データベースシステム 128－9
（２００２．７．１７）

CSS-tree の挿入処理の高速化

本田喜久[†]　川島英之[†]
今井倫太[††]　安西祐一郎[††]

近年, キャッシュを意識して探索を高速化するメモリ索引構造が提案されている. 中でも, 1999 年に提案された Full Cache Sensitive Search Tree(CSS-tree) は, 探索時間がデータサイズによってはハッシュよりも短く, また索引構造のために必要なメモリ空間使用量が B-tree よりも少ない点で優れている. 他方, 更新処理が遅いため, CSS-tree が有用であるのはオンライン解析処理のように更新処理が極めて少ない状況に限られると言われている. しかし, 更新処理を追加処理のみに限定すれば, 索引構造の再構築時間を減らすことができる. そこで本研究では, 追加処理のみが発生する状況において, CSS-tree への挿入処理を高速化する手法を提案する. 提案手法は, (1) 配列と CSS-tree のノードの拡張に要するコストを減らすために, あらかじめ余分な索引構造領域を獲得しておき, (2) 配列と CSS-tree の葉ノードの再マップに要するコストを減らすために, CSS-tree の葉ノードの領域をなくし, CSS-tree の内部ノードのスロットに, 配列のキー値を格納する. 提案手法の有効性を評価するために, C 言語とアセンブラを用いて, SunOS 5.6 上に実験用データベースシステムを実装し, 挿入処理時間を測定した. 実験の結果, 提案手法は, 更新処理を追加処理に限定しない再構築手法と比べて, 最大で 7.18 倍速くなることがわかった.

Accelerating insertion processing to the CSS-tree

Yoshihisa Honda,[†] Hideyuki Kawashima,[†] Michita Imai[††]
and Yuichiro Anzai[††]

In recent years, memory index structures that accelerate search speed utilizing the structure of CPU cache have proposed. Especially, the Full Cache Sensitive Search Tree(CSS-tree) proposed in 1999 is faster than hash depending on data size and require smaller amount of space than B-tree. On the other hand, reconstructing the CSS-tree is so slow that it has been considered that the CSS-tree is only useful under static environments such as OLAP. However, if updating is limited only to additional insertions, the reconstruction cost can be dramatically decreased. Then, in this research, we propose the new techniques that accelerate insertion processing to the CSS-tree. The technique consists of (1) preliminary acquiring excess index structure space to reduce cost for node expanding and (2)omitting leaf nodes to reduce cost for remapping the leaf nodes and instead, storing key value into internal nodes. To evaluate the proposed technique, we implemented an experimental system with C language and assemblers on SunOS 5.6 and measured the insertion time and the size of memory space for the structure. The result showed that the proposed technique accelerates 7.18 times faster than original technique.

1. はじめに

ディスクデータベースシステムよりも検索速度を高速化するために, メインメモリ上での動作が前提とされる, TimesTen や P*TIME のようなメインメモリデータベースシステムが開発されつつある [1)2)].

メインメモリデータベースシステムで検索処理を高速化するには, メモリ索引構造が必要である. 従来のメモリ索引構造には, ハッシュ, B-tree, そして T-tree がある. この中で検索速度が最速であるのはハッシュであり, 索引構造の規模と検索速度のバランスが最良なのは T-tree だと考えられてきた.

この考えはメモリアクセスのコストが一定ならば成り立つが, しかし CPU 速度の年間向上率は 60% であるのに対して, メモリアクセス速度の年間向上率は 10% であるため成り立たない. そのためキャッシュメモリを意識したメモリ索引構造が提案されるように

† 慶應義塾大学大学院 理工学研究科
Gradudate School of Science and Technology, Keio University
†† 慶應義塾大学 理工学部
Faculty of Science and Technology, Keio University

なってきた．

そのなかに，1999 年に提案された Full Cache Sensitive Search Tree(CSS-tree) がある．CSS-tree の長所は，データ量が小さいときには検索時間ハッシュよりも短いことと，索引構造のために必要なメモリ空間量が B-tree よりも少ないことである．

他方，更新処理に伴う再構築処理が遅いために，CSS-tree が有用であるのは，オンライン解析処理のような，更新処理が極めて少ない状況に限られると言われてきた．CSS-tree を再構築するには，更新処理の際に，(1) 配列と CSS-tree のノードを拡張し，(2) 配列と CSS-tree の葉ノードを再マップする必要がある．

しかし，更新処理を追加処理のみに限定すれば，索引構造の再構築時間を減らすことができる．そこで本研究では，追加処理のみが発生する状況において，CSS-tree への挿入処理を高速化する手法を提案する．

提案手法は，(1) 配列と CSS-tree のノードの拡張に要するコストを減らすために，あらかじめ余分な索引構造領域を獲得しておき，(2) 配列と CSS-tree の葉ノードの再マップに要するコストを減らすために，CSS-tree の葉ノードの領域をなくし，CSS-tree の内部ノードのスロットに，配列のキー値を格納する．

提案手法では, CSS-tree への挿入処理の高速化のために，(1) で余分な索引構造領域を獲得するため使用メモリ空間量が増加する．この増加量を減らすために，(2) で CSS-tree の葉ノード領域を使用しないために使用するメモリ空間量を減らす．

提案手法の有効性を評価するために, C 言語とアセンブラを用いて，SunOS 5.6 上に実験用データベースシステムを実装し，挿入処理時間を測定する．

本論文の構成は次の通りである．第 2 章では CSS-tree の構造および動作について述べる．第 3 章では CSS-tree の挿入処理を高速化する手法を提案する．第 4 章では評価実験について述べる．最後に第 5 章では結論を述べる．

2. Cache Sensitive Search-tree

キャッシュを意識した索引構造には，CSS-tree, CSB+-tree[4] があり，そのうち，本研究では CSS-tree の一種である Full CSS-tree[3] への挿入処理の高速化を提案する．これ以降，本論文では Full CSS-tree のことを CSS-tree と表記する．提案手法の理解を容易にするために，本章では Full CSS-tree について説明する．

2.1 Full CSS-tree の構造

Full CSS-tree には，各ノードに m 個のキーが存在する．m は各ノードがキャッシュラインに合うように選択する．例えば図 1に m=2 の Full CSS-tree を示す (箱の中の数はノード番号で各ノード内には 2 つのキーが含まれている)．

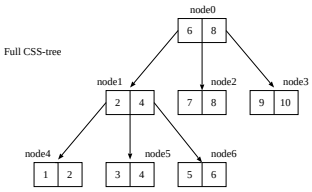

図 1 Full CSS-tree の全体像

Full CSS-tree の葉ノードは図 2のように配列内に格納されている．各ノードは子供へのポインタを持たない．各ノードの子供は配列のオフセットから計算される．Full CSS-tree は配列 **a** に関するディレクトリ構造となっている．

ノード番号は 0 から始まり，キーの値は重複がなくソートされていなければならない．**b** と番号付けされた内部ノードの子供のノード番号は内部ノードの子ノードの先頭番号を s，内部ノードの子ノードの先頭番号を d とすると次式で計算される．

$$s = b(m+1)+1 \quad (1)$$

$$d = b(m+1)+(m+1) \quad (2)$$

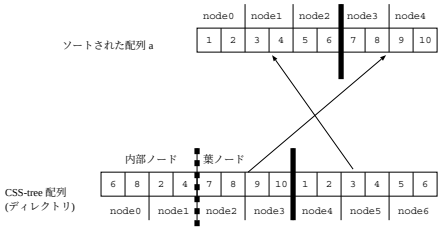

図 2 2*5 element in the array

Full CSS-tree では，配列の左半分の前に右半分 (配列の左半分より高いレベル) の配列を置く．図 2を見るとわかるように，自然な木の順番ではノード 4-6 の前にノード 2-3 が格納されているが，ソートされた配列ではノード 2-3 の前にノード 4-6 が格納されている．

検索を実行する時は，親から子へ図 1のようなディレクトリ構造内のオフセットを計算していく．オフセットの計算がディレクトリ構造内の最後 (図 1のノード 1 の最後のキー) のオフセットを超えると検索を終了する．

葉ノードの 2 つの部分は図 2のようにソートされた

配列にマップされる．葉ノードの 2 つの部分の境界を示す境界を示すために y を使う．y はディレクトリ配列内の最も深いレベルにある最初のキーである (図 2 のノード 4 内の最初のキー)．葉ノード内の，あるキーのオフセット x が与えられると，y と比較しソートされた配列のどの部分に対応するかを決定する．

$x > y$ の場合，ソートされた配列 **a** の先頭の要素から $x - y$ の位置に対応する要素を見つけ出す事ができる．逆に $x < y$ の場合はソートされた配列 **a** の最後の要素から $y - x$ の位置に対応する要素を見つけ出す事ができる．例えば，図 2では，葉ノード 3 の最初のキーはソートされた配列 **a** のノード 4 の最初のキーとして見つけ出す事が出来る．

最後の内部ノードのノード番号とノード番号 y は次の補題で定義される．

補題 1　ソートされた配列 **a** に含まれるキーの数 n は $n = N * m$ であらわす事ができ，N は葉ノードの総数であり，m は各ノード内に含まれるキーの数である．内部ノードの総数は $\frac{(m+1)^k-1}{m} - \lfloor \frac{(m+1)^k-N}{m} \rfloor$ で求める事ができ，ボトムレベルにある葉ノードの最初のノード番号は $\frac{(m+1)^k-1}{m}$ で求める事ができる．上記両式における k は $k = \lceil \log_{m+1}(N) \rceil$ で求める．

2.2　Full CSS-tree の探索

Full CSS-tree の探索は根ノードからはじめる．内部ノードでは常に，そのノード内で 2 分探索をおこない，分岐すべき枝を決める．それは葉ノードに到達するまで繰り返される．葉ノードに到達すると葉ノード内で 2 分探索をおこない，求められた Full CSS-tree の配列の添字からソートされた配列の添字を求める．

ノード内の先頭スロット値より，探しているキー値以下だと一番左の枝に分岐する．また，ノードの最後のスロット値より大きいと一番右の枝に分岐する．この 2 つにあてはまらない時は，2 分探索しているスロットの左のスロット内のキー値より探しているキー値が大きく，探しているキー値が 2 分探索しているスロット値以下のスロットを見つけると，そのスロットから左の枝に分岐する．

図 3に，$m = 3, n = 21$ の時の Full CSS-tree からキー値 4 の探索の例を示す．

(1)　根ノードから探索を開始
(2)　キー 4 はノード 0 の最初のスロット内の値 11 より小さいのでノード 1 に分岐
(3)　ノード 1 内で 2 分探索を行い，キー 4 は $2 < 4 < 5$ であり，ノード 1 は内部ノードなので，ノード 6 に分岐
(4)　葉ノードであるノード 6 内で 2 分探索を行い，キー 4 と一致するスロットをみつけたので，探索終了

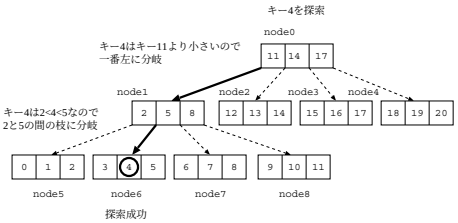

図 3　探索の例 (キー 4 の場合)

2.3　Full CSS-tree の問題

文献 [3] において，Full CSS-tree が有効である環境は，オンライン解析処理のように更新がほとんど発生しない環境だと述べられている．

その理由は，Full CSS-tree の再構築コストが大きいからである．データベースシステムに更新操作をおこなうには，次の手順をふむ必要がある．

(1)　索引構造の更新
　(a)　索引構造へのロック
　(b)　索引構造の変更
　(c)　索引構造へのアンロック
(2)　データの追加

上記操作の内，1(b) 索引構造の変更に要するコストが大きいことが文献 [3] において述べられている．

したがって，索引構造の変更に要するコストを下げることができれば，Full CSS-tree の再構築時間を削減することができる．これにより，従来は難しいと考えられてきた，オンライン解析処理以外への応用ができるようになる．

3.　CSS-tree への挿入処理の高速化

本研究の目的は，CSS-tree への挿入処理を高速化することである．本章では，この目的を達成するための手法を 4 つ提案する．

3.1　単　純　法

新しいデータが到着したあとの，CSS-tree のもっとも単純な構築手法は，図 4のようになる．これを単純法と呼ぶことにする．

3.2　マップ　法

単純法の手順でコストが大きいのは，2 から 5 までにおこなっている，メモリ空間の解放と獲得である．データベースの変更処理が追加処理のみである場合，メモリ空間のマップ手法を工夫することで，このコストを払わずに CSS-tree を再構築できる．本節では，2 つのマップ法を提案する．

(1) データの到着
(2) 配列空間を解放
(3) CSS-tree のノードを解放
(4) 配列要素を獲得
(5) CSS-tree のノードを獲得
(6) 配列の最終要素に到着データを追加
(7) 補題 3.3.1 に基づいて，葉ノードの総数から内部ノードの総数を計算
(8) 葉ノードと配列をマップ
(9) CSS-tree を構築
(10) 挿入完了

図 4 単純法

3.2.1 リアロック法

挿入データが到着するたびに，配列の要素を確保したまま，配列のサイズを CSS-tree の 1 ノード分拡張して拡張部にデータを挿入すれば，メモリ空間の解放コストと獲得コストを削減できる．この手法をリアロック法と呼ぶことにする．リアロック法を用いたデータ挿入処理の手順を図 5に示す．

(1) データの到着
(2) 配列の要素に空きがあれば，そこに挿入して完了．空きが無い場合はステップ (3) へ移動
(3) 配列を CSS-tree の 1 ノード分拡張
(4) データを配列の拡張部に格納
(5) 配列が 1 ノード分拡張されたので CSS-tree の葉ノードの総数がひとつ増える．補題 3.3.1 に基づいて，増加した葉ノードの総数から内部ノードの総数を計算
    (a) 内部ノードの総数が 1 ノード分増加していたら，CSS-tree にノードをふたつ追加
    (b) 内部ノードの総数が増加していなかったら，CSS-tree にノードをひとつ追加
(6) 葉ノードと配列をマップ
(7) CSS-tree の再構築
(8) 挿入完了

図 5 リアロック法

図 6に，もとの配列に含まれるキーの数 $n = 10$, 各ノード内のキーの数 $m = 2$ にキー値 11 が挿入される様子を示す．

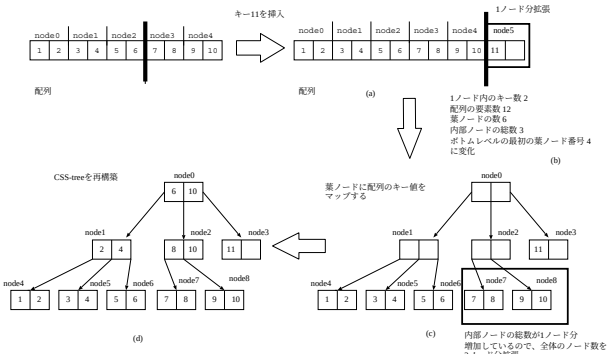

図 6 リアロック法の例 ($m = 2, n = 10$)

3.2.2 プリアロック法

初めて CSS-tree を構築するときに，あらかじめ余分な配列サイズと CSS-tree 全体に必要なノード数を余分に獲得しておけば，メモリ空間を解放し，再獲得する必要がなくなる．そこで，CSS-tree が平衡木になるようにメモリ空間を確保しておき，データが到着すると CSS-tree を再構築する手法を提案する．この手法をプリアロック法と呼ぶことにする．プリアロック法の処理手順を図 7に示す．

(1) データの到着
(2) 全データ数が余分に確保しておいた配列の要素数を越えている調査
    (a) 越えていた場合は，CSS-tree を平衡木にできるように，追加の配列空間と CSS-tree ノードを獲得
    (b) 越えていない場合は処理を続行
(3) データを配列に格納
(4) 葉ノードと配列をマップ
(5) CSS-tree を再構築
(6) 挿入完了

図 7 プリアロック法

プロアロック法の例を図 8に示す．この例では，はじめ，配列に含まれるキー数 $n = 10$，CSS-tree の各ノードのキー数は $m = 2$ であり，平衡木になるように余分な配列空間と CSS-tree 空間が獲得されている．その状態に，キー値 11 が挿入される様子を示す．

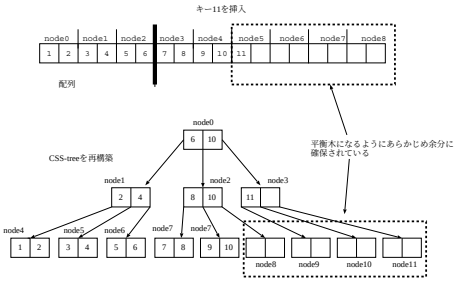

図 8　プリアロック法の例 ($m = 2, n = 10$)

3.2.3　マップ法の問題点

リアロック法とプリアロック法では, 葉ノードに配列をマップする必要がある．このとき，葉ノードを存在させるためにメモリ空間が使用され，マップにも時間がかかる．そこで，このコストをなくすために，マップをおこなわずに CSS-tree を構築する，非マップ法を提案する.

3.3　非マップ法

3.3.1　非マップ-リアロック法

非マップ-リアロック法は，挿入データが到着すると配列の要素を確保したまま，配列のサイズを 1 ノード分拡張し，拡張部にデータを挿入する手法である．非マップ-リアロック法がリアロック法と異なる点は，CSS-tree のノードを拡張するのに，内部ノードの数を増加させる必要しかないことである，このために葉ノードのためのメモリ空間を獲得する必要がなく，葉ノードと配列をマップする必要もなくなる.

非マップ-リアロック法の処理手順を図 9に示す.

図 10に, もとの配列に含まれるキーの数 $n = 10$, 各ノード内のキーの数 $m = 2$ である場合に，キー値 11 が挿入される様子を示す.

3.3.2　非マップ-プリアロック法

非マップ-リアロック法は, 初めて CSS-tree を構築するときに，あらかじめ配列サイズと CSS-tree の内部ノードを余分に獲得しておき，データが到着すると CSS-tree を再構築する手法である．事前の獲得メモリサイズは，CSS-tree を平衡木にさせる値にした.

非マップ-プリアロック法の処理手順を図 11に示す.

図 12に, 初めて CSS-tree を構築するときに，配列に含まれるキー数 $n = 10$, 各ノード内のキー数 $m = 2$ の状態であり，かつ平衡木になるように余分に配列のサイズと CSS-tree 全体に必要なノード数が確保されていた状態に，キー値 11 が挿入される様子を示す.

3.4　ま　と　め

本章では CSS-tree への追加処理を高速化するため

(1)　データの到着
(2)　配列の要素に空きがあれば, そこに挿入して完了. 空きがない場合はステップ (3) へ移動
(3)　配列を CSS-tree の 1 ノード分拡張
(4)　データを配列の拡張部に格納
(5)　配列が 1 ノード分拡張されたので CSS-tree の葉ノードの総数が 1 ノード分増える．補題 3.3.1 に基づいて，増加した葉ノードの総数から内部ノードの総数を計算
　(a)　内部ノードの総数が 1 ノード分増加していたら，もともとの CSS-tree 全体のノード数を 1 ノード分拡張
　(b)　内部ノードの総数が増加していなければ，ステップ (6) へ移動
(6)　CSS-tree を再構築
(7)　挿入完了

図 9　非マップ-リアロック法

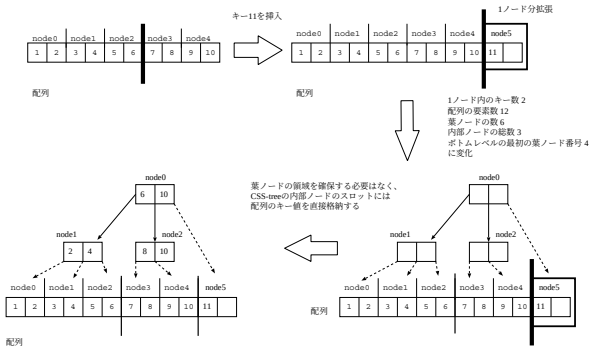

図 10　非マップ-リアロック法の例 ($m = 2, n = 10$)

(1)　データの到着
(2)　全データ数が余分に確保しておいた配列の要素数を越えているか調査
　(a)　越えていた場合は，配列と CSS-tree に必要な内部ノードを獲得．このときの獲得サイズは，CSS-tree を平衡木にさせる値
　(b)　越えていない場合は処理を続行
(3)　データを配列に格納
(4)　CSS-tree を再構築
(5)　挿入完了

図 11　非マップ-プリアロック法

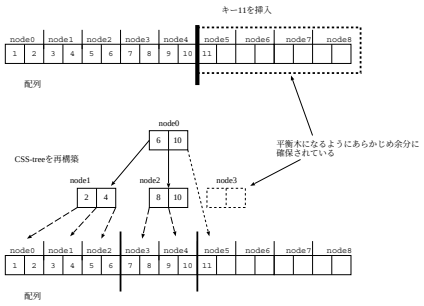

図 12　非マップ-プリアロック法の例 ($m = 2, n = 10$)

に，リアロック法，プリアロック法，非マップ-リアロック法，そして非マップ-プリアロック法の 4 つの手法を提案した．

各手法のステップ数をまとめると表1のようになる．

表 1　提案手法のステップ数

手法	ステップ数
リアロック法	8
プリアロック法	6
非マップ-リアロック法	7
非マップ-プリアロック法	5

表 1において，最もステップ数が少ない提案手法は非マップ-プリアロック法である．これより，最も高速な提案手法は，非マップ-プリアロック法だと考えられる．これを確かめる実験について第 4 章で述べる．

4. 評 価 実 験

本章では，提案手法について，挿入時間，使用メモリ量，そして探索時間の測定と比較を述べる．

4.1 実 験 環 境

実験に使用した計算機の性質を表 2に示す．

表 2　実験に用いた計算機の性質

要素	説明
システムモデル	Ultra 5/10 Model 440
CPU	Ultra Sparc IIi
CPU の動作周波数	444MHz
1 次キャッシュのサイズ	16KB
2 次キャッシュのサイズ	2MB
メインメモリの規模	1024MB
仮想メモリの規模	1.3GB
オペレーティングシステム	SunOS 5.6

実験データベースシステムを，C 言語とアセンブラを用いて SunOS 5.6 上に実装した．コンパイラには GNU の gcc-2.95.3 を使用した．

4.1.1 索 引 構 造

実験で実装した索引構造を表 3に示す．

表 3　索引構造の実装

索引構造名	実装方式
ハッシュ	チェインバケットハッシュ
B-tree	B-tree
CSS-tree	Full CSS-tree

4.2 挿入時間の比較

4.2.1 索 引 構 造

単純法，リアロック法，プリアロック法，非マップ-リアロック法，そして非マップ-プリアロック法について挿入時間を測定した．

各手法とも，CSS-tree のノード内のスロット数は 16，葉ノード内のキーの総数がそれぞれ 1 万，5 万，10 万，50 万，100 万の時に CSS-tree をあらかじめ構築しておき，その状態から 1 件データを挿入した時に，CSS-tree の挿入処理にかかる時間を測定した．

4.2.2 実 験 結 果

実験結果を図 13に示す．

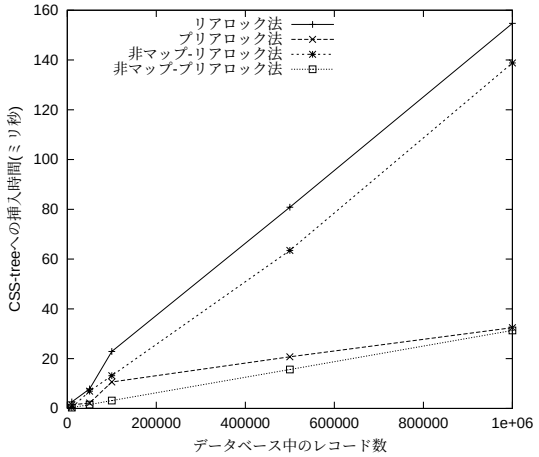

図 13　挿入時間の比較

図 13より，非マップ法はマップ法よりも高速であり，さらに非マップ- プリアロック法は非マップ-リアロック法よりも高速であることがわかる．単純法はあまりに遅いので，図 13には掲載しなかった．

図 13より，提案手法で最速であるのは非マップ-プリアロック法であり，最も遅いのはリアロック法だとわかる．最も差がでているのはレコード件数が 1 万件のときである．このとき，非マップ-プリアロック法はリアロック法よりも 7.18 倍速い．

非マップ法がマップ法よりも高速な理由は，マップ

法は挿入データがあると，CSS-tree の葉ノードと配列の再マップをおこなうが，非マップ法では，CSS-tree の葉ノードがなく，直接内部ノードに配列のキーの値を格納していくからであると考えられる．また，非マップ-プリアロック法が非マップ-リアロック法より高速な理由は，非マップ-プリアロック法はあらかじめ余分に配列および CSS-tree の内部ノードの数を確保しており，それに対し，非マップ-リアロック法はデータの挿入がある度に，配列および必要に応じて CSS-tree の内部ノードの数を確保する必要があるので，メモリ空間の確保にかかる時間が速度差に出たと考えられる．

以上の実験結果より，配列と CSS-tree のノードの拡張に要するコストを減らし，配列と CSS-tree の内部ノードの再マップに要するコストを減らした提案手法である非マップ法の有効性を確認することができた．

4.3 メモリ空間量の比較

リアロック法，プリアロック法，非マップ-リアロック法，そして非マップ-プリアロック法を用いた場合に，索引構造のために必要なメモリ空間量を示す．配列にはレコードのキーとレコードへのポインタを格納し，CSS-tree のノード内にはキー値だけを格納する．レコードへのポインタを格納するため空間を $R = 4$ バイト，キー値を格納するための空間を $S = 4$ バイト，キー値の総数を n，CSS-tree の各ノードに格納するキー数を m とする．また，葉ノードの総数 N は $n = N * m$ で求め，内部ノードの総数 I は $\frac{(m+1)^k-1}{m} - \lfloor \frac{(m+1)^k-N}{m} \rfloor$ で求める事ができ,，上記の式における k は $k = \lceil \log_{m+1}(N) \rceil$ で求める．

図 14に，各索引構造の使用メモリ空間量をバイト単位で示す．

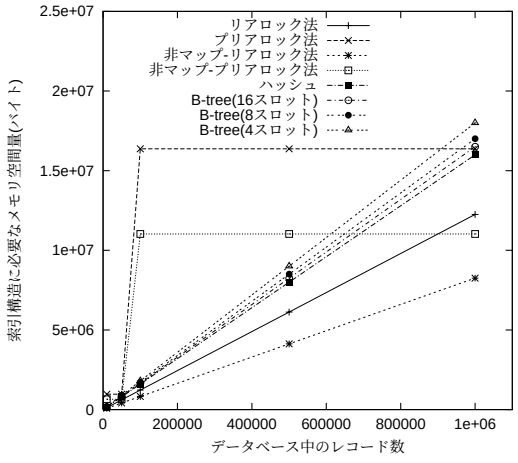

図 14 各索引構造の使用メモリ空間量

図 14より，次のことがわかる．

(1) 非マップ法はマップ法よりも使用メモリ空間量が少ない

(2) レコード数が少ないとき，非マップ-プリアロック法はリアロック法より多くのメモリを使用する．しかしレコード数が約 90 万件を越えると，非マップ-プリアロック法はリアロック法よりも少ないメモリしか使用しない．

(3) リアロック法と非マップ-リアロック法は，ハッシュおよび B-tree よりも少ないメモリしか使用しない．

(4) 非マップ-プリアロック法は，レコード数が約 70 万件を越えると，メモリ使用量がハッシュおよび B-tree よりも少なくなる．

4.4 探索時間の測定

本節では，探索処理に要する時間について述べる．

4.4.1 索 引 構 造

索引構造は，(1) ハッシュ，(2)B-tree，(3) マップ法 CSS-tree，(4) 非マップ法 CSS-tree である．ハッシュ構造では，キー値とハッシュ値が 1 : 1 で対応するようにした．B-tree 構造はノード内のスロット数が，4，8，16 である場合について探索時間を測定した．マップ CSS-tree は，において述べられている Full CSS-tree を表す．非マップ CSS-tree[3] は，第 3 章で提案した構造である．これは Full CSS-tree よりも少ない索引構造をもつ．

探索時間の測定法は，レコード件数 1 万，5 万，10 万，50 万，100 万のそれぞれのときに，ランダムにマッチするキーを選び，1 万回探索を実行した．5 回テストを繰り返し，5 回の平均をとった．キーの重複はないものとし，索引構造を構築するために要する時間は探索時間に含めない．

4.4.2 実 験 結 果

実験結果を図 15に示す．

図 15より，マップ CSS-tree および非マップ CSS-tree は B-tree より速いことと，マップ CSS-tree と非マップ CSS-tree では探索時間がほぼ同じことがわかる．したがって，非マップ-プリアロック法により構築した CSS-tree も，リアロック法により構築した CSS-tree も，探索速度がほぼ同じだと言える．

4.5 ま と め

以上の 4 つの提案手法に関する実験結果より，非マップ-プリアロック法が最も優れていることがわかった．実験結果を以下ににまとめる．

(1) 挿入処理時間

非マップ-プリアロック法を用いた際の挿入処理

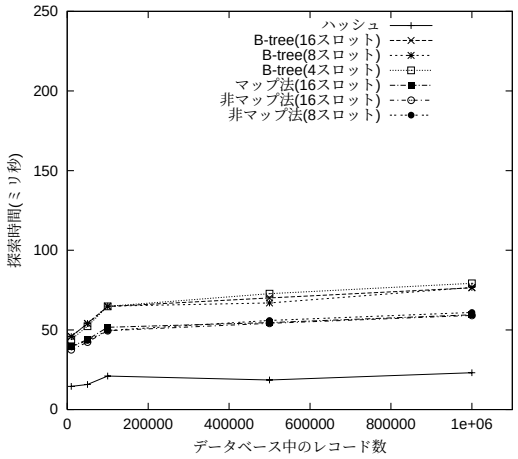

図 15　探索時間の比較

に必要な時間は最も短い．この理由は，処理に必要なステップ数が最も少ないことと，コストの高いメモリ獲得が不要だからである．

(2)　使用メモリ空間量

非マップ-プリアロック法が使用するメモリ空間量を，他の手法の使用量と比較して述べる．データベース中のレコード数が約 60 万件以下であるときには，マップ-プリアロック法に次いで多く必要としてしまう．しかしレコード件数が約 90 万件を越えると，非マップ-リアロック法に次いで少ない量しか必要としない．

これより，非マップ-プリアロック法は大規模なメモリデータベースシステムにおいて有効な手法だといえる．

(3)　探索時間

非マップ-プリアロック法により構築した CSS-tree の探索時間はオリジナルとほぼ同程度だった．これより，非マップ-プリアロック法による探索時間の劣化はないといえる．

以上より，非マップ-プリアロック法はマップ-リアロックに比べて，挿入処理時間を短縮し，使用メモリ空間量を減少させ，さらに探索時間を劣化させないことがわかった．

5. 結　　論

本研究では，更新処理を追加処理のみに限定し，CSS-tree への挿入処理を高速化する手法を 4 つ提案し，実験データベースシステム上で評価実験をおこなった．

実験の結果，4 手法のなかで，非マップ-プリアロック法が最も優れていることがわかった．この手法は，あらかじめノード領域を確保しておくことにより，(1) 配列と CSS-tree のノードを拡張するコストを抑え，(2) 配列と CSS-tree の葉ノードの再マップに要するコストを抑える手法である．

提案手法を比較した結果，非マップ-プリアロック法が最も優れており，リアロック法が最も劣っていることがわかった．非マップ-プリアロック法はリアロック法に比べて，挿入処理を最大で 7.18 倍高速化し，データベースの拡大にともなって使用メモリ空間量を抑えることができ，さらに探索時間を同程度にすることができた．

参 考 文 献

1) The TimesTen Team and TimesTen Performance Software: “In-Memory Data Management for Consumer Transactions The TimesTen Approach,” Proceedings of ACM SIGMOD International Confernce on Management of Data, May 31 - June 3, 1999.

2) J.C.Lee and K.H.Kim and S.K.Cha: “Differential Logging: A Commutative and Associative Logging Scheme for Highly Parallel Main Memory Database,” Proceedings of the 17th International Conference on Data Engineering(ICDE), pp.173-182, April, 2001.

3) Jun Rao and Kenneth A.Ross: “Cache Consicious indexing for decision-support in main memory,” Proceeding of the 25th VLDB Conference, 1999.

4) J.Rao and K. A. Ross: “Making B+-Trees Cache Conscious in Main Memory,” In Proceedings of the SGMOD 2000 Conference, pp.475-486, May, 2000.

5) Chris Nyberg: “a RISC machine sort,” In Proceedings of the SCM SIGMOD conference, pp.233-242, 1994.

6) Goetz Graefe: “Hash joins and hash teams in Microsoft SQL server,” In Proceedings of the 24th VLDB Conference, pp.86-97, 1998

7) Trishul M.Chilimbi and James R.Larus: “Improving pointer-based codes through cache-consicious data placement,” Technical report 98, 1998.