データベースシステム 128－8
（２００２．７．１７）

配列を用いたキャッシュコンシャスな索引木の提案

高見澤秀久† 有次 正義†

† 群馬大学工学部情報工学科
E-mail: †{takamiza,aritsugi}@dbms.cs.gunma-u.ac.jp

あらまし 近年の研究において，キャッシュコンシャスな索引構造は優れた性能を示している．キャッシュコンシャスな索引木における重要な考えは，ノードからのポインタの削除にある．ポインタを削除することにより，ノードが持つ子ノードの数は増加し木の高さは低くなる．その結果として効果的なキャッシュの利用が可能となる．本稿では，キャッシュコンシャスな索引木「Array-Based Cache conscious trees」（ABC 木）を提案する． ABC 木では，木構造を完全木として配列により表現する．この索引木の構造や操作方法は B^+ 木に類似しているが， B^+ 木のように子ノードへのポインタを格納しない．ノード間の相対的な位置関係を計算することで目的のノードを得る．また，その高さの完全木を構築するのに必要となる領域をあらかじめ確保しノード間の位置関係を保つことで更新処理にも柔軟に対応する．そのため，ノードのオーバーフローやアンダーフローが発生した場合のコストも比較的低くなる．
キーワード キャッシュ，索引構造，メインメモリデータベース

Cache Conscious Trees using Arrays : A proposal

Hidehisa TAKAMIZAWA† and Masayoshi ARITSUGI†

† Department of Computer Science, Faculty of Engineering, Gunma University
E-mail: †{takamiza,aritsugi}@dbms.cs.gunma-u.ac.jp

Abstract Cache conscious indexing structures attracts many researchers because of their efficiency. The key idea to implement the trees is to remove pointers from nodes of trees. This allows a node to increase the number of child nodes, thereby making the height of a tree low. As a result, cache is used effectively. In this paper, Array-Based Cache conscious trees (ABC trees in short) are proposed. ABC trees are constructed as complete trees using arrays. While the tree is treated like a B+tree, it does not hold pointers to child nodes. A node of the ABC tree can be extracted by calculating spatial relations among nodes in the tree. Updates can be processed efficiently because the necessary space for nodes of the complete tree are allocated and the whole spatial relations among nodes in the space can be calculated in advance to the processing. Therefore, it is expected that the costs of node split and merge are relatively low.
Key words Cache, Indexing Structure, Main Memory Database

1. はじめに

近年の RAM の低価格化に伴い，巨大なメインメモリを持つ計算機の入手が容易になった．そのため，メインメモリデータベースは様々な分野に適用されることが期待されており，メインメモリデータベースに関連する研究は益々盛んになってきた．

CPU の処理速度とメインメモリへのアクセス時間の差は大きく，その差は今後も更に増加すると言われている．そのため，メインメモリデータベースでは，従来のデータベースのボトルネックであったディスク I/O に加え，メインメモリへのアクセスが新たなボトルネックとなる．

メインメモリへのアクセス回数を減らすための一つの手段として，キャッシュメモリの利用が一般的に知られている．キャッシュメモリとはメインメモリと CPU との間に置かれる高速バッファメモリのことである．その記憶容量は小さいが，主記憶装置に比較してかなり高速にアクセスできるメリットがある．従って，CPU はまずメインメモリではなくキャッシュメモリにアクセスし，データを要求する．CPU から要求されたデータがキャッシュメモリ中に存在する場合は，高速なキャッシュメモリからデータを読み込むことができる．しかしながら，データがキャッシュメモリ中に存在しない場合（キャッシュミス）は，目的のデータを取得するために，キャッシュに比べて低速なメインメモリにアクセスしなければならず，メインメモリへのアクセス時間が掛ることになる．従って，キャッシュミ

スの回数を減らすことで，処理全体の効率を向上させることが出来る．「キャッシュコンシャス」とは，キャッシュの効果的な利用を考慮して効率の向上を目指すことである．

索引技術は，データベースシステムにおいてデータ処理の効率を向上させる手法の一つであり，メインメモリデータベースにおいても同様の効果を期待することができる．従来のデータベースシステムでは，ディスク I/O のボトルネックに注目し，ディスク I/O を減らす研究が行われてきた．しかしながら，メインメモリデータベースではキャッシュミスがボトルネックとなる．そこで，索引技術にキャッシュコンシャスの概念を取り入れていくことにより処理効率の向上を望むことができる．

キャッシュコンシャスな索引木を実現する手法の一つとしてポインタの削除がある．削除したポインタの代わりにデータを格納することで，ノード当たりのデータ数は増加する．これにより，ノードを参照する際の計算に不必要となるポインタをメインメモリからキャッシュメモリに読み込まずに済む．そして計算に必要となるデータのみがキャッシュに読み込まれることでキャッシュミスの回数は減り，結果的に高速に処理することが可能となる．また，ノード当たりのデータ数が増加することから，一つのノードが持つ子ノードの数は増加する．その結果として，木全体の高さは低くなる．これにより，検索においてノードを参照する回数，つまりノードをキャッシュに読み込む回数が減り，キャッシュミスの回数も減る．

キャッシュコンシャスな索引木については従来から研究されている．[3] では，キャッシュコンシャスな探索木である「Cache Sensitive Search trees」（CSS 木）を提案している．CSS 木では構造が配列により表現されているため，全てのノードはその子ノードへのポインタを全く持っていない．これにより効果的なキャッシュラインの利用へと繋がり，探索時間の高速化を実現した．また，[4] では頻繁な更新処理にも対応すべく，キャッシュコンシャスな B^+ 木，「Cache Sensitive B^+ trees」（CSB^+ 木）を提案している．一つのノードからの子ノード全てを隣接して格納し，そのノードは先頭の子ノードへのポインタのみを保持することで，ポインタの削除を実現している．また，構造は B^+ 木に類似しているため，頻繁な更新処理への対応を実現している．

本稿では，キャッシュコンシャスな索引木「Array-Based Cache conscious trees」（ABC 木）を提案する． ABC 木の論理的な構造は，B^+ 木に類似している． ABC 木では木構造を配列により表現しているため，配列のオフセット計算により子ノードを求めることができる．このため，子ノードへのポインタを保持する必要がない．これにより，キャッシュの効果的な利用を期待することができる．また， ABC 木は B^+ 木と類似した構造を持つため，頻繁な更新にも対応し得る．

ポインタを削除する手法を適用したキャッシュコンシャスな索引木では，ノード間の位置関係を保つことが更新処理における重要な問題点となる． CSS 木では OLAP 環境を想定し，更新処理を一括で行うことでこの問題を避けてきた． CSB^+ 木では，ノードが子ノードへのポインタを一つ保持することで，この問題の解決を謀った．その結果，必要となる領域をあらかじめ確保する CSB^+ 木が B^+ 木と比較して高速であると結論づけている． ABC 木では木構造を完全木として実現する上で必要となる領域をあらかじめ確保することで，更新時におけるノード同士の位置関係の問題の解決を謀る． ABC 木ではノード間の位置関係はそのノードの識別子を計算することにより一意に求めることが可能であるため，親ノードや子ノードはポインタを辿ること無く計算により一意に求めることができる．更には索引部を構築する場合や更新処理において，リーフノードのキーとなる値の索引部における格納場所を求めたい場合においても，計算により一意に求めることが可能となる．以上のことから， ABC 木での処理の高速化が見込まれる．

本稿の構成は以下の通りである． 2 章でキャッシュコンシャスに関連する研究を紹介し， 3 章で我々の提案する ABC 木の概要を述べる． 4 章で ABC 木のアルゴリズムを説明し，最後に 5 章でまとめとする．

2. 関 連 研 究

キャッシュの動作を考えることは処理の高速化を考える上で重要な課題である．そのため，キャッシュを効果的に利用するための研究が盛んに行われている．例えば [1] では，リレーショナルデータベースにおいて，ページを分割し，分割された領域に属性ごとにデータを格納することでキャッシュコンシャスなデータの格納方式を実現した．また [2] では， R 木においてそのキーである MBR を相対化・量子化により圧縮し，キャッシュライン当たりの参照可能なキーの数を増加させることで，処理の効率化を実現している．

索引構造に関する研究においてもキャッシュの有効利用を考えたものがある．以下にキャッシュコンシャスな索引構造に関する研究を紹介する．

CSS-trees. CSS 木（Cache Sensitive Search trees）[3] は， OLAP （On-Line Analytical Processing）環境において，検索処理速度の向上を目指している．木構造を配列を用いて表現することにより，ポインタが不要となる構造を実現している．

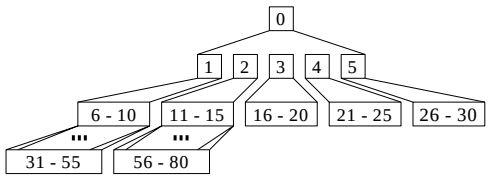

図 1　m=4，リーフノード数 64 の CSS 木

図 1はノード当たりのデータ数（m）が 4，データ部のノード数が 64 の CSS 木である．一つのノード当たり 4 個のデータを持つため，一つのノードが持つ子ノードの数は，$4 + 1 = 5$ 個となる．また，ノード番号が **31 - 80** の部分と **16 - 30** の部分がデータ部となり，残りの **0 - 15** の部分は内部ノード，つまり索引部となっている．

CSS 木では，検索における子ノードの識別は配列のオフセッ

トの計算により可能である．また，ノードサイズをキャッシュラインサイズに合せることで，一つのノードを参照するためのキャッシュミスが多くとも一回で済む．しかしながら，CSS 木は検索処理速度に関しては B^+ 木を上回っているが，頻繁な更新については考慮していない．

CSB^+-trees. CSB^+ 木（Cache Sensitive B^+ trees）[4] は，頻繁な更新にも対応することの出来るキャッシュコンシャスな B^+ 木である．CSB^+ 木は B^+ と類似したデータ構造を持つため，頻繁な更新にも対応することが可能となる．B^+ 木のノードが全ての子ノードへのポインタを保持しているのに対して，CSB^+ 木のノードは最初の子ノードへのポインタしか保持していない．一つのノードからの全ての子ノードは，一つのノードグループとして扱われる．ノードグループ内ではノードが隣接して格納してあるため，最初の子ノードへのポインタを辿ってオフセットを計算することで目的のノードを得ることができる．また，CSS 木と同様にノードサイズをキャッシュラインサイズに合わせている．ただし，ノードは先頭の子ノードへのポインタとノード内のデータ数を示すパラメータを保持しているため，CSS 木に比べてノード当たりのデータ数が少ない．

図 2ではノード当たりのデータ数が 2，データ部のデータ数が 9 の CSB^+ 木である．破線はノードグループを表している．一つのノードは $2+1=3$ 個の子ノードを持っており，それらの子ノードはノードグループ内で隣接して格納されている．また，ノードグループ間で隣接するリーフノードは双方向のポインタでつながれているため，データのみの検索も比較的容易に行うことができる．しかしながら，リーフノードにおいてもノード当たりのデータ数も CSS 木に比べて少ない．

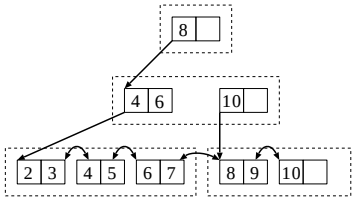

図 2　m=2，データ数 9 の CSB^+ 木

[4] では，CSB^+ 木の様々なバリエーションについても提案し，それらの比較検討を行っている．その中でも特に，ノード分割時のコストを削減するために，あらかじめノードグループに必要な領域を確保している Full CSB^+ 木が B^+ 木と比較して，探索，挿入，削除それぞれの処理速度において高いパフォーマンスを示している．

3. ABC　木

本稿で提案する ABC 木は，全てのポインタを削除することで，キャッシュコンシャスな索引木を実現する．[4] では，必要な領域をあらかじめ確保しておくことにより，更新処理に対しても高いパフォーマンスを得られることが実証された．ABC 木では，ある高さの木構造を構築するのに必要な領域をあらかじめ確保してあるため，現在の木の高さが変わらない限り，ノードの分割が発生しても新たに領域を確保することなく処理を実行することができる．このため，更新処理においても高いパフォーマンスを期待することができる．

ABC 木で扱うデータについては，あらかじめソートされているものとする．また，データの重複は認めない．

3.1　ABC 木の構造

ABC 木は，CSB^+ 木と同様にキャッシュコンシャスな B^+ 木である．完全木としてルートノードから幅優先順に固有の識別子を 0 から割り当てた配列に格納することにより，全てのポインタを削除することができる．ABC 木の論理的な構造は B^+ 木に類似しているため，頻繁な更新にも対応している．CSB^+ 木との相違点は，CSB^+ 木のノードは子ノードの先頭要素へのポインタを保持しているのに対して，ABC 木は子ノードへのポインタを持たないところにある．ABC 木は CSS 木と同様，全てのノードは子ノードへのポインタをいっさい保持しておらず，配列によって木構造を表現している．

図 3はノード当りのデータ数が 2，データ部のデータ数が 9 の ABC 木である．一つのノードは，$2+1=3$ 個の子ノードを持つ．全てのノードは一つの配列として隣接して格納されている．また，最後のリーフノードはデータが入ってはいないが，完全木として必要な領域を確保しているため空ノードとなっている．

3.2　ノードの取得

表 1　ノード ID の計算式

nodeID	formula
child nodeID	$nodeID*(m+1)+offset+1$
parent nodeID	$\left\lceil \frac{nodeID}{m+1} \right\rceil - 1$

ABC 木は作成時に完全木となるように，データの入っていないノードの分の領域も確保する．そのため全てのノードが隣接して格納されている．これによりノード間の位置関係が相対的に保たれることになるから，子ノード，親ノード，隣接ノードは，ポインタの参照を繰り返すことなく，計算により求めることができる．ノード ID が $nodeID$ であるノードの子ノード ID $childID$ の範囲は，

$$nodeID*(m+1)+1 \leqq childID \leqq (nodeID+1)*(m+1)$$

となる（ただし，m はノードの持つ最大データ数）．そのため，$nodeID$ の子ノードを取得する際には，ノード内におけるオフセットを求めることにより，子ノードの ID を求めることができる．また，ノード ID が $nodeID$ であるノードの親ノード ID $parentID$ は，

$$parentID = \lceil nodeID/(m+1) \rceil - 1$$

により一意に求めることができる．表 1に以上の計算式を示す．

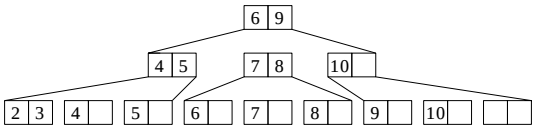

図 3　m=2，データ数 9 の ABC 木

3.3　データ部と索引部の関係

ABC 木は完全木として領域を確保しデータを格納しているため，リーフノードのキー値に対応する索引部のノードを一意に求めることができる．つまり，索引部を探索すること無く，多くとも一回のキャッシュミスで，あるリーフノードのキー値に対応する索引部のノードを得ることができる．

あるリーフノードにおいて，その値が索引部におけるキー値である場合は，リーフノードの ID から，索引部におけるノード ID を計算し，そのノードにもキー値を格納する．前述のように，あるノードの親ノード ID は，計算により一意に求めることができるから，そのキー値が親ノードにおいて $(m+1)$ 番目以外の要素である場合は，親ノードが目的となるノードであるから，適当な位置にそのキー値を格納する．また，そのキー値が $(m+1)$ 番目の要素である場合は，そのノードのさらに親ノードにおいて同様に判定を行う．そのキー値が親ノードにおいて $(m+1)$ 番目以外の要素になるか，ルートノードに到達するまで同様の処理を行うことで，索引部におけるキー値の適当な場所を決定することができる．

3.4　空 ノー ド

リーフノードにおいて空ノードが存在する場合，その空ノードの場所に気をつけなければならない．空ノードの場所として適切なのは，空ノードのキー値が索引部の最下層のノード内で最後の要素となるような場所である．空値が索引部における最下層ノード以外のノードには出現しないことで，削除処理における空ノードの扱いが容易になる．木構造を保つ上では，空ノードの扱いが重要な問題となってくる．

また，データのバランスについても検討課題である．その理由はバランスを考慮することによる処理の複雑化にある．ABC 木は，前述のように木構造を配列により表現しているため，空ノードの扱いには注意を払わなければならない．そのため従来の B^+ 木のようなバランスの取り方を考えることにより，結果的に処理が非効率的になることも考えられるからである．

4.　アルゴリズム

ここでは ABC 木の Bulkload, Search, Insertion, Deletion について，以下のパラメータを用いて説明する．

n　：　総データ数

m　：　ノードあたりの要素数

4.1　Bulkload

ABC 木の Bulkload では，リーフノードに値を格納しながら随時索引部にも値を格納していく．

まず, 木の高さ $height$ をデータを格納するのに最低限必要なリーフノード数 LN を用いて以下の条件に従って求める．

$$(m+1)^{height-1} < LN \leqq (m+1)^{height}$$

リーフノード数 LN は，データを m 個ずつのノードに分配することから $LN = \lceil n/m \rceil$ となり，以下の式により高さ $height$ を計算することができる．

$$height = \left\lceil log_{m+1} \left\lceil \frac{n}{m} \right\rceil \right\rceil$$

次に，全体のノードの数 TN を計算し，上位ノードから幅優先の順に固有の 0 からのノード ID を割り当てる．TN は以下に示すように，上位レベルから順に加算していけば良い．

$$\begin{aligned} TN &= \sum_{i=0}^{height} (m+1)^i \\ &= 1 + (m+1) + (m+1)^2 + \cdots + (m+1)^{height} \\ &= \frac{(m+1)^{height+1} - 1}{m} \end{aligned}$$

索引木のリーフノード部，つまりノード ID が $\{(m+1)^{height} - 1\}/m$ から $\{(m+1)^{height+1} - 1\}/m - 1$ のノードに，値を小さい順に格納していく．

表 2　ABC 木の構造に関する計算式

item	formula
height of a tree	$\left\lceil log_{m+1} \left\lceil \frac{n}{m} \right\rceil \right\rceil$
total number of leaf nodes	$(m+1)^{height}$
total number of nodes	$\frac{(m+1)^{height+1} - 1}{m}$
the head of leaf nodes	$\frac{(m+1)^{height} - 1}{m}$

本稿で扱う ABC 木では，あるノードのキーは右部分木の内で最も大きい値となるようにする．そのため，先頭以外のリーフノードにおける最初の要素が索引部でのキー値となる．前述のように，あるリーフノードにおけるキー値の索引部におけるノード ID は，一意に求めることができるから，リーフノードに格納された値が索引部におけるキー値として適当である場合は，リーフノードの ID から，索引部におけるノード ID を計算し索引部にもキー値を格納する．また，リーフノードに空ノードが発生した場合は，空ノードが上位ノードに影響を及ぼさないように注意しなくてはならない．前述の通り，空ノードとしてはその親ノードの最後の要素のキーとなるリーフノードを選ぶ．Bulkload に必要となる計算式をまとめたものを表 2に示す．

4.2　Search

ルートノード（$nodeID = 0$）からノード内の値を二分探索することにより，次に進むべき子ノードを決定する．ノード ID$nodeID$ を持つノードの子ノードの先頭ノードの ID は，$nodeID * (m+1) + 1$ であるから，ノード内の二分探索に

よって求めた位置をオフセットとして目的の子ノードの ID を式 $nodeID*(m+1)+offset+1$ で求める．この子ノードを次の探索の対象として同様の処理を行う．この作業を対象となるノードがリーフノードになるまで繰り返していく．探索の対象となるノードがリーフノードになった場合は，サーチキーと等しい値をそのリーフノード内にて探索する．この場合も同様にノード内を二分探索することで目的の値を探索する．ここで，サーチキーと等しいキー値を発見した場合は探索成功，サーチキーと等しいキー値を発見できなかった場合は探索失敗となる．

4.3 Insertion

ABC 木の挿入処理は，B^+ 木の挿入処理と似ている．しかしながら，ABC 木ではノード内のオーバーフローの際にも，木全体のデータ数によっては，新たな領域を確保する必要が無い場合も存在する．つまり，総データ数が，$(m+1)^{height}*m$ 未満の時は，配列内に空き領域が存在しており，現在使用している領域がオーバーフローを起こすことはない．しかしながら，総データ数が $(m+1)^{height}*m$ 以上の場合は，現在使用している領域だけではオーバーフローを起こしてしまうため，新たな領域を確保しなければならない．以下では，新たな領域を確保する必要がない場合，つまり木の高さが変わらない場合と，新たな領域を確保する必要がある場合，つまり木の高さが変わる場合についての処理をそれぞれ説明する．

[木の高さが変わらない場合]

リーフノードにデータを格納する余地がある場合については，まず挿入の対象となるリーフノードを検索する．挿入対象となるリーフノードに値を格納する余地があれば，そのリーフノード内を二分探索し値を挿入する．挿入対象となるリーフノードに値を格納する余地が無い場合は，値を挿入する余地を持つリーフノードを見つけ，そのノードにデータをシフトすることで挿入対象ノードに空き領域を確保する．データをシフトするノードについては，挿入対象となるノードから最も近いものを選ぶ．空き領域にデータをシフトする際，リーフノードのキーをシフトした場合は索引部の更新も必要となる．索引部の更新については，更新されたリーフノードのノード ID から適当な索引部のノード ID とノード内におけるそのキーの格納場所を計算により求め，索引部の目的となるノード内の値を更新する．

[木の高さが変わる場合]

リーフノードにデータを格納する余地が無い場合は，現在の木の高さではデータを格納することができない．そこで，次のレベルのノード分の配列を作成し，木構造を表した配列とその配列をポインタでつなぐことで，一つ高い木構造を表現することができる．この様子を図4に示す．

すべてのデータを格納することができるだけの領域を確保したので，次は現在のリーフノードのデータを次のレベルのリーフノードに移動させる．ここでは Bulkload の場合と同様に，空ノードの格納位置やノード当りのデータ数を計算して，データ部の内容を次のレベルの適当な位置に小さい値から順に格納していく．移動するデータが次のレベルのリーフノードのキーであった場合は，その値を索引部の適当な位置に格納しながら順次処理を行っていく．

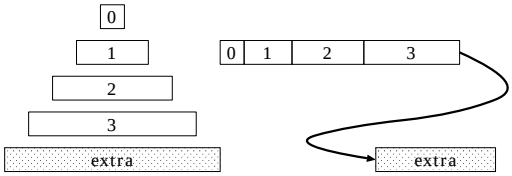

図4　配列の追加

4.4 Deletion

ABC 木の削除処理では，B^+ 木のようにノード内のデータ数を半分以上に保たなければならないという制約を課してはいない．しかしながら，削除によって空ノードが数多く発生しすぎると，現在の木構造を保つのが困難になる．そこで，リーフノードに存在するデータ数が $(m+1)^{height}+1$ 以下の場合は，一段階レベルの低い完全木とすることにより，総データ数に見合った木構造を，リーフノードからデータを移動することにより構築する．また，リーフノードに存在するデータ数が $(m+1)^{height}+1$ より大きい場合はリーフノードからの削除を行えば良いのだが，空ノードが発生する場合は処理がやや複雑になる．以下では，木の高さが変わらない場合と，木の高さが変わる場合についての処理をそれぞれ説明する．

[木の高さが変わらない場合]

リーフノードのデータ数が $(m+1)^{height}+1$ より大きい場合は，データを削除しても，現在の木構造の整合性が保たれるから，木の高さを低くする必要はない．そこで，削除の対象となるデータを格納しているリーフノードを検索し，そのノード内のデータの削除を行えば良い．削除対象となるノードから，その値を削除しても空ノードにならない場合は，目的のデータを削除すればよい．削除した値がノードのキーであった場合は索引を更新する．

しかしながら，対象ノードからデータを削除することによって，ノードが空ノードになってしまう場合も有り得る．もし，削除対象ノードが空になったとしても索引部に影響を及ぼさない場合（対象ノードのキー値が，索引部の最下層のノード内にあり，かつそのノード内における最終要素である場合）は，対象ノードからデータを削除し，そのノードを空ノードにすればよい．対象ノードが空ノードとして適切で無い場合については，削除しても空ノードにならないか，空ノードになっても問題が無いノードの内，対象ノードから最も近いノードを選び，そのノードから値を一つずつシフトすることで，削除対象となるノードが空ノードにならないようにする．値をシフトする際に，リーフノードのキーが変更された場合は，索引部の変更も必要となる．

[木の高さが変わる場合]

リーフノードのデータ数が $(m+1)^{height}+1$ 以下の場合は，データを削除することで，適切でない位置に空ノードが生じてしまい，現在の木構造の整合性を保つことが出来ない．そのため，現在の木構造から一段階だけ木を低くすることで，木構造の整合性を保つ．

木の高さを低くすることで，リーフノードのルートからの深さは一段階浅くなる．そこで，削除対象のリーフノード以外のノードを順に上位レベルの新しいリーフノードに移動する．Bulkload の処理と同様に，新しいリーフノード部分における空ノードの格納位置を計算して，現在のリーフノードから新しいリーフノードへ，小さい値から順に移動していく．移動したデータが新しいリーフノードにおけるキーとなる場合には，順次索引部の更新も行い，木を構築していく．元のリーフノードは未使用領域として，挿入処理においてオーバーフローが発生し，新たに領域が必要となった場合に備えてそのまま保持する．この様子を図5に示す．

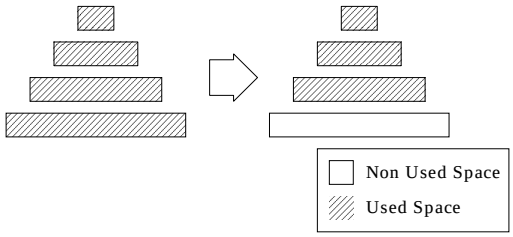

図5　削除によって木の高さが低くなる場合

5. おわりに

本稿では，キャッシュコンシャスな索引木 ABC 木を提案した．ABC 木は木構造を完全木として配列により表現することで B^+ 木からのポインタの削除を実現した．これにより，キャッシュライン当りのデータ数の割合が増加し，B^+ 木に比べて，よりキャッシュの効果的な利用が可能となった．また，完全木として必要な領域をあらかじめ確保することで，頻繁な更新にも高速に対応することが期待される．ただし，本稿では，木のバランスの取り方については言及していない．そのため，処理効率を向上させる最適なデータの配置を求めることが今後の課題となる．

文　献

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, editors, *Proc. 27th International Conference on Very Large Data Bases*, pages 169–180. Morgan Kaufmann, 2001.

[2] K. Kim, S. K. Cha, and K. Kwon. Optimizing multidimensional index trees for main memory. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 475–486. ACM Press, 2001.

[3] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. In M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, editors, *Proc. 25th International Conference on Very Large Data Bases*, pages 78–89. Morgan Kaufmann, 1999.

[4] J. Rao and K. A. Ross. Making B^+-trees cache conscious in main memory. In W. Chen, J. F. Naughton, and P. A. Bernstein, editors, *Proc. ACM SIGMOD International Conference on Management of Data*, pages 475–486. ACM Press, 2000.