSQL との比較評価が 4 節で示される．

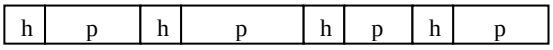


図 3　RDF におけるエンコードタプルの可変長割当て

4. 評価

4.1. 評価環境

実験を行った計算機環境を表 1 に示す．

表 1 HPMD 計算機環境

CPU	Intel Core i7
クロック周波数	2.67GHz
キャッシュサイズ	8MB
主メモリ容量	3GB
OS	CentOS5.5
PostgreSQL	Ver 8.4.4, 64bit 版

実行時間の測定にはファイル入出力にかかる時間を含めるために，また psql における¥timing コマンドは gettimeofday()により実現されており条件の統一のためにこれを用いて行った．いずれの場合も他のユーザプロセスが CPU を使用しない状況にて測定した．

4.2. HPMD, C-HPMD および HPRD

本節では HPMD, C-HPMD および HPRD の比較をするとともに, C-HPMD のランクによる変化を考察する.

本評価においては, 次元数を 5, カージナリティをいずれの次元も 512 とし, 百万件のタプルを多次元空間に一様に分布させたデータを用いた.

4.2.1. 二次記憶サイズ

HPMD と HPRD では実データを格納する RDT や RDF に加え, 各属性のメタ情報, 経歴値テーブルと境界ベクトルテーブル, および属性値テーブルを記録する必要がある. これらのデータの原本は二次記憶上におかれるが, HPMD や HPRD を使用するときには, これらの原本を読み出し, メモリ上にこれらのデータ構造および CVT を構築する.

C-HPMD ではこれらに加えチャンクのタプル登録情報を記録するビットマップも二次記憶に記録され, 使用時にメモリ上にロードされる. このビットマップは空でないチャンクに対してはビット 1 がセットされ, 空のチャンクに対してはビット 0 がセットされ, 空のチャンクの検索を回避するために使われる.

以上のデータ構造を各方式について RDT/RDF とそれ以外に分け, 図 4 に示す. 基本的に RDT/RDF 以外の記憶コストは小さいが, ランク 3 の場合の C-HPMD ではチャンク数が非常に多く, このためにビットマップが極めて大きなサイズを占めている.

HPMD の経歴値は大きな値とならないため 1 バイト, C-HPMD のチャンク番号は大きくなりうるため 8 バイトで格納しており, RDT についてはこれらのサイズに依存したサイズとなっている. RDT に対し B+ 木ではなく単純な構造である RDF は小さく収まっており, HPRD の高い記憶効率を実現している.

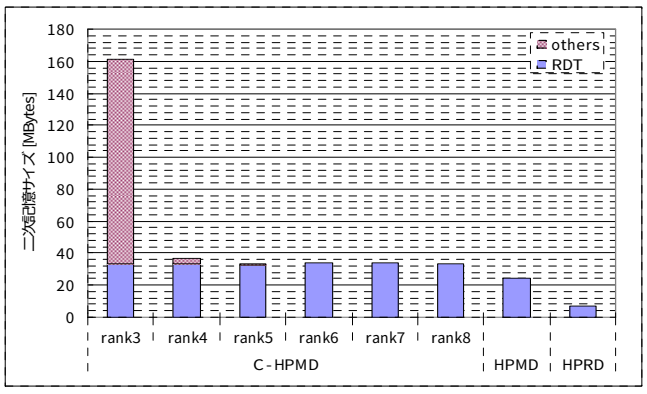


図 4 HPMD, C-HPMD, HPRD の二次記憶サイズ

4.2.2. 構築時間

図 5 より, C-HPMD における構築時間はランクの影響をほとんど受けないことが分かる. また C-HPMD は HPMD より構築に時間を要する. C-HPMD ではチャンク座標とチャンク内座標の 2 段階の計算が必要であるためと考えられる.

HPMD, C-HPMD に比べ HPRD は特に高速である. これも RDF による構造の単純化によるといえる.

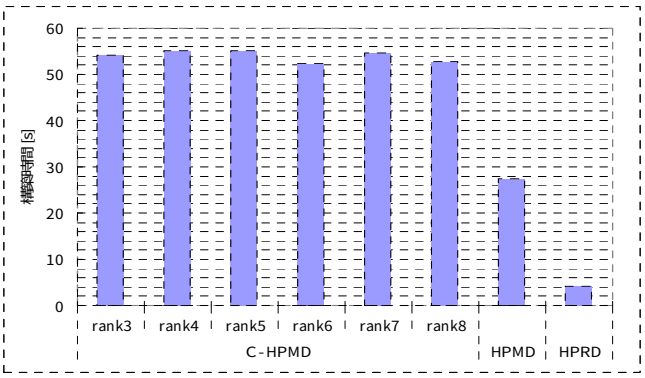


図 5 HPMD, C-HPMD, HPRD の構築時間

4.2.3. 検索時間

1 次元スライス検索の比較を図 6 に示す. C-HPMD における検索コストはランクに大きく依存することが確認できる.

ランク 3 の場合はチャンク総数が非常に多く, このため空チャンクを除く処理に大きく時間を要していると考えられる. 一方ランク 5 ではチャンク総数は減少するが, 空チャンクを除外した上での検索候補チャンクが最も多い. これら候補チャンクすべてに対して RDT にアクセスして検索を行うため, ディスクアクセスの回数が多くなり特に大きく時間を要すると考えられる.

ランク 7 やランク 8 では候補チャンクは少なく, 比較的高速に行えるチャンク内全検索が大部分を占めるために特に高速となると考えられる. 特にランク 7 では HPMD, HPRD よりも高速な検索が可能である.

また, HPRD は HPMD よりも高速となっているが, 二次記憶上のデータの圧縮による効果と考えられる.

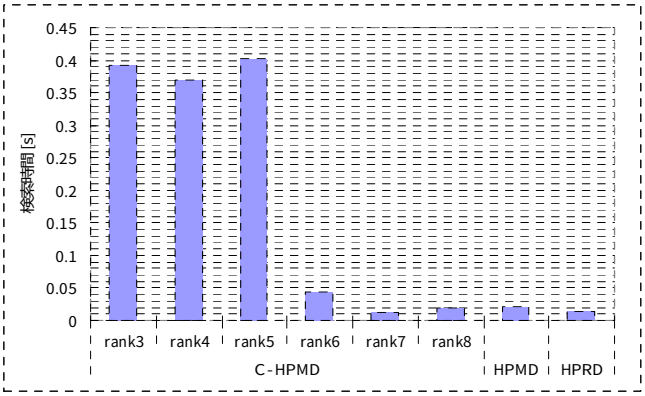


図 6 HPMD, C-HPMD, HPRD の検索時間

4.3. ベンチマーク用データによる HPRD の評価

本節では HPRD の評価を行う. 代表的な関係データベースシステムである PostgreSQL と比較する. 本評価

においては，ベンチマークに用いられる TPC-H[6]の LINEITEM テーブル(表 2)を用いた．ただしタプル数は 23,996,604 とし，L_COMMENT カラムは除いた．

表 2 LINEITEM テーブル

col_name	type	cardinality
L_ORDERKEY	int	6000000
L_PARTKEY	int	800000
L_SUPPKEY	int	40000
L_LINENUMBER	int	7
l_QUANTITY	double	50
L_EXTENDEDPRICE	double	1079204
L_DISCOUNT	double	11
L_TAX	double	9
L_RETURNFLAG	char[1]	3
L_LINESTATUS	char[1]	2
L_SHIPDATE	char[10]	2526
L_COMMITDATE	char[10]	2466
L_RECEIPTDATE	char[10]	2555
L_SHIPINSTRUCT	char[25]	4
L_SHIPMODE	char[10]	7
L_COMMENT	char[44]	15813794

4.3.1. 二次記憶サイズ

二次記憶 HPRD の記憶サイズとしては RDF に加え，テーブルオープン時に必要なメモリ上の HPRD データ構造の再構築に必要な各データを含む．

HPRD においては経歴値とパターンというコンパクトな形にエンコードし，順次格納するというシンプルな格納を行っている．このため図 7 に示すように PostgreSQL(psql と表記)に比して 1/6 程度のサイズに収まっていることが確認できる．

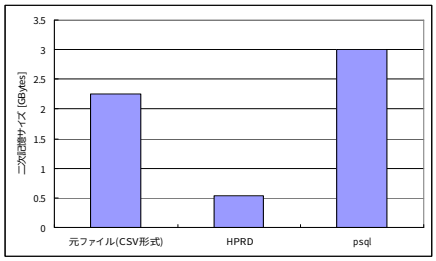


図 7 HPRD および PostgreSQL の二次記憶サイズ

4.3.2. 構築時間

構築に要する時間では HPRD の方が PostgreSQL よりもわずかによいことが図 4.5 より分かる．挿入にコストのかかる B+木を廃した格納構造の単純化による結果であるといえる．

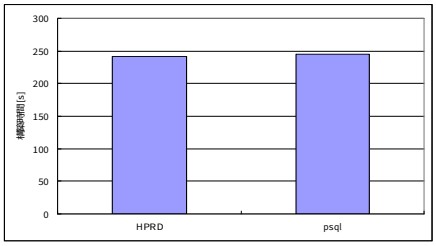
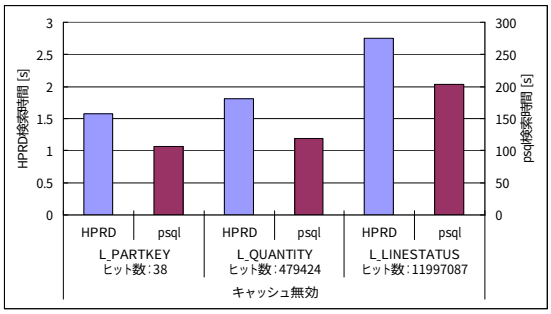


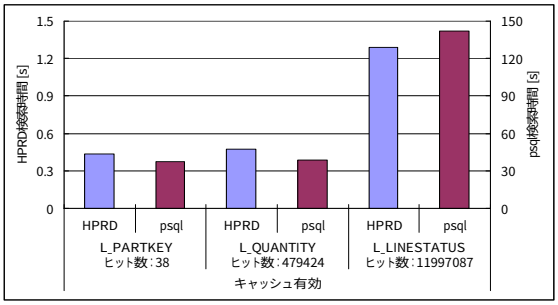
図 8 HPRD および PostgreSQL の構築時間

4.3.3. 検索時間（TPC-H）

1 次元スライス検索に関する評価を行った．この際，実データを格納しているファイルがリードキャッシュに乗っていない状態（キャッシュ無効）と乗っている状態(キャッシュ有効)とで大きく時間が異なるため，両方の場合について測定を行った．



(a) キャッシュ無効



(b) キャッシュ有効

図 9 HPRD と PostgreSQL の検索時間

検索時間は検索するカラムによっても変化するため，代表として L_PARTKEY，L_QUANTYTY および L_LINESTATUS に対しての結果を図 9 に示した．各カラム名とともに検索結果のタプル数を併記した．また HPRD と PostgreSQL の検索時間は大きく異なるため，HPRD は左軸に，PostgreSQL は左軸の 1/100 のスケールとして右軸に分けてグラフ化した．

図 4.6 より HPRD と PostgreSQL ではキャッシュによる検索時間の変化は異なることが見て取れる．キャッシュ無効の場合では HPRD が PostgreSQL に対して約 70 倍，キャッシュ有効の場合では約 100 倍高速な検索を行うことができる．いずれであっても HPRD の検索速度が非常に優れているといえるが，キャッシュが効いている場合の方がより有利であることが分かる．

4.4. 各種データによる HPRD と PostgreSQL の検索時間

二次記憶サイズおよび構築時間については，挿入されるデータの形やタプル数によって，HPRD と PostgreSQL の傾向はさほど変化しない．一方で検索時間は場合によって HPRD と PostgreSQL の比が大きく変化するため，複数の異なるデータを用いて測定した．

用いたデータを表 3 に示す.

表 3 テスト用データ各種

データ名	次元数	タプル数	分布
次元 5 小	5	5,000,000	全次元 512 種一様分布
次元 5 中	5	50,000,000	同上
次元 5 大	5	100,000,000	同上
次元 10 小	10	5,000,000	同上
次元 10 大	10	50,000,000	同上

表 4 に HPRD の RDF および PostgreSQL の記憶サイズを記す.

表 4 記憶サイズの比較

データ名	RDF サイズ [MByte]	PosgreSQL サイズ [MByte]
次元 5 小	33.2	249
次元 5 中	325	2430
次元 5 大	649	4860
次元 10 小	76.3	326
次元 10 大	594	3184

ここでも 4.3.3 節と同様にリードキャッシュの無効有効両場合について測定を行った. 結果を図 10 から図 14 に示すが, いずれも HPRD を左軸, PostgreSQL を右軸にとっている.

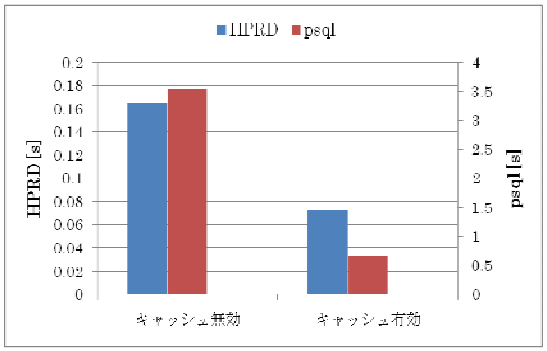


図 10 次元 5 小

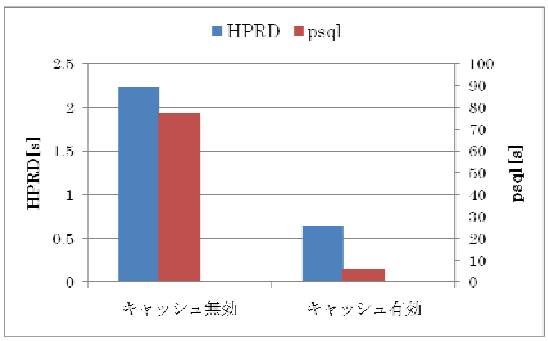


図 11 次元 5 中

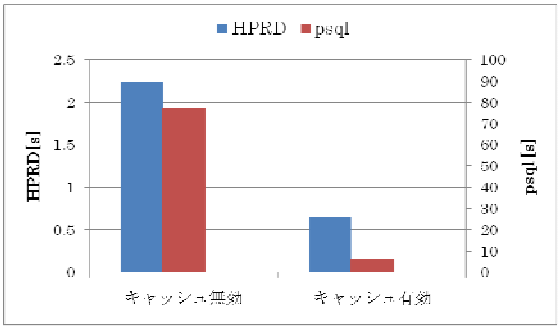


図 12 次元 5 大

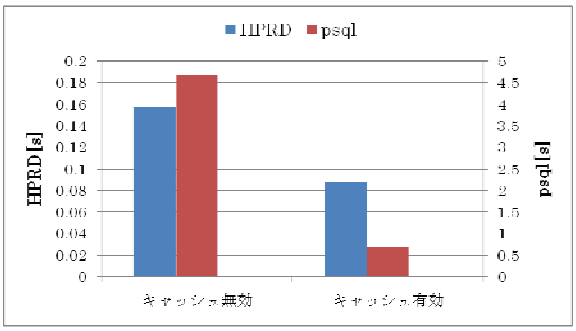


図 13 次元 10 小

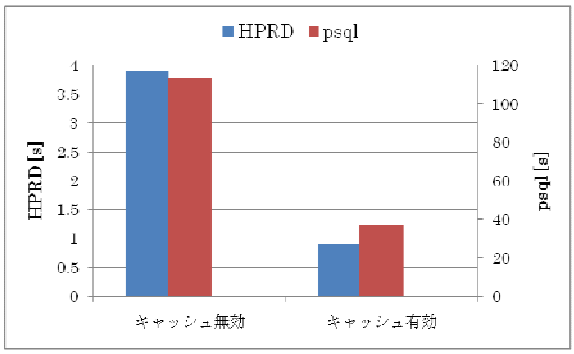


図 14 次元 10 大

図 10 から図 14 より, いずれの場合もリードキャッシュが効いていない状態では HPRD が PostgreSQL に対して約 30 倍高速となっている.

次にリードキャッシュが有効な場合で比較する. 図 10, 図 11 及び図 13 においては HPRD が PostgreSQL に対して約 8 倍から 9 倍高速となっており, キャッシュが無効の場合より比は悪くなっている. 一方で図 12 と図 14 では HPRD が PostgreSQL に対して 40 倍程度高速で, この場合はキャッシュが利いている方が比がよい. これは PostgreSQL においてはリレーションサイズが大きい場合にリードキャッシュの効果が薄くなっているためと見られる. 図 11 と表 5 及び図 14 と表 8 の比較により, PostgreSQL のリレーションサイズが主メモリ容量の約 3GB を超えるときはキャッシュが有効

な場合に HPRD がより有利に検索を行えると考えられる．またこの節において行った一様分布させるように人工的に生成したデータでの測定と比べ，4.3.3 節にて行った TPC-H での測定の方が，HPRD が PostgreSQL に対してより高速に検索できるという結果になっている．

5. 関連研究

多次元データの実現方式について，現在まで，種々のアプローチにより膨大な研究がなされているが，ここでは，まず，本研究におけるような拡張可能配列に基づいた研究について，2,3 紹介する．[2]ではハッシュ技法に基づいた，また，[3]ではインデックス配列に基づいた拡張可能配列の実現方式を提案している．筆者等は拡張可能配列について[3]の時間・空間コストの改善方式を提案している[6]，[3]~[6] 等の研究は，高密度の拡張可能配列の二次記憶装置上の表現を論じているが，経歴オフセット法[8]や経歴パターン法[10]のような 2 項組のタプルエンコードや圧縮の考えには至っていない．また，[7]では拡張可能ハッシュ技法[9]を[3]の拡張可能配列に基づいて多次元キーに適用している．しかし，経歴パターン法とはアプローチが異なり，添字の連接パターンの考え方や境界ベクトルによる圧縮エンコードやデコードの考えはない．経歴・オフセット法の応用として，[11]では，データキューブの差分構築方式を，また，[13]では XML 木のラベル付けの方式を提案している．

多次元データの圧縮格納方式としては，大規模多次元配列を扱う MOLAP では，疎配列問題に対して chunk-offset 圧縮[1]と呼ぶ方法が，多く用いられるが，各次元サイズが固定であるために，新たなカラム値の出現に対して，柔軟に対処できない．

6. おわりに

経歴・パターンエンコード方式に基づく多次元データの実装について述べ，その性能を比較評価した．その結果，PostgreSQL と比較して記憶コストと検索コストの両面で高い性能を示した． この性能は，境界ベクトルによる座標パターンの圧縮により，データ本体サイズがかなり小さく押さえられていること，および，シフト演算とマスク演算のみでタプルのエンコード，デコードが行えることに大きく依存しているといえる．また，データ本体以外の必要な補助データ構造の記憶コストが本体サイズに比べて，一般にはかなり小さいことも注目される．今後は検索機能の充実と種々の条件での本方式の評価を行いたい．

参 考 文 献

[1] Zhao, Y. , Deshpande, P. M. and Naughton, J. F. : An array based algorithm for simultaneous multidimensional aggregate, Proc of the ACM SIGMOD Conference, pp. 159-170, 1997.

[2] A.L.Rosenberg, L.J.Stockmeyer, "Hashing Schemes for Extendible Array's", JACM, Vol.24, pp.199-121,1977.

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

[4] D. Rotem, J. L. Zhao, “Extendible Arrays for Statistical Databases and OLAP Applications”, Proc. of SSDBM1996, 1996.

[5] 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.

[6] E. J. Otoo, D. Rotem, A Storage Scheme for multidimensional databases using extendible files, Proc. of STDBM2006, pp.67-76, 2006.

[7] E. J. Otoo,”A Mapping Function for the Directory of a Multidimensional Extendible Hashing”, Proc. of VLDB1984, pp.493-506, 1984.

[8] K. M. A. Hasan, T. Tsuji, K. Higuchi, “An Efficient Implementation for MOLAP Basic Data Structure and Its Evaluation”, Proc . of DASFAA2007, pp. 288-299, 2007.

[9] R. Fagin, J. Nievergelt, N. Pippenger, Strong H. R. Strong, “Extendible Hashing: A Fast Access Method for Dynamic Files”, ACM. Trans. on Database Syst.,Vol.4, No. 3, pp. 315-344, 1979.

[10] 都司達夫，水野広治，松本宗純，樋口健，"動的多次元データセットのコンパクトな実現方式の提案"，日本データベース学会論文誌 Vol.8, No.3, pp.1-6, 2009.

[11] D. Jin, T. Tsuji, T. Tsuchida, K. Higuch, “An Incremental Maintenance Scheme of Data Cubes”, Proc. of DASFAA 2008, pp.172-187, 2008.

[12] R. Zhang, P. Kalnis, B.C. Ooi, K.L. Tan, “Generalized multidimensional data mapping and query processing”, ACM Trans. on Database Syst., pp. 661-697, 2005.

[13] B. Li, K. Kawaguchi, T. Tsuji, K. Higuchi, “A Labeling Scheme for Dynamic XML Trees Based on History-offset Encoding”, 情報処理学会論文誌: データベース, Vol.3, No.1, pp.1-17, 2010.

[14] http://www.tpc.org/tpch/