データ圧縮型インデックスVA-TREEの検討

吉田 忠城 小西 史和 赤間 浩樹 山室 雅司
NTT サイバースペース研究所

本稿は，大量な多次元ベクトルデータの高速検索を目的として，ノードの矩形情報を圧縮して保持する動的多次元空間インデックス VA-TREE について述べている．画像検索システムに用いられている色彩(Hue) 情報 16 次元ベクトルの実データに対する評価実験の結果，VA-TREE は VA-File と比較して約 75%の距離計算回数の削減とディスクへのランダムアクセスとなる約 99%の実ベクトルアクセス回数の削減が可能となった．

Similarity Search Index using Vector Approximation VA-TREE

Tadashiro Yoshida, Fumikazu Konishi, Hiroki Akama, and Masashi Yamamuro
NTT Cyber Space Laboratories

This paper describes the similarity search index VA-TREE, which uses the vector approximation method to represent an internal node and a leaf node in a tree structured index. Our experiment, using real 16-dimension hue data which is applied in many image retrieval applications, shows the advantage over the flat structured vector approximation index VA-File.

1. はじめに

デジタルカメラや音声高圧縮方式 MP3 を用いた録音装置に代表されるように，近年，マルチメディアデータのディジタル化が急速に進んでおり，WWW などを含む広い意味でのマルチメディアデータベースが大規模化している．蓄積されたデータを有効利用するために，従来のテキスト情報に加え，これらのマルチメディアデータに対して検索を行いたいという要求は必然的であると言える．

マルチメディアデータの検索において，キーワード付与の人的コストの削減，およびキーワードの主観性排除を目的として，テキスト情報によるキーワードを用いるのではなく，各メディアから直接抽出される特徴量そのものを用いて検索を行う方式[1,2,3]が盛んに研究されている．このような検索方式では，一般に特徴量を多次元ベクトルとして表現しデータ間の類似度としてベクトル間の距離を用いているが，データの大容量化に加え，特徴量を表現するベクトル自体の高次元化，およびそれに伴う距離計算量の増加が検索性能に大きな影響を与えている．

従って，高次元大容量ベクトルデータに対して高速な検索を行うことがマルチメディアデータベースを実現するうえでの大きな課題の一つになっている．

この課題を解決するために様々なインデックス方式[4-9]が提案されている．その多くのものはデータ空間（またはデータ集合）をある基準に従って分割し，部分空間（部分集合）を木構造により管理する．木構造インデックスは，特徴量を表現するベクトルデータである実ベクトル，実ベクトルへのポインタを 1 つ以上格納するリーフノード，複数個のリーフノードおよび中間ノードを格納する中間ノード，木構造の頂点に位置するルートノードより構成される(図 1)．検索時には上位階層より，常に検索キーからより近い中間ノードを優先的に掘り下げることにより，不必要な距離計算を省略（枝刈り）して検索効率を上げている．

しかしながら，高次元ベクトルデータにおける木構造インデックスではインデックス容量が大きくなるために，インデックスのすべてをメモリ上に実現することは極めて困難であり一部またはすべてをディスク上に配置することが必要になる．

従って，ディスク I/O と距離計算量の両方を考慮したインデックス方式が重要となる．

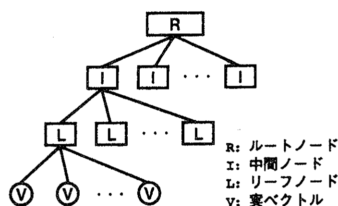


図1 木構造インデックス

本稿では(1)実ベクトルをディスク上に配置し，(2)リーフノードおよび中間ノードの矩形情報を圧縮してメモリ上に配置する，動的な木構造インデックス VA-TREE について検討した結果について述べる．画像検索に用いられている 16 次元の色彩(Hue)ベクトル 10 万件の実データに対する評価実験の結果，同じ圧縮型のインデックスでありフラットな構造を持つ VA-File と比較して，各次元軸を 4 ビットに圧縮するとき，VA-TREE は約 2 倍のインデックス容量を必要とするものの，75%の距離計算回数の削減とディスクへのランダムアクセスとなる 99%の実ベクトルアクセス回数の削減が可能となりその有効性を確認した．

以下，第 2 章においてインデックス方式における関連研究を，第 3 章において提案する VA-TREE について述べる．第 4 章において複数のパラメータによる評価実験結果について述べ，第 5 章において本稿のまとめを述べる．

2. 関連研究

木構造インデックスにはベクトル間の相対距離を用いて分割を行う距離空間インデックス[4]と，多次元ベクトルの値そのものを用いて分割を行う多次元空間インデックスがあるが，本章では VA-TREE と関連の深い多次元空間インデックスについて述べる．また，ベクトル圧縮を用いたインデックスについて述べる．

2.1. 多次元空間インデックス

多次元空間インデックスは，R-tree や k-d-tree に代表されるような低次元空間において有効性を示すインデックス手法をベースにしているものが多く，基本的に分割時に任意の 1 軸について垂直分割していく方式であり，分割方法（分割軸の選択と分割位置）および中間ノードの構成情報(部分空間の形状）に特徴がある．

VAMSplit R-tree[5]は，各軸毎にデータの分散を計算し最大分散値をもつ軸においてほぼデータを 2 分する位置で分割する方式である．部分空間は多次元矩形により管理される．データの動的な追加には対応していないがインデックスの構築時間が速く，高次元における高速な検索方式としてその有効性が報告されている．SS-tree[6]は，データの動的更新に対応したインデックスである．分散が最大となる軸において，分割時に分割対象空間のデータを再投入し分割後の分散の総和が最大になる位置で分割するが，部分空間を多次元矩形ではなく多次元球で管理するところに大きな特徴がある．

一方，SR-tree[7]は，中間ノードを多次元矩形と多次元球の両方を用いて管理し，検索キーとの距離を両者のより小さい方を採用することにより効率的に検索を行っている．

これらのインデックスは分布に偏りがあるデータについて有効性を示しているが，実際のアプリケーションではデータ分布に偏りがある場合が多いと考えられるため適用範囲は広いと考えられる．

2.2. ベクトル圧縮

ベクトルデータが一様に分布している高次元空間では，木構造インデックスが効果的に機能しないという報告がある[7,8]．これは検索時の枝刈りが効かずに多くのノードおよび実ベクトルと距離計算を行う必要があることを意味しているが，特に中間ノードがディスクに配置されるディスクベースのインデックスではディスクアクセスがランダムになることから大きな問題となる．

VA-File[8]は，他の多次元空間インデックスのように階層的な木構造を用いてデータを管理するのではなく，全体空間の各次元軸を等分割してメッシュ状の多次元矩形（セル）を構築し，1 階層のフラットな構造で管理するディスクベースのインデックスである．VA-File では，実ベクトルを含む各セルをビット列で表現（ベクトル近似）し，これを実ベクトルの圧縮表現としてファイル(VA File)に格納する．VA-File での検索は全件検索を基本としており，圧縮されたビット列（すなわちセル）での枝刈りと実ベクトルのアクセスによる枝刈りを組み合わせることにより検索効率を上げている．また，VA-File では，ベクトル圧縮によりディスクページあたりの格納効率を上げてディスク I/O を抑えるとともに，圧縮ベクトルのディスクへのアクセスをシーケンシャルアクセスにす

ることにより，一様分布である高次元データに対して優位性を示している．

一方，部分空間符号化法[9]は，実ベクトルデータを用いて構築される R-tree などの木構造インデックス(Real Part)の各最小包囲領域を，VA File と同様にメッシュ状に分割したセルを表現するビット列を用いて最小包囲領域を包含する仮想包囲領域として表現し，仮想包囲領域から構成される木構造 Virtual Part を構築する手法である．ディスクベースの木構造インデックスにおいて，ノードの矩形情報を圧縮することによりページあたりの格納効率を上げ，ディスクアクセス回数の大幅な削減を実現している．

以上のように，ベクトル圧縮法はディスクベースのインデックスにおいて有効性を示しているが，ディスク I/O コストは非常に高いため，メモリを併用するインデックスにおいてもインデックスのできるだけ多くの部分をメモリ上に構築する上で重要な手法であると言える．

3. VA-TREE

本稿で提案する圧縮型インデックス VA-TREE は，VA-File のセル構築法（空間分割法）およびベクトル圧縮法を階層構造に拡張することにより，分布に偏りのあるデータに有効である木構造インデックスの長所と，インデックスのより多くの部分をメモリ上に保持することを可能とするベクトル圧縮法の長所をあわせもつインデックスであり以下の特徴がある．

(1) ノードの矩形情報のベクトル圧縮
実ベクトルをディスク上に配置し，ベクトル圧縮により中間ノードおよびリーフノードをメモリ上に配置する．これによりディスク I/O を抑えることが可能となる．

(2) 全次元軸を用いての分割
偏りのあるデータ空間から一様分布空間まで対応が可能である．特に偏りのあるデータ空間については，距離計算回数の削減が期待できる．

(3) 軸毎に全階層において同分割数
各階層におけるノードのビット列表現は上位ノード内の相対位置を用いるが，多次元矩形の絶対位置情報より算出することが可能であるため，ビット列構築に必要な部分的な計算結果（例えば，矩形辺長）を予め求めておくことや複数の次元軸で共有が可能である．従って，ベクトル圧縮のための計算オーバーヘッドを抑えることができる．また，構造が単純であり並列化を容易に行うことが可能である．

3.1. 構造

任意の階層におけるベクトル圧縮の考え方は VA-File をベースにしている．すなわち，d 次元のベクトルデータ p_i ($1 \leq i \leq N$: N はデータ総数) を次元軸 j を b_j ($1 \leq j \leq d$) ビットで表現する総ビット長 b ($b = \sum_{j=1}^{d} b_j$) を用いて表現するとき，b_j は以下のとおりである．

$$b_j = \left\lfloor \frac{b}{d} \right\rfloor + \begin{cases} 1 & \text{if } j \leq (b \bmod d) \\ 0 & \text{otherwise} \end{cases}$$

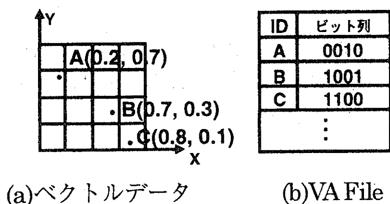
また，p_i が次元軸 j において区間[0, w_j] ($1 \leq j \leq d$)に存在し，p_i の次元軸 j における座標値を p_{ij} とするとき，p_{ij} の次元軸 j のビット列表現は w_j を 2^{b_j} 分割したときの区間番号 m_{ij}($0 \leq m_{ij} \leq 2^{b_j}-1$) と等価になる．すなわち，

$$m_{ij} = \left\lfloor \frac{p_{ij}}{w_j / 2^{b_j}} \right\rfloor - \begin{cases} 1 & \text{if } p_{ij} = w_j \\ 0 & \text{otherwise} \end{cases}$$

この m_{ij} を全次元軸についてビット列として表現しさらに連結したものが p_i のベクトル圧縮表現となる．

図 2 に VA File の模式図を示す．例えば，各次元が区間[0,1]空間に存在する 2 次元データに対し（図 2(a)），各次元軸を 2 ビット(b_j =2)（すなわち，各軸 4 分割）でベクトル近似するとき，データ A は X 軸については区間 0 に存在するためにビット列は"00"，Y 軸については区間 3 に存在するためにビット列が"10"となり，データ A (およびデータ A が存在するセル) を示すビット列は"0010"となる．VA-File では，以上のようにベクトル近似された実ベクトルのビット列をファイル(VA File)に格納し（図 2(b)），このビット列に対して全件検索を行うことにより枝刈りを行う．

一方，VA-TREE では，ベクトル圧縮されたビット列が示すセルを木構造インデックスの



(a)ベクトルデータ　　　(b)VA File

図 2 VA-File の構造

MBR (Minimum Bounding Rectangle)とみなし，任意の階層において同じビット列表現を持つ実ベクトルが閾値(δ)個以上存在するとき（すなわち，同じセルにδ個以上の実ベクトルが存在するとき）に，そのセルを新たな全体空間とみなして繰り返し分割することにより木構造へ拡張する．

各階層におけるビット列表現は，以下の理由により上位ノード内の相対位置を用いる．

(1) セル表現の必要ビット数の削減
相対位置表現とすることで全階層において同じビット数とすることや，任意の階層において必要なビット数を最小化することが可能となる．

(2) スケーラビリティのある動的追加への対応
固定的なビット列表現は，データの総数および偏りに応じて軸分割数を動的に変更する必要が生じるため木構造全体への波及が大きいが，上位ノードの相対位置表現とすることで，軸分割が局所的になりスケーラビリティが向上する．

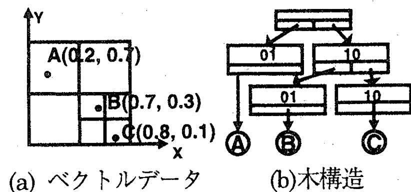
従って，実ベクトル p_i が次元軸 j において区間 $[0, w_j]$ $(1 \leqq j \leqq d)$に存在するような全体空間の $w_j (1 \leqq j \leqq d)$が与えられるとき，任意の階層 l における p_i の次元軸 j における区間番号を m_{ij}^{l} は以下の式で求められる．

$$m_{ij}^{l} = \left\lfloor \frac{p_{ij} - cs_j^{l-1}}{w_j^{l}} \right\rfloor - \begin{cases} 1 & \text{if } p_{ij} = w_j^{0} \\ 0 & \text{otherwise} \end{cases}$$

但し　$w_j^{l} = w_j^{l-1} / 2$

$cs_j^{l-1} = w_j^{l-1} * m_{ij}^{l-1}$

例えば各次元が区間[0,1]空間に存在する 2 次元データに対し（図 3(a)），各次元軸を 1 ビット(b_j =1) でベクトル圧縮するとき，第 1 階層におけるベクトル圧縮されたビット列はそれぞれ A(01),B(10),C(10)である．δ=1 とすると B,C は同一のセルに存在し同じビット列表現になるため



(a) ベクトルデータ　　　(b)木構造

図 3 VA-TREE の構造

分割が生じる．このとき，第 2 階層においては X 軸が区間[0.5,1]，Y 軸が[0,0.5]であるセルを全体空間とみなして，B,C の各軸の値を補正し，第 1 階層と同様にベクトル圧縮を行う．この結果，第 2 階層における B,C のベクトル圧縮表現は B(01),C(10)となる．以上のように構築される木構造を模式的に表わしたものが図 3(b)である．

なお，上位ノードの相対位置を用いるという各次元軸におけるビット列生成法の考え方は基本的に部分空間符号化法[9]と同じであるが，VA-TREE のすべてのノードは，メッシュ状に分割された上位ノードにおける 2 つのセルの位置情報を用いて表現されるのではなく，必ず 1 つのセル自身の位置情報として表現される．また，VA-TREE では次節に示すように木構造の作成の際に Real Part を必要とせずに直接ビット列を用いた木構造を作成することが可能である．

3.2. 構築処理

図 4 に構築アルゴリズムを示す．ステップ 2 において，ビット列生成の基準となる分割情報(各次元軸における区間長) を生成する．このとき，階層 level における vaVector は上位の部分空間（level=1 の場合は全体空間）における相対的な位置を示すビット表現となるが，VA-TREE における MBR はすべての階層において軸毎に同数に等分割された空間であるため，任意の階層における任意の軸の区間長は階層の深さと次元軸により決定される．このため，上位ノードを参照しなくても事前に計算しておくことが可能（ステップ 2）であり高速化を図ることが可能となる．

ステップ 4 において階層 level における挿入ベクトルデータ(insertVector)を圧縮表現しビット列 vaVector を構築する．

ステップ 5 では，ステップ 4 で求められたベクトル圧縮 vaVector がノード vaNode に存在するかを判定する．存在しない場合には，vaNode に追加して終了する（ステップ 7）．

一方，vaNode に vaVector が存在する場合に

```
    Procedure StoreVectorToVANode( vaNode, insertVector, level )
 1:     if level > deepestLevel then
 2:        MakeDivisionInfo( TOTALWIDTH, level);
 3:     end;
 4:     vaVector = MakeBitVector( insertVector, level );
 5:     SearchInVANode( vaNode, vaVector );
 6:     if NOT found then
 7:        appendVectorToVANode( vaVector, level );
 8:     else
 9:       if foundVAVector.type = VANODE then
10:          StoreVectorToVANode( childVANode, insertVector, level+1);
11:       else
12:          MakeChildVANode( level+1);
13:          StoreVectorToVANode( childVANode, insertVector, level+1);
14:       end;
15:     end;
16: end.
```

図4 構築アルゴリズム

は，そのノードタイプを判定する．ノードタイプがVANODEであるときには，さらに下位にノード（中間ノード）を持つことを意味し，LEAFであるときには実ベクトルをポイントすることを意味する（リーフノード）．ノードタイプがVANODEのときには下位ノードに挿入するために，StoreVectorToVANode()を再帰的に実行する（ステップ10）．また，ノードタイプがLEAFであるときには，新しく下位ノードを作成し，既に登録されていた実ベクトルを下位ノードに挿入し（ステップ12），挿入ベクトルデータについても下位ノードに挿入する（ステップ13）．

なお，図4では，説明を簡略化するために複数の実ベクトルを含むLEAFの処理や同一ベクトルの管理を省略している．

3.3. 探索処理

探索処理では，検索キーベクトルと各ノードとの距離を基に，下位方向に探索範囲を掘り下げる対象ノードに優先度を持たせることにより枝刈りを行う[10]．ベクトル圧縮を導入したことによる特徴的な点は，ノード（中間ノードおよびリーフノード）の復元と，実ベクトルへのアクセスを削減するためにノードを掘り下げるときに得られる下位ノードがリーフノードである場合にも一度候補ノードリストに挿入する点である．

図5に探索アルゴリズムを示す．探索処理を行うために，各々ソートされた候補ノードを管理するCandidateNodeList，および最終結果リストを管理するResultListを用いる． ステップ1において，ルートノードをCandidateNodeListに設定する．ステップ3では，CandidateNodeListより先頭のノードを取り出し，ノード内に含まれるitem（中間ノードまたはリーフノード）を抽出する．ステップ3において抽出したitemのタイプが中間ノード（VANODE）である場合には，さらに下位ノード(childItem)を抽出し（ステップ5），抽出したすべての下位ノードについてセルを復元し（ステップ7），検索キーとの距離を計算して（ステップ8），候補ノード用リストへ挿入する（ステップ9）．

```
    Procedure SearchVATree( vaTreeRoot, queryVector, returnNum )
 1:     insertCandidateNodeList( vaTreeRoot );
 2:     while ( CandidateNodeList not empty ) do
 3:       item = popCandidateNodeList();
 4:       if  item.type = VANODE then
 5:         childItemList = getChildItemList( item );
 6:         for each childItem in childItemList do
 7:           childItem.MBR = rebuildMBR( childItem );
 8:           childItem.nearestDistance = calcNearestDistance( queryVector, childItem.MBR );
 9:           insertCandidateNodeList( childItem );
10:         end;
11:      else
12:         if ResultListKthNearestDistance > item.nearestDistance then
13:           item.vector = readFromDisk( item );
14:           item.nearestDistance = calcRealNearestDistance( queryVector, item.vector );
15:           if ResultListKthNearestDistance > item.nearestDistance then
16:              insertResultList( item );
17:              alterResultListNearestKthNearestDistance();
18:           end;
19:         end;
20:      end;
21:    end;
22: end.
```

図5 探索アルゴリズム

一方，ステップ3において抽出されるitemがリーフノードである場合には，既に計算されている検索キーとMBRとの距離が最終結果リストのk番目（kは検索結果の返却数）の距離より小さいときにのみ，ディスクに格納されている実ベクトルを読み出し（ステップ13），実距離を計算する（ステップ14）．その後，再度，最終結果リストのk番目の距離と比較を行い，距離が小さいときのみ最終結果候補リストに挿入し（ステップ16），最終結果候補リストのk番目の距離を更新する（ステップ17）．

4. 実験

VA-TREEをSUN Ultra60(UltraSPARC-II 450M)上に実装し，k近傍検索実験（kを20に設定）を行った．木構造決定の主要要因は，リーフノード内に格納する最大データ個数(δ)および圧縮ベクトルを表現するためのビット数である．実験では，これらのパラメータとデータ総数の変化に対して，構築処理についてはインデックス容量および構築のためのCPU時間の観点から，探索処理についてはノードおよび実ベクトルとの距離計算回数，ディスク上に存在する実ベクトルの読み出し回数の観点から評価を行った．また，VA-Fileについても実ベクトルをディスク上に配置しVA Fileをメモリ上に配置する構成で実装を行いVA-TREEとの比較を行った．

4.1. 実験データ

実験に用いたデータは，風景などを収めた日常写真から抽出されるオブジェクト画像[1]の色特徴量の一つである色彩(Hue)の 16 次元ベクトルデータである．多次元空間インデックスにおいてはデータ分布が重要な意味を持つ．図 6 に実験に用いたベクトルデータの分布を示す．

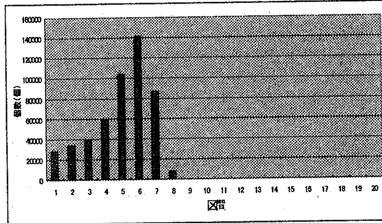


図6 データ分布

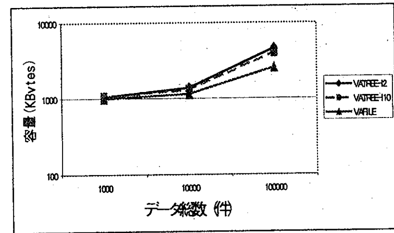
図6は10万件のデータからランダムに抽出したデータ1000件についてデータ相互間の距離(ユークリッド距離）を測定し，すべてのデータが存在する全体空間の最大距離(16 次元の[0,1]空間であることからの理論値 4.0)を 20 分割したときの距離分布を示している．図 6 より実験に用いたデータの分布の偏りが比較的大きいことがわかる．

4.2. 構築処理の評価

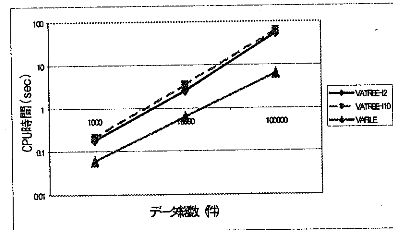
圧縮ベクトルの表現ビット数を 64 ($b_j = 4,\ 1 \leqq j \leqq 16$) に固定し，リーフノード内の最大データ個数($\delta$)を変化させたときのインデックス容量を図 7(a)に，構築のための CPU 時間を図 7(b)に示す．インデックス容量は top コマンドにより取得した値である．

図 7(a)(b)において，suffix の l2,l10 はそれぞれδが 2,10 であることを示す．また，vafile は VA-File における測定結果である．

一方，リーフノード内の最大データ個数を 2 に固定し，ベクトル圧縮表現のためのビット数を変化させたときのインデックス容量を図8(a)に，および構築のための CPU 時間を図 8(b)に示す．図 8(a)(b)において suffix の b64,b128 はそれぞれベクトルの圧縮表現に用いたビット数が 64/128 ビットであることを示す．また，VAFILE-b64 はビット数が 64 ビットである VA-File を示しており，図 8(b)における 128 ビットの VA-File については 64 ビットと同一値であったため表示を省略している．

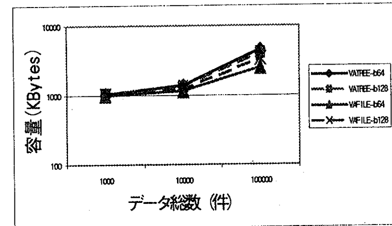


(a) インデックス容量

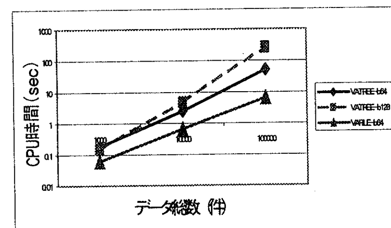


(b) 構築時間(CPU Time)

図7 リーフノード内最大個数の変化（構築）



(a) インデックス容量



(b) 構築時間(CPU Time)

図8 表現ビットの変化（構築）

インデックス容量は，図 7(a)，図 8(a)に示すように，VA-File も含めていずれのパラメータにおいてもデータ総数の増加に対して比例的に増加していないが，2 つの原因が考えられる．

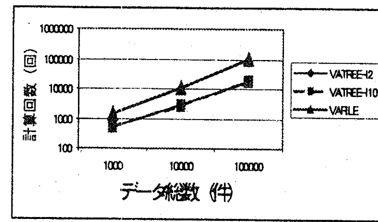
一つは動的インデックスのためのオーバーヘッドである．VA-TREE は動的インデックスであるために，ノード内のデータが増えるたびに領域の拡張を行うが，この管理のオーバーヘッドが影響している．(VA-File についても動的追加に対

応した実装を行っている.)

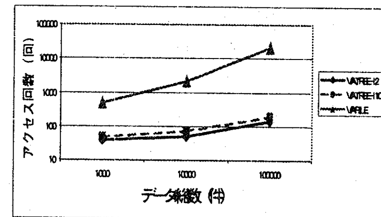
もう一つは，VA-TREE における木の構築アルゴリズム自体の問題であると考えられる．インデックスの容量はノード数に大きく依存する．VA-TREE における分割法は 2^k 空間分割法に類似しているが，2^k 空間分割法ではデータ分布の偏りに対して木のバランスが悪くなりノード数が多くなることが知られている．また，木のバランスが悪いことは各ノードの充填率が小さいことを意味しており，無駄な領域が多くなっているためと考えられる．

VA-TREE の構築のための CPU 時間はデータ数の増加に対応してほぼ比例的に増加している．これは実装に大きく依存しているところがある．任意のノードへのデータ挿入時には，圧縮されたビット列表現がそのノードに既に既に登録されているかを判定する必要があるが，現在の実装ではシーケンシャルに比較を行っている．特に vatree-b128（図 8(b)参照）では，1 階層における分割数が多く，これは VA-TREE では枝数が多くなることを意味している．従って登録有無判定のためのビット列比較回数（図 4 ステップ 5）が増加する．この問題については登録されているビット列に対して B-tree などを採用することによる時間削減が期待できる．

また，図 7 においてリーフノード内の最大データ個数が多い方が容量が小さいにも関わらず逆に構築時間が多くなっているが，これは同一ベクトルの管理のために，リーフノード内の各実ベクトルをディスクより読み出しその一致性を判定しているためである．同様に図 8 においても容量と CPU 時間の関係に逆転が見られるが，これはノードにおける登録有無の判定処理が原因であると考えられる．

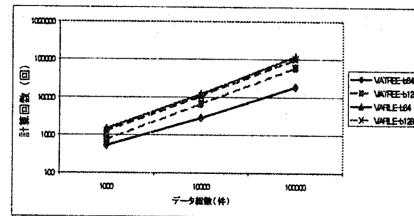


(a) 距離計算回数

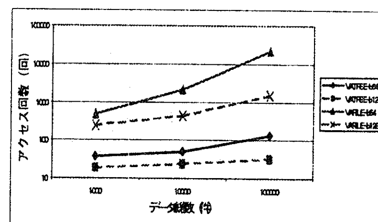


(b) 実ベクトルアクセス回数

図 9 リーフノード内最大個数の変化（探索）



(a) 距離計算回数



(b) 実ベクトルアクセス回数

図 10 表現ビットの変化（探索）

4.3. 探索処理の評価

図 9(a)(b)，図 10(a)(b)にそれぞれの構築パラメータに対応して，ノードとの距離計算回数，ディスク上の実ベクトルへのアクセス回数を示す．測定結果はランダムに選択した検索キー 100 回の検索における平均である．なお，距離計算回数には実ベクトルとの距離計算回数を含んでいる．

図 9(a)に示すように，距離計算回数はリーフノード内の最大データ個数には影響を受けていない．しかしながら，図 9(b)に示すように実ベクトルのアクセス回数に違いが見られる．特にデータ総数 10 万件では，VATREE-l2 は VATREE-l10 と比較して約 40%アクセス回数が減少している．これはリーフノード内の最大データ個数が小さくなると，階層がより深くなりリーフノードの MBR が小さくなり枝刈りがより有効に機能したためである．

一方，図 10(a)(b)に示すようにベクトル圧縮表現のためのビット数の違いはより顕著に現れている．10 万件データにおける VATREE-b64 と VATREE-b128 では（圧縮率が 2 倍），距離計算回数が VATREE-b128 が VATREE-b64 より約 3 倍多いが，逆に実ベクトルのアクセス回数は VATREE-b64 が VATREE-b128 より約 4 倍多い．

これは，ベクトル圧縮した中間ノードおよびリーフノードにおいてできるだけ距離計算を行い枝刈りを行うことでディスクアクセス回数を抑えるVA-TREE の性質が出ているが，VATREE-b128 の方がより小さなリーフノードを構成できるために枝刈りが有効に機能しているためと考えられる．しかしながら，128 ビットを用いた分割では，1 階層における分割数が多く，枝数が多くなり距離計算の回数の増加をもたらしている．

また，データ分布の偏りが影響して VA File では枝刈りが有効に機能しておらずいずれのパラメータにおいても距離計算回数，実ベクトルへのアクセス回数とも VA-TREE より多くの回数を必要としている．例えば，ベクトル圧縮表現のためのビット数が 64 ビットのとき，リーフノード内の最大データ個数 2 個の VA-TREE(VATREE-b64)と VA-File(VAFILE-b64)とを比較すると，VA-TREE が VA-File に比べ約 75%の距離計算回数の削減が，また，約 99%以上の実ベクトルアクセス回数の削減が実現されており，VA-TREE の有効性を確認することができる．

5. おわりに

大容量のマルチメディアデータベースに必須な高速インデックスを実現するためには，ディスク I/O の削減と，ベクトル距離計算回数の削減が重要である．

本稿では，実ベクトルをディスク上に配置し，木構造の中間ノードおよびリーフノードの矩形情報を圧縮してメモリ上に配置する動的多次元空間インデックス VA-TREE について提案した．

VA-TREE を実装し，多くの画像検索システムで用いられている色彩情報(Hue)を用いた実データについて評価実験を行った結果，ベクトル圧縮したデータをフラットな構造で管理する VA-File と比較して，各次元軸 4 ビットに圧縮するとき，VA-TREE は約 2 倍のインデックス容量を必要とするものの，75%の距離計算回数の削減とディスクへのランダムアクセスとなる 99%の実ベクトルアクセス回数の削減が可能となり本手法の有効性を確認することができた．

高次元空間インデックスにおいては，データ量，次元数，およびデータ分布が重要な意味を持つ．今後異なる次元数やデータ分布，1000 万オーダの大量データについて検証を行う必要がある．データの高次元化については特に重要な問題であると考える．データが高次元化すると 1 階層における分割セル数が多くなるため，データ分布の偏りがなくなるにつれ構造が VA File に近づき距離計算回数の削減が期待できなくなるためである．VA-TREE は構造が単純であるため並列化が一つの解となると考える．

[参考文献]

[1] 串間，赤間，紺谷，山室：“色や形状等の表層的実にもとづく画像内容検索技術”，情報処理学会論文誌：データベース，Vol.40, No.SIG3(TOD1)，pp.171-184.

[2] 西原，梅田，紺谷，山室，福本：“大規模音楽 DB に対するハミング検索方式”，アドバンスト・データベースシステムシンポジウム 98，pp.117-124，1998.

[3] M.Flickner, H.Sawhney, : “Query by Image and Video Content: The QBIC System”, IEEE Comuputer, pp.23-32, 1995.9.

[4] T.Bozkaya, and M.Ozspyoglu: “Distance-Based Indexing for High-Dimensional Metric Spaces”, SIGMOD '97, pp.357-368.

[5] D.A.White, and R.Jain: ”Similarity Indexing: Algorithms and Performance”, In Proceedings of the SPIE: Storage Retrieval for Image and Video Database Ⅳ, 1996 San Jose, CA, Vol.2670, pp.62-75, 1996.

[6] D.A.White, and R.Jain: “Similarity Indexing with SS-tree”, Proc. of the 12th Int. Conf. on Data engineering, pp.516-523, 1996.

[7] N.Katayama, and S.Satoh: “The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries”, Proc. of 1997 ACM SIGMOD, pp.369-380, 1997.

[8] R.Weber, H.J.Schek, S.Blott: “A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces”, Proc. Of the 24th VLDB Conference, pp.194-205, 1998.

[9] 櫻井，吉川，植村，児島：“多次元空間における類似探索手法の提案”，情処技報データベースシステム 119-18，pp.103-108，1999.

[10] N.Roussopoulos, S.Kelly, and F.Vincent: “Nearest Neighbor Queries”, Proc. ACM SIGMOD '95, pp.71-79, 1995.