データベースシステム 128－10
（２００２．７．１７）

並列環境のためのページングＢ木における キャッシュの有効活用

小倉 匠吾　三浦 孝夫
法政大学 工学研究科 電気工学専攻

我々は、並列処理と分散データのバランス化を目的としたページングＢ木を提案している。ページングＢ木を用いた分散化では、Ｂ木を部分木から成るいくつかのページに分割する。論理的なＢ木と物理的なＢ木の対応とることで、物理的なＢ木を論理的なＢ木から独立して管理する。これにより、分散データのバランスをとるためにページを任意のプロセッサーに移動させることが出来る。今研究では、ページングＢ木におけるキャッシュの活用について論じ、実験によりその有用性を検証する。

Practical use of the Cache in Paging B-tree for Parallel Environment

Shogo OGURA　Takao MIURA
Dept.of Elect. and Elect. Engr.,HOSEI University

We propose *Paging B-tree* for the purpose of parallel database processing and well-balancing of data distribution. By the technique we can *divide* a B-tree into several pages which contain the sub-trees. We give the relationship between the logical and the physical trees. Physical B-trees are managed independent of logical ones so that we can move them into any processors to obtain well-balance of data distribution. In this work, we consider practical use of the cash in Paging B tree. And we verify the usefulness by experiment.

1　前書き

まず、分散Ｂ木でどのようにデータを分割し、各計算機にどのように配置するかを論じる。並列環境上では、データを均等に分散することが全体の処理性能の向上につながるため、データの偏りを解決する問題は、広く処理の最適化ととらえることができる。

従来の研究ではデータによって分割する方法として、値域を指定して分割する方法とハッシュ関数を用いて分割する方法が知られている。しかし、値域を指定して分割する方法はデータの分布によって各計算機間でデータ数の偏りが生じる場合がある。ハッシュ関数を用いて分割する方法では、データ数の偏りは少なくなるが、領域指定の問い合わせが出来なくなってしまう。これらの欠点を解決する方法として、Ｂ木のノード単位として分割する方法が考えられている。これはスプリットによって新たなノードが生まれたときに配置する計算機を決定する。決定方法としては、Random, Round Robin, Local Balancing[1] などの方法があるが、いずれもデータが削除された場合は保証していない。

また、各計算機にＢ木のルートから葉までの部分木を配置する FatBtree[4] という方法も考えられている。これはルートに近いほど多くの計算機がコピーを持ち、葉に近いほどコピーを持たない。データの更新は大抵の場合、葉の近くが書き換えられることから更新処理が少なくなることが期待されるが、ルートの更新が起こった場合には全ての計算機においてディスクアクセスを必要とする多大な同期のオーバーヘッドが生じてしまう。

そこで、我々は新たな分散 B 木構造としてページング B 木を提案している。既に我々は部分木自体を B 木構造を用いたページの対応付けに関する性能評価 [7] において、その有用性を示している。

今研究では、ページング B 木におけるキャッシュの活用について論じる。またルートへのアクセスが集中の問題にも対応する。そして実験によりそれらの有用性を検証する。ページング B 木ではデータのコピーを持たないので、並列環境においても単一環境と変わらぬキャッシュの効果が見込めることから、単一環境での実験を行なった。

2 章では部分木ページング B 木の説明とその処理の流れを述べる。3 章でページング B 木におけるキャッシュの活用について論じる。4 章では実験によりその有用性を検証し、5 章で結びとする。

2 ページング B 木

2.1 構造

部分木ページング B 木では、B 木を部分木単位に分割し並列処理を行う。この方法はデータの重複が起きないのでデータの同期の必要がなく、領域指定の問い合わせにも対応している。また、論理的な B 木を物理的な B 木によって仮想化することで、任意に物理位置を変換可能となり、データの偏りの問題に対応している。

部分木ページング B 木では特定した深さの部分木に分割する。分割した部分木は各々独立して管理する。あるノードの次にアクセスされるのはその下層のノードとなるので、それらを一つのページとして処理する。例えば深さ 6 の B 木を深さ 2 の部分木に分割すると図 1 のように 3 レベルに分割される。

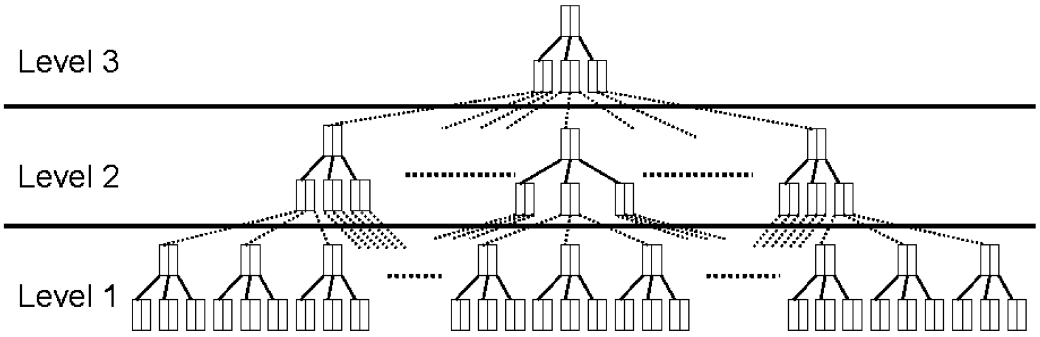

図 1: 部分木分散

ページング B 木は部分木の物理キーと論理キーの対応をとることで、B 木の仮想化をしている。これによりページの物理位置を変更しても物理キーが変更されるだけで論理的な B 木構造を維持することができる。ここでは論理キーを部分木のルートノードの先頭の値（ルートキー）と部分木の配置されている高さ (レベル) の組み合わせとし、物理キーをホスト名とホスト内位置の組合せとする。各部分木には図 2 のようなページマップが用意されている。ページマップは下層部分木の物理キーのリストである。部分木の葉からのポインタは対応するページマップの行を指している。図 2 において”DS”を論理キーとする下層部分木の物理キーは、上層部分木において論理キー”DS”を検索することで上層部分木のページマップ内から得ることが出来る。このようにしてキーの対応を取ることが出来る。

2.2 検索

ページング B 木における検索の手順を説明する。例えば図 2 で値”DZ”を検索するには、まずヘッダ情報として保持している根部分木の物理キーより根部分木を読み込む。そして根部分木で”DZ”を検索すると、

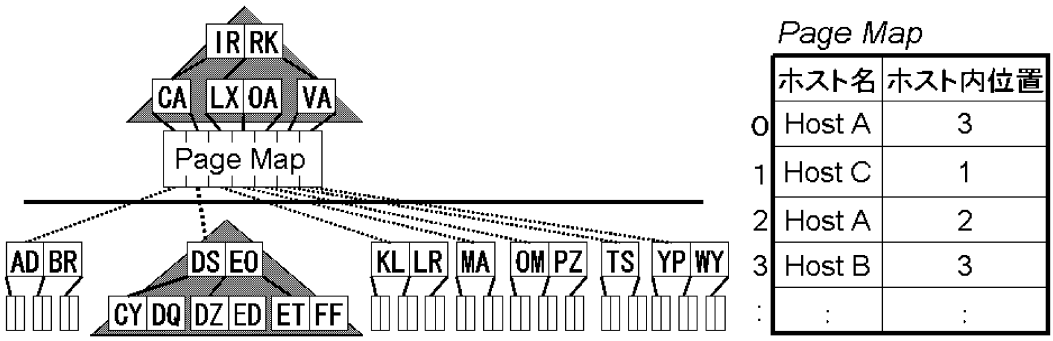

図 2: ページマップ

ノード [CA] の 2 つ目のポインタに辿り着きページマップ内の物理キーを得る。最後に得られた物理キーの下層部分木を読み込み、”DZ”を検索すると値”DZ”が発見できる。

2.3 挿入

図 2 の状態に”EA”を挿入した場合の説明をする。このページング B 木の位数は 1 とする。まず”EA”の挿入場所を検索する。挿入場所は上述の例で検索した”DZ”と隣の”ED”の間である。挿入を行うとこのノードはオーバーフローしてスプリットが起こり、”EA”が部分木のルートノードに挿入される。ルートノードでもオーバフローが起こりスプリットが起こる。部分木のルートノードがスプリットする場合には、部分木がスプリットを起こす。ルートノードのスプリットで新しく生まれたノード [EO] がルートノードとなる部分木が生まれる。そして”EA”が上層の部分木に送られる。上層では新しい部分木の物理キーがページマップに挿入され、部分木では”EA”の挿入場所を探して挿入する。その際に新しいページマップの行もポインタとして挿入される。挿入後の部分ページング B 木の状態を図 3 に示す。

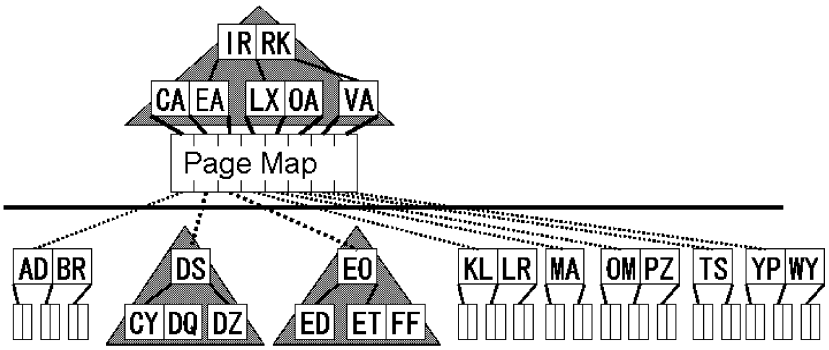

図 3: 部分木スプリット後

2.4 データ移転

まずデータの更新により新しく生成される部分木の配置について述べる。スプリットにより新しい部分木が生まれた時には元の部分木と同じホストに配置する。これは任意に部分木配置を変更できるので、データの転送の必要ない同じホストに配置するのが最も効率が良いからである。この方法では、まず一つのホストに全てを配置し、そしてあるタイミングで任意のホストに振り分けることになる。その後も任意のタイミングで移動させることが可能である。ページング B 木で全ての部分木が自由に任意のホストに移動す

ることが可能なのは、キーの対応を取ることで論理的なB木構造を保持しているからである。データを移動させることで、分散データのバランスさせることが可能である。

またデータを任意のホストに移動可能ということは、データ順序 (辞書式順序) 以外の物理配置も可能になる。例えばキーが住所になっていて、”トウキョウ*”の多い部分木と”カナガワ*”の多い部分木を近くに配置したり、”トウキョウ*”の多い部分木と”トクシマ*”の多い部分木を遠くに配置するなどの地理的順序の配置ができる。

3　キャッシュの活用

ここではページングB木でのキャッシュの活用について考える。既に述べたとおり分割された部分木がページである。これにはページマップも含まれる。また、根部分木へのアクセス集中の問題もキャッシュによって対応していく。

3.1　キャッシュ方法の決定

ページングB木ではページの処理の結果によって次にアクセスするページが決定するので、先読み（Read Ahead）やパイプライン化には不向きである。そこで読み込んだページを保存して再利用する手法を取る。次にページ置き換えアルゴリズムを考えていこう。B木は根に近いほどアクセス頻度が高く葉に近いほどアクセス頻度が低いが、Range Queryにより連続データが参照される。また、一時的にアクセスが集中することも予測される。これらのことから、参照される頻度が最も低いページを書き出すLFU (Least Frequently Used) 方式や一番古くからあるページを書き出すFIFO（First In First Out）方式ではキャッシュヒット率が悪くなるであろうと予想され、最後に参照されてからの経過時間が最も長いものを書き出すLRU（Least Recently Used）方式が有効であると考えられる。

3.2　根部分木キャッシュ

根部分木には問い合わせに対し必ず最初にアクセスが必要であり、極端にアクセスが集中してしまう。この対策として、根部分木のキャッシュを全ホストにコピーする。そして問い合わせのスタートは各ホストをローテーションすることでアクセスを分散させる。各ホストがコピーを持つことになるので同期が必要となる。根部分木を更新するには、まずキャッシュを施錠（ロック）するためにブロードキャストで各ホストに知らせる。そしてデータを更新し、ブロードキャストで各ホストに更新データを送り、最後にキャッシュを更新して解錠（アンロック）する。同期に必要なコストはブロードキャスト2回と各ホストの施錠・解錠とキャッシュの更新となる。各ホストでのディスクアクセスが無いことから比較的少ないコストであると言えるだろう。また根部分木の更新頻度は全部分木中最小であると予測される。

4　実験

4.1　実験の狙い

ここでは、単一環境で動作しているキャッシュを用いたページングB木による実験を行い、先に論じた方法の有用性を検証する。まず、ページングB木におけるLRU方式のページングリプレースメントの有用性を検証するためにキャッシュヒット率を測定する。そして、根部分木キャッシュを全ホストにコピーする手法のコストとなる根部分木の更新頻度を測定する。

4.2 実験手順

今回用いるページング B 木のキーは整数（int 型），位数は 6，部分木の深さは 2 である。このページの大きさは 3628Byte となる。ページング B 木にはあらかじめランダムな 10^5 件のデータを挿入しておく。これにより総ページ数が 1531 の 3 レベルのページング B 木となった。これに対して 10^4 回のランダム検索を行いキャッシュヒット率を測定する。検索は Range Query で値の幅は $10^0 \sim 10^3$ と変化させた。また、キャッシュサイズはページ 10 個とページ 100 個でそれぞれ測定した。その後、ランダムな 5×10^4 件の挿入を行い根部分木が更新される回数を測定した。

4.3 実験結果

検索の結果を表 1 に示し、そのキャッシュヒット率を図 4 のグラフに示す。レンジ幅が広がるほどキャッシュヒット率が高くなっているのが分かる。また、キャッシページ数が 10 よりも 100 の方がキャッシュヒット率が高いのが確認できる。次に挿入による結果を表 2 に示し、その根部分木更新頻度を図 5 のグラフに示す。根部分木更新頻度は、一定して低い頻度であることが分かる。

キャッシュページ数	10			
レンジ幅	1	10	100	1000
総ページ参照回数	29749	299571	2997344	29975466
キャッシュヒット回数	17176	286523	2981913	29937256
キャッシュヒット率	57.74 %	95.64 %	99.49 %	99.87 %
キャッシュページ数	100			
レンジ幅	1	10	100	1000
総ページ参照回数	29749	299571	2997344	29975466
キャッシュヒット回数	24651	294113	2990032	29952589
キャッシュヒット率	82.86 %	98.18 %	99.76 %	99.92 %

表 1: 10^4 回検索結果

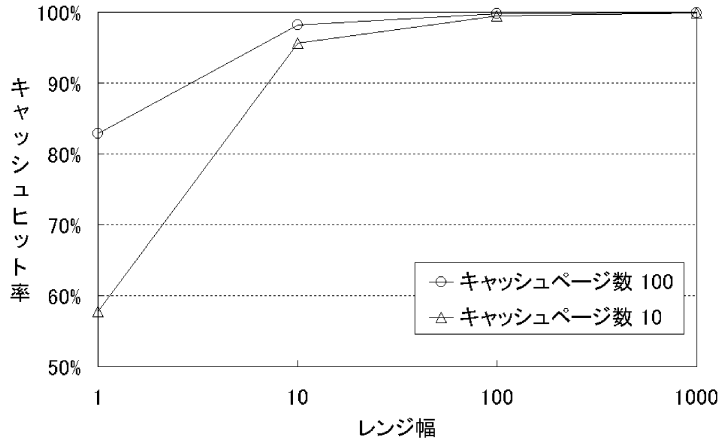

図 4: キャッシュヒット率

挿入件数	10000	20000	30000	40000	50000
根部分木更新回数	2	5	7	10	13
根部分木更新頻度	0.020 %	0.025 %	0.023 %	0.025 %	0.026 %

表 2: 5×10^4 件挿入結果

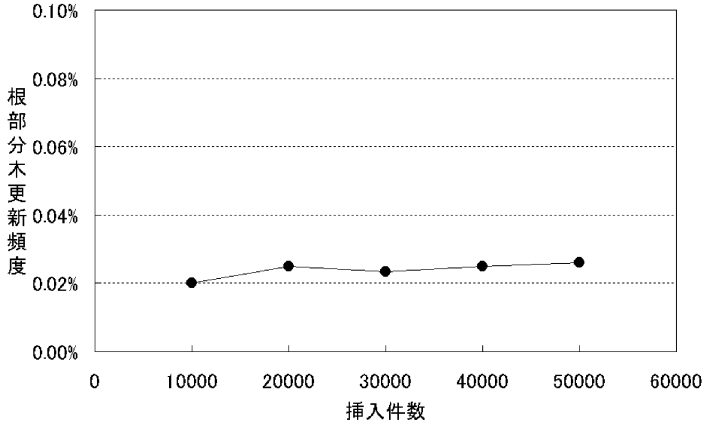

図 5: 根部分木更新頻度

4.4 考察

まずキャッシュヒット率を見るとレンジ幅 10 以上では全て 95%以上の高いポイントとなっている。これは Range Query の連続データの参照がほぼ同じパスを通って検索するので、LRU 方式のページ置き換えではパスの全てのページがキャッシュされているためであり、非常に有効であることが分かる。レンジ幅 1、つまり Single Qurey においても、ページ数 1531 に対しキャッシュページ数 10 とキャッシュ率 0.65%の非常に少ないキャッシュサイズでも 50%以上の確立となり、その有用性が分かる。キャッシュページ数 100 を見るキャッシュヒット率 80%と非常に高い確率ながらキャッシュ率は 6.53%と僅かであることからも、その有用性が見て取れる。

根部分木更新頻度を見ると平均 0.024%と低い値で一定している。実際に更新が起きたとしても、先に述べたようにそのコストは比較的少ないであろう。また、根部分木に対するアクセスが 1 ホスト集中していたのが各ホストに均等に分散されるメリットはに比べれば、僅かなコストであると言えるだろう。

5 結び

現在、我々はページング B 木の MPI[3] を用いた並列環境での実装を行っている。本稿ではページング B 木におけるキャッシュの活用を提案した。そして単一環境での実験により高いキャッシュヒット率を得ることで、並列環境においける有用性も予測できた。また根部分木キャッシュのコピーによって根部分木へ集中するアクセスを分散することが僅かなコストで実現できることが予測できた。

参考文献

[1] B.Seeger and P.Larson: ”Multi-Disk B-tree”, proc. ACM *SIGMOD* Conference 1991, pp.436-445

[2] PVM (Parallel Virtual Machine) , http://www.epm.ornl.gov/pvm/

[3] MPI (Message Passing Interface) , http://www-unix.mcs.anl.gov/mpi/

[4] H.Yokota, Y.Kanemasa, and J.Miyazaki: "Fat-Btree: An Update-Conscious Parallel Directory Structure" , proc. IEEE *ICDE* 1999, pp.448-457

[5] T.Miura, W.Matsumoto, I.Shioya, and Y.Wada, "Extensible Perfect Hashing", proc.ACM *CIKM* Conference 2000

[6] S.Watanabe, and T.Miura, : "Reordering B-tree", proc.ACM *SAC* Conference 2002, to appear

[7] S.Ogura, and T.Miura, : "Paging B-Trees for Distributed Environment", proc.ACM *WDAS* Conference 2002, to appear