修 士 論 文

並列データベースシステムにおける
更新を考慮したディレクトリ構成に関する研究

指導教官　横田治夫 助教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

金政 泰彦

1998 年 2 月 13 日

要 旨

無共有並列計算機上のデータベースシステムにおいて、処理負荷をできるだけ均等化させながら、絞り込みによる高速アクセスを実現するための手法として、Fat-Btree という新しい並列ディレクトリ構成法が提案されている。Fat-Btree では、検索のみでなく、更新処理にも注目し、更新によりディレクトリ構造に変化が生じる場合でも、並列プロセッサ間での同期をできるだけ抑えることを目指している。本研究では、この Fat-Btree の特性について検討するとともに、これまでに提案されていた B-tree 全体を全プロセッサにコピーする方式との比較を行ない、確率的な解析によりスループットもレスポンスタイムも Fat-Btree 方式が優れることを示す。また本研究では、Fat-Btree を実際に実装する方法に関する検討も行ない、通常の B-tree との違いに起因して実装の際に生じる選択肢について考察する。

目 次

図 目 次

表 目 次

第 1 章

序論

1.1 本研究の背景と目的

データベースシステムの処理能力向上のため、並列マシン上にデータベースシステムを構築することが多くなってきている。特にスケーラビリティの面から、メモリもディスクもプロセッサ間で共有しない無共有 (Shared-Nothing) 型の並列マシンを用いたデータベースシステムの研究/開発が多数行なわれている [2]。

無共有型並列マシン上のデータベースシステムでは、処理はプロセッシングエレメント (PE) 間のメッセージ通信によって進められ、データベースの検索/更新処理は、参照されるデータが格納されている各 PE 上で並列に実行される。一方、並列システムでハードウエア量に応じた性能向上を望むためには、各 PE 間で負荷を均等化することが重要である。負荷に偏りがある場合には、偏った負荷が割り当てられた PE の処理時間に全体の処理時間が依存することになる。このため、各 PE 間の負荷を均等にするための研究がこれまで多数行なわれてきている [5]。

データベース処理の負荷の偏りには、処理の途中で生成されるデータ量の偏りも含めて色々な原因があるが、負荷バランスにまず第一に影響するのは、処理対象となるデータの配置によるものである。特に、1 つのデータベース演算を並列化するために、データベースを水平に分割して配置し、分割単位で処理を並列に行なう場合には、データベースの分割方法が問題となる。

これまでに考えられてきたデータの水平分割の主な手法は、値域分割、ラウンドロビン分割、ハッシュ分割の 3 種類である [2]。値域分割は、分割の基準値を静的に決定するの

で、分割当初は偏りが少なくても、更新を重ねることによりデータ量の偏りが大きくなるという欠点がある。一方、ラウンドロビン分割は更新によるデータ量の偏りを生じることはないが、いかなる問い合わせに対しても全てのディスクにアクセスにいく必要が生じて、典型的な問い合わせに対するディスクの絞り込みができない。当然、値域分割は絞り込みが可能である。この両者の間を取ったのがハッシュ分割で、絞り込みが可能で、更新によるデータ量の偏りも少なくなる。しかしハッシュ分割は、値域分割では可能である範囲を指定するような問い合わせや、連続アクセスのディスク I/O 回数を削減するためのクラスタ化アクセスには対応できない。そこで、それらを組み合わせたデータ分割方式等も提案されている [1, 3]。

データ水平分割と同時に、PE 内の高速アクセスのためにインデックスも重要となる。そこで、値域分割の長所を生かしながら、高速アクセスを実現する方法として、並列データベースに B-tree を用いる方法が提案されている [4]。インデックスとして B-tree を構築すると同時に、データをページ単位で分配し、B-tree を全 PE にコピーして、自 PE 内の高速アクセスに用いる。B-tree を用いることにより、値域分割の欠点であった更新によるデータ量の偏りにも、B-tree のページの移動によって、ある程度対応することができるようになる。

しかし、そのような並列ディレクトリ構造では、ディレクトリ構造自身の更新コストには、注意を払っているものが少ない。特に、文献 [4] のように、全ての PE に B-tree のディレクトリ全体をコピーして配置すると、ディレクトリ更新時に全 PE への同時アクセスが必要となり、それがシステムのスループットを大きく低下させてしまうことになる。一方、全 PE 同時更新を避けるために 1 つの PE のみに B-tree の全ディレクトリを配置すると、その PE にアクセスが集中し、そこがボトルネックになってしまう。

そこで本研究では、Fat-Btree[7] という新しい並列ディレクトリ構成方式を利用する。Fat-Btree は、B-tree 構造上での更新頻度と分散配置との関係から、コピーを木構造の根に近い部分に絞ることにより、ディレクトリ更新時の PE 間の同期をできるだけ抑えることを目指している。本論文では、この Fat-Btree の特性の検討を行なうとともに、確率を基にした解析によって、文献 [4] の B-tree のディレクトリ全体を全ての PE にコピーする方式との性能比較を行ない、Fat-Btree が従来の並列ディレクトリ構成方式よりもスループット、レスポンスタイム共に優れることを示す。また本論文では、Fat-Btree を実際に実装する方法に関する検討も行ない、通常の B-tree との違いに起因して実装の際に生じ

る選択肢について考察する。

1.2 本論文の構成

本論文の構成は以下の通りである。第 2章で、並列データベースシステムにおけるデータの水平分割手法と、並列 B-tree に関する従来の研究について述べる。第 3章では、本論文で扱う新しい並列ディレクトリ構成方式である Fat-Btree について説明するとともに、その特性に関する考察を述べる。第 4章では、Fat-Btree の有効性を検証するために、確率に基づく解析によって、Fat-Btree と従来の手法との性能比較を行ない、その結果について考察する。そして第 5章では、Fat-Btree を実際に実装する方法に関する検討を行なう。最後に第 6章で、まとめと今後の研究課題について述べる。

第 2 章

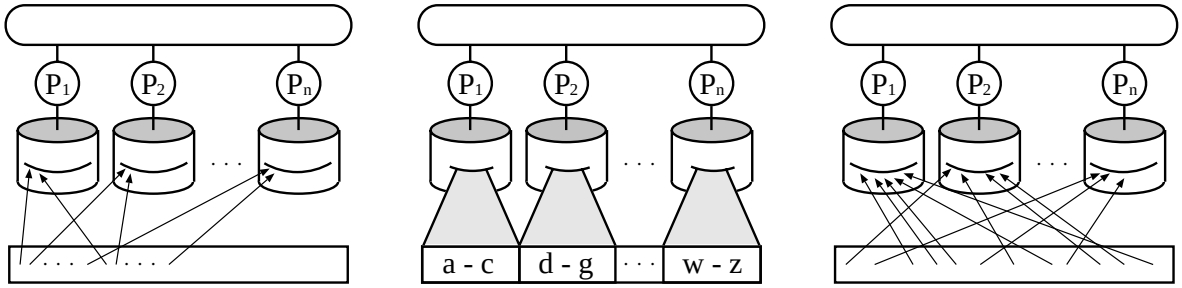
並列データベースシステムにおけるデータ分配

2.1 従来のデータ分割手法

並列データベースシステムでは、データベースを水平に分割して、それらを並列計算機の各 PE に分散配置することによって、データベース演算を並列に行なうことができ、速度向上や規模向上が可能となる。しかしこの場合、データの分割手法が性能向上の非常に大きな鍵となる。従来のデータ分割手法には、以下のようなものがある [2]。

ラウンド ロビン分割 この手法では、タプルを単純にラウンド ロビン風に割り当てる (図 2.1(a))。データを分配する PE の総数を p とすると、i 番目のタプルは、PE($i \bmod p + 1$) に割り当てられる。この手法では、PE 間でのデータ配置の偏りは最大でも 1 タプルに抑えられ、これはデータの挿入/削除が行なわれても変わらない。それゆえに、この手法は、リレーションを順にスキャンして全部にアクセスする問い合わせでは、負荷を各 PE 間で均等にすることができ、優れた性能を示す。しかし、タプルの特定の属性値を指定して連想的にアクセスする問い合わせに対しては、タプルの格納されている PE を絞り込むことができず、常に全 PE を使っての検索が必要となる。

値域分割 この手法では、キーの値を p の間隔 $I_1, .., I_p$に分割する区切りを定める。そして、キーの値が I_jにあるタプルを PEjに格納する (図 2.1(b))。そうすることによっ



(a) ラウンドロビン分割　　(b) 値域分割　　(c) ハッシュ分割

図 2.1: 3 つのデータ分割手法

て、この手法では、単一タプルにアクセスする問い合わせは 1 つの PE に制限され、範囲を指定した問い合わせはその大きさに応じていくつかの PE で処理される。また、近い属性値を持つタプル同士を同じ PE 内でクラスタ化して、連続アクセスを行なう問い合わせにおいてディスク I/O の回数を削減することもできる。しかしこの手法では、分割の基準値を適切に決められなければ、データ配置に偏りが生じてしまう。また、もし分割の基準値を適切に決めることができて、初期状態におけるデータ配置を PE 間で均等にすることができたとしても、分割の基準値を静的に決めてしまっているので、データの挿入/削除が繰り返される間にデータ配置に偏りが生じてしまう可能性がある。

ハッシュ分割 この手法では、ハッシュ関数がタプルのキーに適用され、その結果に応じてそのタプルが配置される PE が決まる (図 2.1(c))。ハッシュ関数を適切に選ぶことによって、この手法ではデータを各 PE に均等に分配することができる。またこの手法では、タプルの特定の属性値を指定した連想的なアクセスにおいて、データの存在する PE を一意に特定することもできる。しかしハッシュ分割では、範囲を指定した問い合わせは、該当するタプルが全ての PE に存在する可能性があるので絞り込みができず、たとえそれが小さな範囲の問い合わせであっても、常に全 PE での検索を必要とする。これは、小さな範囲の問い合わせに関して、値域分割がその問い合わせの実行を 1PE だけに抑えられるのと比較して、PE 間での通信コストためにレスポンスタイムが低下するだけでなく、余計な負荷を生成するのでシステム全体のスループットも低下することになる。また、近い値のタプルが全く別々の

PE に配置されるので、クラスタ化アクセスによって連続アクセスのディスク I/O 回数を削減することもできない。

値域分割では、各 PE 間でデータ配置が均等になっていたとしても、問い合わせによって参照されるデータが少数の PE に集中していて、問い合わせ処理がその少数の PE に集中する**実行の偏り** (execution skew) を起こす危険性がある。それを避けるために、値域分割において、非一様分散 (non-uniformly-distributed) 分割基準を用いる研究がなされている。例えば Bubba では、値域分割によってリレーションを多数の小さなフラグメントに分割し、そのフラグメントを各 PE 間でサイズ (タプル数) ではなくアクセス頻度 (**heat**) が等しくなるように、各 PE に割り付けるという方法を採っている [1]。また、Ghandeharizadeh らは **hybrid-range partitioning strategy (HRPS)** という手法を提案している [3]。HRPS でも、リレーションは小さなフラグメントに分割され、そのフラグメントはラウンドロビン風に各 PE に配置される。ただし HRPS では、リレーションにアクセスする問い合わせの資源要求とシステムの処理能力から、1 つのフラグメントの大きさを最適化している。これによって HRPS では、小さな範囲の問い合わせの実行は少数の PE だけに絞り、大きな範囲の問い合わせの実行には多くの PE を用いて実行の偏りを防ぐ、ということが可能となっている。Bubba の手法と HRPS の問題点は、両方式ともフラグメントへのアクセス頻度や問い合わせの資源要求といった動的な要素を、予め適切に予測できることを前提としていることである。また、これらの方式では、その多数のフラグメントの分配という方式上、データへの高速アクセスを提供するためにはインデックスの構成に工夫が必要となると思われるが、それに関しては特に考慮されていない。

2.2 並列 B-tree

並列データベースシステムでは、データベースの水平分割法と同時に、PE 内の高速アクセスのためのインデックスも重要となる。そこで、高速アクセスを提供するデータ構造である B-tree を並列化して、並列データベースシステムのインデックスとして用いることが研究されている [4]。また、B-tree 構造は単にインデックスとしてだけではなく、ページ単位に分けられたその構造をそのままデータ分配に利用することもできる。以下で、その並列 B-tree について説明する。

2.2.1 B-tree 並列化の手法

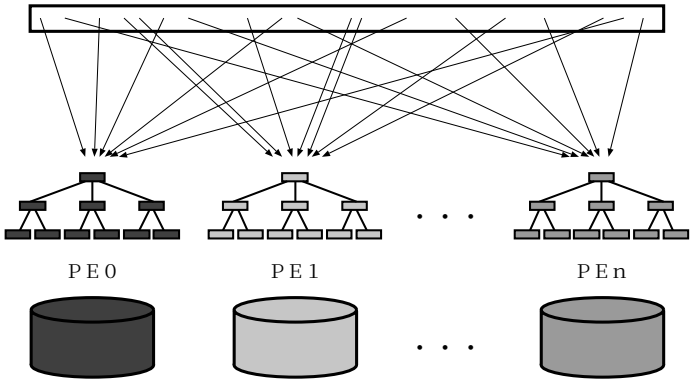
並列データベースシステムでインデックスとして用いるために B-tree を並列化する手法には、大きく分けて以下のような 3 通りがある。

レコード分配 この手法では、各 PE あたりに 1 つの B-tree を用意し、タプルをそれらの B-tree に分配する (図 2.2(a))。タプルを各 PE に割り当てる方法は、節 2.1 で説明したデータ分割法のいずれを用いても構わない。この方法では、B-tree は PE 内での高速アクセスのためのインデックスとしてのみ用いられており、データ分割には関与しない。よって、データ分割に用いた手法の長所短所が、そのままこの手法の長所短所になる。もしデータ分割に値域分割を用いると、データの挿入によってデータ配置に偏りが生じた際に、データ配置の均等化を行なうためにはデータの再配置のコスト以外に B-tree 再構築のコストもかかるので、データ偏り制御は非常に困難となるまた、もしデータ分割にハッシュ分割を用いると、小さな範囲の問い合わせにおいても、全ての PE で B-tree の検索を行なう必要が生じ、レスポンスタイムもシステム全体のスループットも低下してしまう。

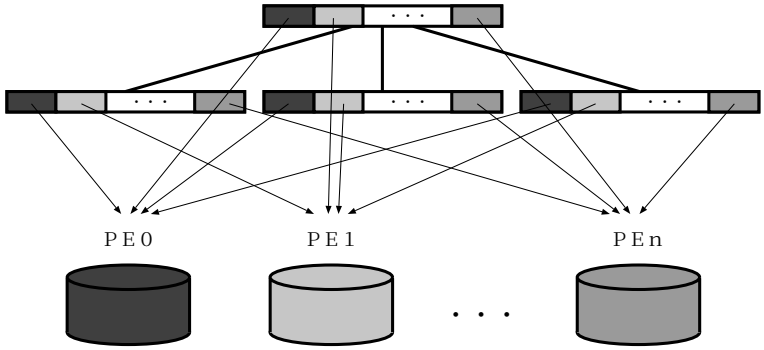
巨大ページ この手法では、とても大きなページ (ここではスーパーページと呼ぶ) を使用し、各ページを p 個の PE に渡って分割する (図 2.2(b))。したがって、スーパーページは p のサブページから構成され、それぞれのサブページは別々の PE に格納される。スーパーページの検索には、1PE あたり 1 ディスクアクセス、全体で p のディスクアクセスを必要とする。この手法の主な利点は、1 つのページが大きくなるので通常の B-tree に比べて木の高さが低くできることと、データがクラスタ化されている 1 つの B-tree ページを全ての PE で並列に読み込むので、大きな範囲の問い合わせにおいてレスポンスタイムが向上することである。しかし、この手法では小さな範囲の問い合わせはもちろん、単一タプルにアクセスするオペレーションでも、p 個の PE 全てを使う必要がある。これは、一番処理の遅い PE を待つ必要性からレスポンスタイムを悪化させるだけでなく、無駄に資源を消費するのでシステムのスループットを非常に低いものとする。

ページ分配 この手法では、通常の B-tree を用いて、そのページを p 個の PE 間に分配する (図 2.2(c))。ページの分配方法にはランダム分配とラウンドロビンがある。ランダムなページ分配では、分割によって生成された新しいページは、ランダムに PE

（a）レコード分配 (ハッシュ分割使用)



（b）巨大ページ



（c）ページ分配

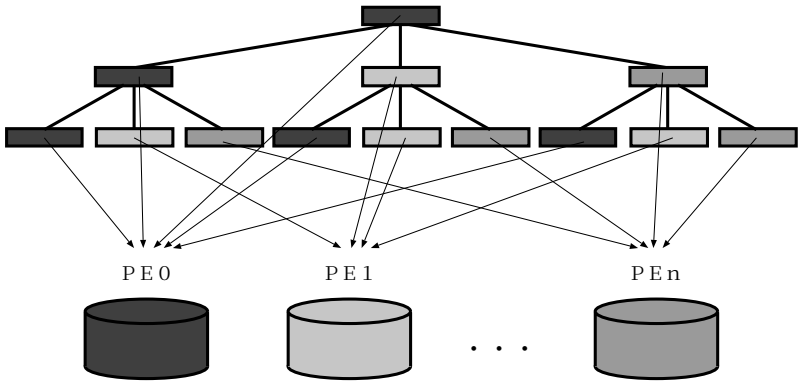


図 2.2: 3 つの B-tree 並列化手法

の内の 1 つに割り当てられる。この分配法は良い負荷バランスを提供し、どの基本オペレーションも通常の B-tree 以上のディスクアクセスを必要としない。しかし、この分配法ではシステムのスループットは良くなるが、局所的な負荷バランスの保証がないので、最悪の場合は要求されるページが全て 1 つの PE 上に存在し、範囲の問い合わせなどでレスポンスタイムが悪くなる可能性がある。一方、ラウンドロビンによるページの分配では、各 PE に割り当てられているページの数のトラックを保持していて、各ディスクのページの数のバランスをとりながら、系統的な方法でページを割り当てる。ページの削除がないときには、この分配法は完全なデータのバランスを成し遂げる。しかし、この方法でもページの削除や挿入が行なわれたときに、やはり局所的な負荷バランスの保証がなくなってしまう。

商用の並列データベース管理システムである Oracle8 では、値域毎に B-tree を分割して PE に置くことをサポートしている [8]。これは値域分割を用いたレコード分配型並列 B-tree であるが、この方法は前述の通り、値域分割の欠点であるデータ配置の動的な偏りに対処できない。

レコード分配と巨大ページという並列 B-tree の両アプローチでは、1 つ以上の基本オペレーションに関して不必要に高い資源消費があって、それがシステムのスループットを低下させてしまう。一方、ページ分配による並列 B-tree には、そのような欠点はないように思われる。しかし、ページ分配型並列 B-tree では局所的な負荷バランスの保証はしていないので、その結果として範囲の問い合わせにおいてレスポンスタイムが低下する可能性がある。

Seeger らは、局所的な負荷バランスを実現するページ分配型並列 B-tree として、LOB-tree を提案している [4]。LOB-tree では、連続する $\left\lceil \frac{p}{2} \right\rceil$ 以下のデータページの中のどの 2 つのページも同じ PE に格納されないことを保証するので、範囲の問い合わせにおいて負荷バランスが保たれる。しかしこの LOB-tree では、上位のインデックスページは、コピーして全ての PE に配置されている。これでは、その上位インデックスページを書き換えるようなディレクトリの更新が発生したときに、全ての PE がその更新処理に巻き込まれ、それによってシステムのスループットが低下してしまう。

ページ分配型並列 B-tree は、3 つの B-tree 並列化手法の中で唯一、全ての基本オペレーションにおいて問題が少ない手法で、並列 B-tree の実現において最も好ましい手法であると思われる。しかし、この手法では上記のように、ディレクトリの配置場所によって問

題が生じることがある。次節では、ページ分配型並列 B-tree におけるディレクトリ配置場所の問題について、さらに詳しく述べる。

2.2.2 ページ分配型並列 B-tree のディレクトリ配置

ページ分配型並列 B-tree は、他の B-tree 並列化手法と違って、ディレクトリを PE へ配置する方法の選択に自由度がある。例えばレコード分配型並列 B-tree では、ディレクトリは各 PE 毎に構成され、巨大ページ型並列 B-tree では、各インデックスページが分割されて PE 間に分配される。しかし、ページ分配型並列 B-tree では、データページは各 PE 間に分配されるが、インデックスページについては、PE 間へ分配する以外の配置法を選択することも可能である。

ページ分配型並列 B-tree のディレクトリ配置法の選択肢には、次の 3 通りがある。

B-tree 全コピー方式 この手法では、B-tree のディレクトリはコピーされて全 PE に配置される。この場合、ディレクトリの検索は各 PE で独立して行なえるので、問い合わせ実行の並列性が上がり、システムのスループットは良くなるように思われる。しかしこの手法では、B-tree のページがスプリットを起こし、B-tree のディレクトリの構造が更新されるときには、全ての PE で同期してその更新処理を行なう必要が生じる。それゆえに、頻繁にディレクトリが更新されるような場合には、増加するメッセージ通信やディスクアクセスのために、システムのスループットが低下する。

ディレクトリ集中配置方式 この手法では、B-tree のディレクトリを 1 つの PE のみに配置する。データにアクセスする問い合わせは、まずディレクトリの置いてある PE で B-tree の検索を行ない、データの格納されている PE 番号とその PE 内での物理ページ番号を調べる。それから、データの格納されている PE で、実際のデータの参照を行なう。この方法はディレクトリがコピーを持たないので、確かにディレクトリ更新処理に必要なコストは低いかもしれない。しかし、データにランダムアクセスする問い合わせは全て、1 つの PE に置かれているディレクトリを参照するので、その PE にアクセスが集中し、そこがボトルネックとなってシステムのスループットが低下する。

インデックスページ分配方式 この手法では、インデックスページもデータページと同様に各 PE 間に分配する。この方法が、本来のページ分配型並列 B-tree のディレクト

リ配置法である。この方法では、ルートページが 1 つの PE に置かれることになるので、やはりその PE にアクセスが集中してしまう。また、ディレクトリが PE 間に跨っているので、B-tree の 1 段毎に別々の PE でページの検索を行なうことになり、必要となるメッセージ通信のためにレスポンスタイムが悪化する。

ページ分配型並列 B-tree の本来のディレクトリ配置法は、インデックスページ分配方式であるのだが、上で述べたようにこの方法は、ルートページの置かれている PE にアクセスが集中するし、レスポンスタイムも悪くなるので非現実的である。B-tree において、ランダムアクセスを行なう全ての問い合わせはルートページを参照するので、ルートページを 1 つの PE のみに置くと、その PE にアクセスが集中することは避けられない。ゆえにページ分配型並列 B-tree では、ルートページを始めとする上位インデックスページは、コピーして複数の PE に置くのがよいと考えられる。

2.3 本研究のアプローチ

並列データベースにおけるデータの水平分割に、ページ分配型並列 B-tree を用いることは、非一様分散分割基準を用いた値域分割と同様の長所を示す。データはページ単位でまとめられて PE に配置されるので、近い属性値を持つタプル同士は物理的にも近く配置され、小さな範囲の問い合わせに効率的に対応できる。また、値域分割の欠点であったデータ配置の偏りには、ページの分配法の工夫や PE 間でのページ移動で対応できる。さらに、ページ分配型並列 B-tree は、B-tree 構造なので当然のことながら高速検索が可能になる。

ページ分配型並列 B-tree で問題であるのは、そのディレクトリの配置場所である。ページ分配型並列 B-tree では、ディレクトリは、少数 PE へのアクセス集中を避けるために、コピーして複数の PE に置く必要がある。しかし、ディレクトリのコピーを複数 PE に配置すると、ディレクトリの更新時に、その複数 PE 全てで同期して更新処理を行なう必要があり、これがシステムのスループットを低下させることになる。

本研究の Fat-Btree は、ページ分配型の並列 B-tree である。Fat-Btree では、B-tree 構造上での更新頻度と分散配置との関係から、ディレクトリのコピーを木構造の根に近い部分に絞っている。アクセス頻度の高い上位インデックスページは、少数 PE へのアクセス集中を防ぐためにコピーして多数の PE に分配して、更新頻度の高い下位インデックス

ページは、ディレクトリ更新時の PE 間の同期を抑えるために殆んどコピーを持たないようにしている。これによって Fat-Btree では、ページ分配型並列 B-tree の長所を保ったまま、ディレクトリ更新の多い状況でのシステムのスループット低下を避けることを狙っている。

第 3 章

Fat-Btree

3.1 概要

Fat-Btree は、ページ分配型の並列 B-tree である。その葉ページ (最終インデックスページ) は、値域分割風に各 PE に均等に分配される。Fat-Btree が他のページ分配型並列

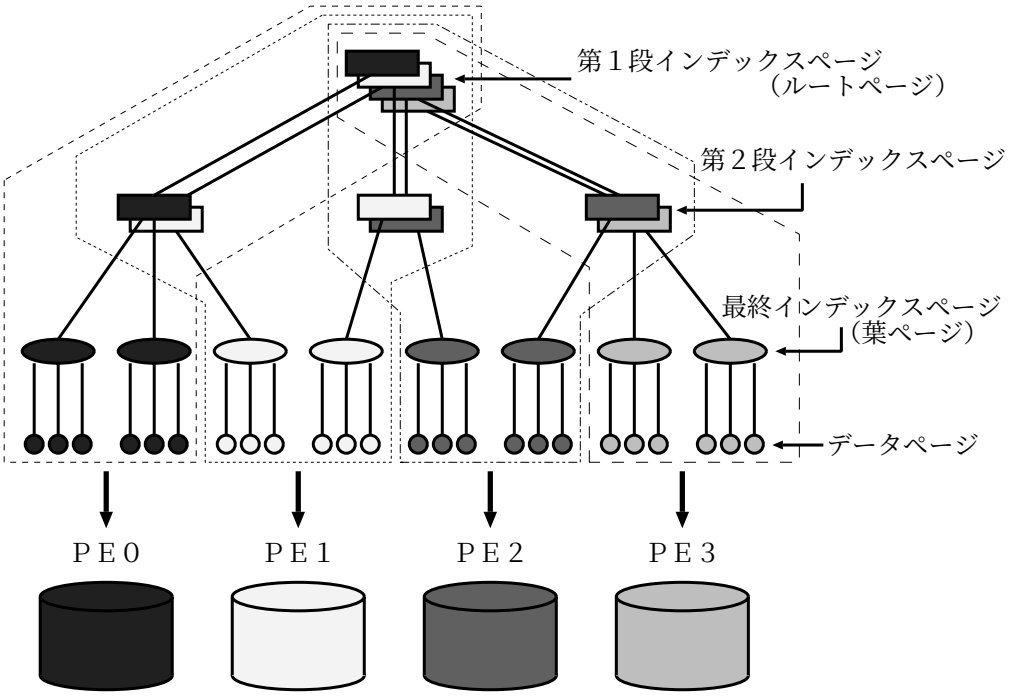


図 3.1: Fat-Btree 構造

B-tree と決定的に異なるのは、そのディレクトリの配置法である。コピーされるのを上位のインデックスページに限るために、B-tree の葉ページ以外のインデックスページは、各 PE に配置されている葉ページの再帰的に上位にあたるページのみを、その同じ PE に配置するようにしている。これによって、Fat-Btree において各 PE には、B-tree のルートページから各 PE に均等に分配された葉ページまでの部分木が格納される (図 3.1)。

Fat-Btree では、多くの葉ページへのパスに共有される比較的上位のインデックスページは、コピーされて多数の PE に配置される。特にルートページは、全ての PE にコピーされることになる。しかし、ディレクトリの更新によって書き換えられるのは、大抵の場合は比較的下位のインデックスページで、それらのインデックスページは少数の PE にしか配置されない。また、ルートページからその PE に配置されている葉ページまでの全てのパスはその PE 上にあるので、各 PE 毎にデータの検索が可能で、特定の PE への処理の集中を避けることができる。

3.2 Fat-Btree の性質

3.2.1 ページの更新/参照頻度とコピー数

B-tree 構造において、データベースでの更新により変更が起こるのは、葉ページの部分である。葉ページ内のエントリ数が増えて、そのページに収まり切らなくなると、分割が生じる。この時に、その葉ページの親にあたるインデックスページを書き換える必要があり、ディレクトリが更新される。このため、ディレクトリは、葉ページに近い部分の更新が頻繁に起こることになる。しかし、逆に根に近い部分の更新頻度は低いことが分かる。

Fat-Btree では、多くの葉ページのパスに共有される木の根に近い部分のページは、多数のコピーを持つことになり、ルートページは全ての PE にコピーされることになるが、それらは相対的に更新頻度が低いページである。一方、更新頻度の高い下位のインデックスページは殆んどコピーを持たない。つまり、コピーの数が更新頻度に応じて適切に抑えられるようになっている。これによって、頻繁なディレクトリの更新が行なわれる状況においても、PE 間の同期を極力抑えることができ、システムのスループット低下を防ぐことができる。

また B-tree 構造では、各ページの参照される頻度も、そのページのレベルに関係する。この場合は更新頻度とは逆で、上位のレベルほどページの数が少ないので、上位のペー

ジほど参照される頻度が高くなる。特にルートページは、全てのデータ検索において参照されることになる。Fat-Btree の、上位のページほどコピーの数が多く、下位のページは殆んどコピーを持たないという特徴は、このページの参照頻度にもちょうど合っていて、参照頻度が高いページほど並列にアクセスができるようになっている。これによって Fat-Btree では、上位ページが置かれている特定 PE へのアクセス集中が生じないようになっている。

3.2.2 検索時のオーバヘッド

Fat-Btree では、各 PE にはディレクトリ全体は無く、その一部しか配置されていない。データの検索において、もし検索しているデータが問い合わせを実行している PE 上にある場合は、Fat-Btree ではそのデータまでの全てのアクセスパスは同一 PE 上にあるので、検索はその PE 単独で行なえ、問題は生じない。しかし、もし検索しているデータが他の PE 上にある場合は、問い合わせを実行している PE には、そのデータまでのアクセスパスの一部しか存在しない。この場合、その処理はアクセスパスの続きを保有している他の PE に引き継がれることになる。無共有型並列計算機では、PE 間の問い合わせの受渡しはメッセージ通信で行なわれることになるのであるが、このメッセージ通信のコストは無視できない。

さらに、Fat-Btree では各インデックスページがコピーを持っている可能性がある。よって、アクセスパスの続きのインデックスページが、コピーされて複数の PE に存在する場合がある。この場合、どの PE にデータが存在するのか特定することができないので、いずれかの PE に問い合わせを引き継ぐことになる。もし、その引き継ぎ先の PE にデータが存在しなければ、その PE はまた他の PE へ問い合わせを飛ばすことになる。こうして Fat-Btree では、最悪の場合、アクセスパスの 1 ページを参照するたびに、他の PE へその問い合わせを引き継ぐことになる。もっともこれはあくまでも最悪のケースで、下位のインデックスページは殆んどコピーを持たないことを考えると、問い合わせの引き継ぎの発生回数は、平均すればそんなに多くならないものと考えられる。

このような、アクセスパスの分断は、B-tree 全コピー方式やディレクトリ集中配置方式のページ分配型並列 B-tree(節 2.2.2参照) では発生しない。しかし、インデックスページ分配方式 (節 2.2.2参照) では、アクセスパスの 1 ページ毎が殆んど別々の PE に配置されていて、その度に問い合わせの引き継ぎが発生してしまう。それと比べると Fat-Btree で

は、配置されているデータへの全アクセスパスを同一 PE 上に置くことによって、問い合わせの引き継ぎの発生回数が抑えられている。

3.2.3 キャッシュメモリ要求容量

データへのアクセスの高速化のために、ディレクトリの一部はメモリ上にキャッシュしておくことが、一般的である。文献 [4] でも、ディレクトリ自体はメモリ上にあることを前提としている。しかし、データベースのサイズや、他のデータベースの同時利用などの環境を考えると、必ずしも全ディレクトリをメモリ上にキャッシュしておけるとは限らない。

そのような場合、Fat-Btree は、他のページ分配型並列 B-tree に対して更に有利である。つまり、B-tree 全コピー方式やディレクトリ集中配置方式は、ディレクトリ全体の容量に対応したメモリをディレクトリの配置されている PE に要求するのに対し、Fat-Btree は当該 PE が格納している葉ページの上位ノードのみでよいからである。これは、メモリ容量がディレクトリ容量よりも小さい場合には、キャッシュヒット率の向上を意味する。また、B-tree 全コピー方式で、他の PE への検索に対しても、自分のキャッシュを使ってしまうと、アクセスの局所性が落ちることになるが、Fat-Btree ではアクセスの局所性を保つことができる。

3.3 データの偏り制御

Fat-Btree では、構築時にはデータを各 PE に均等に配置することができるが、その後タプルの挿入や削除が繰り返されるうちに、データ配置に偏りが生じる可能性がある。このことは一見、値域分割方式と同様であるように思われるかもしれない。しかし、値域分割方式ではデータ分割の基準値を固定してしまっているので動的に生じたデータ配置の偏りを均等化するには膨大なコストがかかるのに対して、Fat-Btree では葉ページを PE 間で移動させることによって比較的小さなコストで動的にデータ偏り制御を行なうことができる。以下に、Fat-Btree におけるデータ偏り制御の 1 つの方法を述べる。

まず各 PE が持つ葉ページ数の情報を格納するリストを、常に一定間隔で順番に PE 間を巡回させて、各 PE の最新の葉ページ数を持ち回る。これによって、PE 間のデータ配置の偏り具合を、各 PE が知ることができる。葉ページ数が平均値に対して一定の閾値を越

えて多くなったPEは、過剰分の葉ページを両隣のPEに移動させる操作を行なう。このとき、葉ページの移動に伴い、関係する全てのインデックスページも更新する。過剰分の葉ページを受けとった両隣のPEでは、もしそれによって葉ページの数が増加し過ぎた場合は、それ自身が持っている葉ページの一部をさらに反対側の隣接PEに移動させる。これを繰り返し行なうことによって、Fat-Btreeではデータ配置を均等化することができる。

Fat-Btreeのデータ偏り制御における一連の葉ページ移動は、必ずしも全てのPEで協調して行なう必要はない。もちろん、葉ページの移動によってルートページが書き換えられるような場合は、全PEに置かれているルートページのコピーを変更する必要があり、結局は全PEが葉ページの移動に巻き込まれることになるので、最初から全PEで協調して作業を行なう方が効率が良い。しかしながら、葉ページの移動で上位の方のインデックスページが書き換えられない場合は、全体で協調せずに一部のPEのみで葉ページの移動を行なうことも可能で、その方が全体で同期をとる必要がないので効率良くデータ偏り制御を行なえる。

第 4 章

計算による性能解析

本章では、Fat-Btree の有効性を検証するために、確率に基づく性能解析を行ない、Fat-Btree と従来の手法である B-tree 全コピー方式 (節 2.2.2参照) のスループットとレスポンスタイムを比較する。

4.1 解析手法

本解析においては、インデックスページのコピーの存在する確率やページが分割を起こす確率を考慮して、READ や WRITE の 1 回の処理に必要なページアクセス数と通信メッセージセットアップ数を求め、それから 1 回の処理の総処理時間とレスポンスタイムを求め、システム全体のスループットとレスポンスタイムを計算する。なお、本解析においては、簡略化のために以下のような仮定を置いている。

- READ は 1 タプル READ のみを考える。
- WRITE は 1 タプル INSERT のみを考える。
- ロック獲得の待ち時間は考えない。
- PE 間通信にブロードキャストは利用しない。
- インデックスページの一部はメモリ上にキャッシュしておける。
- B-tree 中のデータは完全にバランスしている。
- 各 PE に与えられる負荷は完全に均等である。
- 各データは均等にアクセスされる。
- タプルは複数データページにまたがらない。

ただしFat-Btreeにおいては、WRITEを行なう際に処理に巻き込まれるPEの数とページの数は、更新されるインデックスページの持つコピーの数に依存し、これによって総ページアクセス数やメッセージ数が大きく変化する。したがって、性能解析するにあたって、まず各インデックスページに対するコピーの数の期待値を見積もる必要がある。

またFat-Btreeでは、ディレクトリの部分木しか各PEに置いてないので、処理対象のデータまでのアクセスパスに対応するインデックスページがそのPEに存在する確率を見積もる必要がある。もし存在しない場合には、自分の持つ部分木からアクセスパスが存在する可能性のあるPEを推定し、そのPEに問い合わせを引き継ぐことになる。この際、既にコピーが存在しないほど下まで降りてきている場合には、PEは一意に定まるが、まだ下にコピーが存在する場合には、コピーのいずれか1つに問い合わせを引き継ぐことになる。問い合わせを引き継いだ方では更に処理を続けることになるが、そのPEで最低1つは木を下に辿ることになるため、コピーの数は減っていき、データの格納されているPEを同定することができる。

それから、WRITE操作において、更新する必要のあるインデックスページの数は、ページの分割の発生の仕方に依存するので、ページ分割の発生確率も見積もる必要がある。さらに、ページアクセスに要する時間は、ページがメモリ上にキャッシュされている場合と、キャッシュされておらずディスクにアクセスにいく場合では、大きく異なるので、ページがメモリ上にキャッシングされている確率も見積もる必要がある。

以下では、これらの期待値や確率の導出法について説明する。なお、解析式の詳細に関しては付録Aで述べる。

4.1.1 コピー数の期待値

Fat-Btreeの各段のインデックスページのコピーの総数は、リレーションの分配されるPEの総数をN_{PE}とすると$(N_{PE}-1)$を越えない(ここで、コピーとはオリジナルを含まない)。なぜならFat-Btreeでは、下位のページがPE間に跨っているインデックスページにのみコピーが生じるため、各PEの間では高々1つのコピーしか発生せず、最大限コピーが発生したとしても、植木算と同様に$(N_{PE}-1)$にしかならないからである。

ここでは簡略化のため、全ての段のインデックスページについて、最悪の場合である総数$(N_{PE}-1)$のコピーが生じると仮定する。このとき、元のB-treeの第i段のインデックスページの総数を$IP(i)$とすると、Fat-Btreeの第i段インデックスページ1ページあ

たりに作られるコピーの数の期待値は、

$$\frac{N_{PE}-1}{IP(i)} \tag{4.1}$$

となる。つまり、1 段あたりのページ数に対して PE 数が大きくなるほど、1 ページあたりのコピー数の期待値が増える、つまりコピーされる可能性が高くなるということを表している。ルートページの場合には、$IP(i)=1$ であるため、$(N_{PE}-1)$ のコピーが作られる。

4.1.2 当該ページがその PE に存在する確率

前節の前提から、Fat-Btree の第 i 段目のインデックスページの総数は、コピーの数を含めて$IP(i)+(N_{PE}-1)$で、各 PE がそれらを均等に持つことになるので、各 PE にある第 i 段インデックスページの数は、

$$\frac{IP(i)+(N_{PE}-1)}{N_{PE}} \tag{4.2}$$

となる。

次に、ある任意の第 i 段インデックスページが、検索しているものである確率は $1/IP(i)$ であるから、ある PE が検索している第 i 段インデックスページを持っている確率は、

$$\frac{IP(i)+(N_{PE}-1)}{N_{PE}\times IP(i)} \tag{4.3}$$

となる。つまり、1 段あたりのページ数と PE 数が大きいほど確率は低くなる。なお、ルートページの $IP(i)=1$ を当てはめれば、この確率は 100%となる。

4.1.3 ページ分割の発生確率

インデックスページの最大エントリ数を Eとすると、B-tree の性質から、ページ分割を起こした直後の葉ページのエントリ数は $E/2$ で、それからタプルが $E/2$ だけ挿入されると、その葉ページは一杯になり再びページ分割を起こす。つまり、タプルの挿入だけを前提とすると、葉ページはタプルの挿入 $(E/2)$ 回に 1 回の割合でページ分割を生じることになる。これより、新しいタプルが挿入された時に葉ページがページ分割を起こす確率は、$\frac{1}{E/2}$ であると考えることが出来る。

葉ページより上位のインデックスページに関しても同様に考えると、新しいタプルが挿入された時に、第 i 段インデックスページがページ分割を起こす確率は、次の式で表すことができる。ここで H は木の高さである。

$$\left(\frac{2}{E}\right)^{H-i+1} \tag{4.4}$$

つまり、木の高さが増え、上位のインデックスページになるほど、ページ分割の確率は減る。

4.1.4 ページのキャッシュヒット率

B-tree では、ページのアクセス頻度は、そのページのレベル (ルートページからの段数) に大きく関係する。データへのアクセスが均等に行なわれると仮定すると、多くのアクセスパスに共通する上位のページほど、アクセス回数が多くなる。よって、B-tree においてページをメモリにキャッシュする場合は、ページのレベルによって優先度を変え、上位のページを優先してメモリにキャッシュすることが望ましい。また、同じレベルのページ同士に関しては、LRU 法などによって、そのレベル内でアクセス頻度の高いページを優先することが望ましい。

本解析では、データへのアクセスは均等に行なわれると仮定しているので、ページのキャッシングの優先度はレベルのみで決められて、同じレベルのページ同士は同優先度とした。つまり同じレベルのページに関しては、メモリにキャッシュするページはランダムに決めるとしている[1]。この場合、Fat-Btree の i 段目の任意の 1 ページが、メモリ上にキャッシュされている確率は次のように表される。

$$\begin{array}{cc} 0 & \left(\sum_{k=1}^{i-1} IP_{\mathrm{1PE}}(k) \geq C\right) \\ \frac{C-\sum_{k=1}^{i-1} IP_{\mathrm{1PE}}(k)}{IP_{\mathrm{1PE}}(i)} & \left(\sum_{k=1}^{i-1} IP_{\mathrm{1PE}}(k) < C < \sum_{k=1}^{i} IP_{\mathrm{1PE}}(k)\right) \\ 1 & \left(\sum_{k=1}^{i} IP_{\mathrm{1PE}}(k) \leq C\right) \end{array} \tag{4.5}$$

ここで $IP_{\mathrm{1PE}}(i)$ は、Fat-Btree において、各 PE にある第 i 段インデックスページの数 (コピーを含む) で、これは式 4.2で求められる。また Cは、Fat-Btree の 1PE あたりのキャッシュメモリのページ数である。

[1]データへのアクセスが完全に均等に行なわれると仮定した場合、B-tree の同レベルのページ間での参照の局所性はないので、いかなるページ置換アルゴリズムを用いても結果は変わらない。

4.2 解析結果

本節では、節 4.1の確率や期待値から求められた解析式を用いて、Fat-Btree 方式と B-tree 全コピー方式 (節 2.2.2参照) のスループットの比較と、Fat-Btree におけるメモリによるページのキャッシュ効果、処理に占める WRITE の割合の変化によるスループットの推移、およびレスポンスタイムの比較を行なう。

表 4.1に、解析結果を求める際に用いた設定値を示す。この中でハードウエアに依存する値は、並列計算機 nCUBE3 での値を基にして決めている。また、インデックスページの利用率の ln2 は、一般的な B-tree におけるページ利用率の期待値である。それから PE 数と Relation のタプル数は、スケーラビリティを調べるために、上限を大きくとっている。

また、表 4.1の各設定値を用いた際の B-tree の構成を表 4.2に示す。表の第 1 列目は Relation のタプル数を表していて、2 列目から 6 列目が B-tree の各段のインデックスページの数を、最後の列が B-tree の木の高さを表している。2 列目から 6 列目の各段のインデックスページ数は、左にいくほど木の上位の段を表していて、記入されている内で一番左側がルートページで、一番右側が葉ページのページ数を表している。

表 4.1: 解析に用いた設定値

ページサイズ	4KB
インデックスページの最大エントリ数	200
インデックスページの利用率 (ルートページ除く)	ln2
メッセージセットアップ時間	200μs/message
ページアクセス時間 (ディスク)	10ms/page
ページアクセス時間 (メモリ)	1 μs/page
PE 数 (ディスク数)	32 $\sim$ 512
Relation のタプル数	100K $\sim$ 1G
メモリにキャッシュできるページの数	0 $\sim$ 1000

表 4.2: 本解析における B-tree の構成

タプル数	下から 5 段目	同 4 段目	同 3 段目	同 2 段目	葉ページ	木の高さ
100K			1	6	722	3
1M			1	53	7214	3
10M		1	4	521	72135	4
100M		1	38	5204	721348	4
1G	1	3	376	52035	7213476	5

4.2.1 スループットの比較

最初に、メモリにキャッシュできるページ数を 100、全オペレーションに占める WRITE の割合を 0.1 として、PE の数が 32 から 512 の 5 種類の場合について、タプル数を変化させながら本方式と B-tree 全コピー方式のスループットの推移を求めた (図 4.1, 図 4.2)。

それぞれの方式のスループットを比べると、特に PE の数が多い時には Fat-Btree の方が大幅にスループットが高くなっているのが分かる。これは、B-tree 全コピー方式では、PE の数が増えるとディレクトリ更新のコストが非常に大きくなって、それがスループットを低下させるからである。それゆえに、Fat-Btree では PE 数の増加にともなってスループットが向上するのに対して、B-tree 全コピー方式では PE の数が 256 の場合を境にして PE 数が増加するとスループットが低下している。

タプル数に関しては、図 4.1の Fat-Btree ではタプル数 1M から 10M のあたりで、図 4.2 の B-tree 全コピー方式ではタプル数が 100K から 1M あたりでスループットが低下し始めている。これは、タプル数が少ない時には B-tree の全てのページがメモリに収まっていたのが、タプル数が増えるにつれてメモリに入りきらないページが出てきて、ディスクアクセスの回数が増えるからである。両方式でスループットが低下し始めるタプル数が異なるのは、Fat-Btree では各 PE にディレクトリの一部しか置かないので、各 PE に置くページ数が少なくて済み、1PE 当たりのメモリ量が同じ場合には全てのページがメモリに収まり易くなるためであると考えられる (節 3.2.3参照)。

また、図 4.1では、Fat-Btree のスループットが PE 数が 512 のときだけ、タプル数が 100K から 5M の間で大きく増加しているのも特徴的である。Fat-Btree では、PE 数が多いときにタプル数が少なくてインデックスページが少ないと、インデックスページ 1 ペー

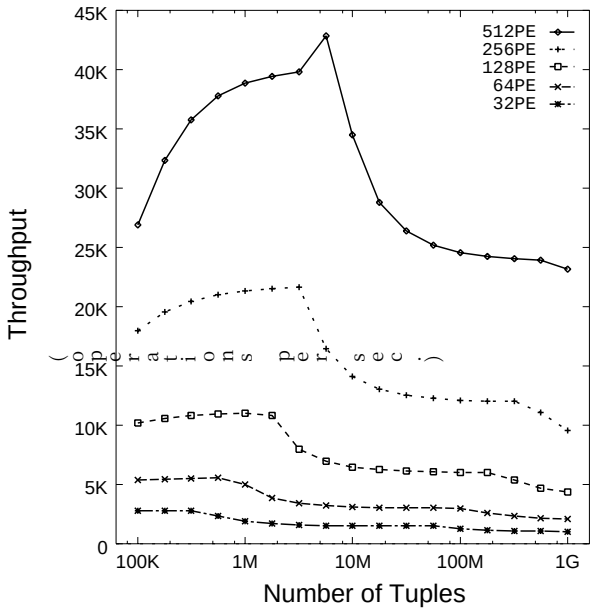


図 4.1: Fat-Btree のスループット
[キャッシュメモリページ数 : 100, WRITE 割合 : 0.10]

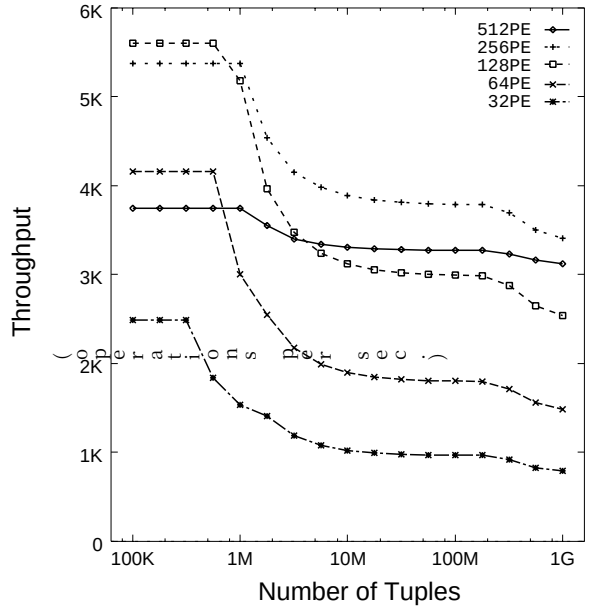


図 4.2: B-tree 全コピー方式のスループット
[キャッシュメモリページ数 : 100, WRITE 割合 : 0.10]

ジあたりのコピーの数が増加する (式 4.1参照)。その結果、WRITE の処理に巻き込まれる PE 数が増加し、システム全体のスループットが低下する。ゆえに、PE 数が 512 のときはタプル数が少ないとスループットが低いのである。

なお、PE 数が 512 のときのスループットがタプル数が 3M から 5M の間で特に増加しているのは、タプル数の増加によってここでルートページがページ分割を起こしているからである。ルートページがページ分割を起こすと木の高さが 1 段高くなり、ルートページが WRITE の処理に巻き込まれる可能性が低くなる (式 4.4参照)。Fat-Btree では、ルートページは全ての PE にコピーされているので、WRITE の処理でルートページが更新される場合は全ての PE がその処理に巻き込まれることになり、その処理は非常に重くなる。ゆえに、ルートページが WRITE の処理に巻き込まれる可能性が低くなると、システム全体のスループットが向上すると考えられる。

4.2.2 キャッシュメモリ量の影響

次に Fat-Btree におけるメモリキャッシュの影響を調べるために、キャッシュに利用できるメモリのページ数を変えてみた。その結果が図 4.3、図 4.4、図 4.5である。

図 4.3は、キャッシュに利用できるメモリのページ数を 1000 に増やした場合の Fat-Btree のスループットである。図 4.1と比べると、キャッシュメモリのページ数が増えたことによって、スループットが低下し始めるときのタプル数の値が大きくなっているのが分かる。

図 4.4、図 4.5は、キャッシュメモリのページ数を 0、つまりメモリによるページのキャッシングの効果がなく、ページアクセスは毎回ディスクに対して行なうとした場合の Fat-Btree と B-tree 全コピー方式のスループットである。図 4.1、図 4.2と比べると、ディスクアクセスのコストの分だけ両方式とも全体的にスループットが低下しているのが分かる。この場合でもなお、PE の数が多くなるほど Fat-Btree の方が B-tree 全コピー方式よりもスループットが高くなっている。

なお、図 4.4、図 4.5で、タプル数が 5M のあたりと 500M のあたりの 2 箇所で PE 数に関わらずスループットが低下しているのは、そこで B-tree のルートページがページ分割を起こし、木の高さが 1 段高くなったために、データにアクセスするのに必要なディスクアクセスの回数が 1 回増えたためである。

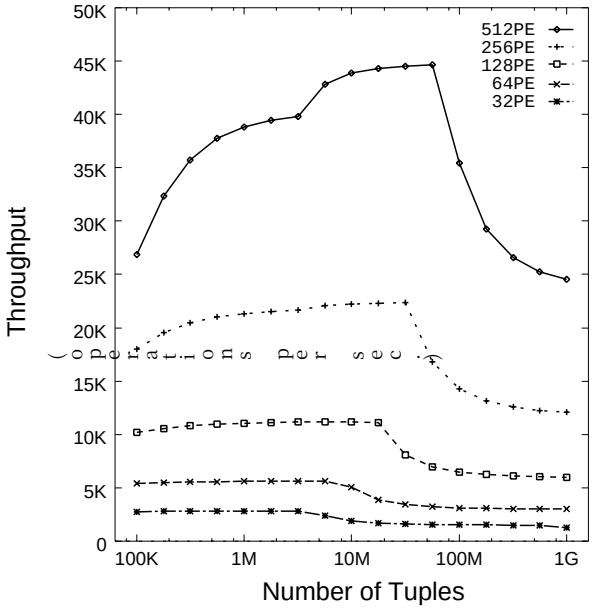


図 4.3: キャッシュ効果大の Fat-Btree のスループット
[キャッシュメモリページ数 : 1000, WRITE 割合 : 0.10]

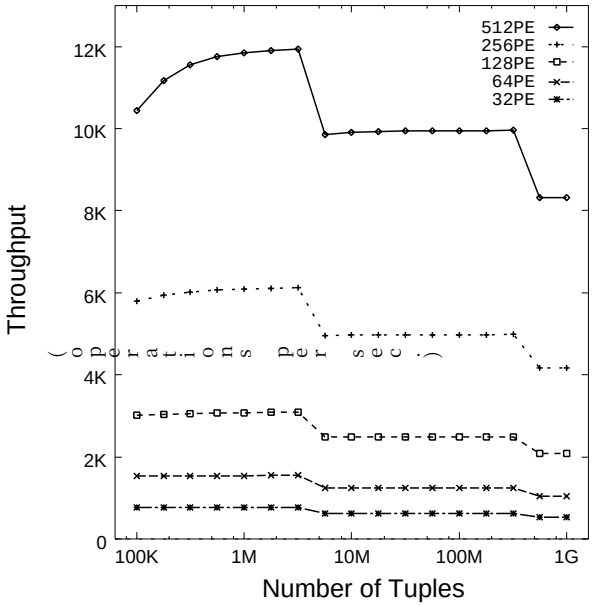


図 4.4: キャッシュ効果 0 の Fat-Btree のスループット
[キャッシュメモリページ数 : 0, WRITE 割合 : 0.10]

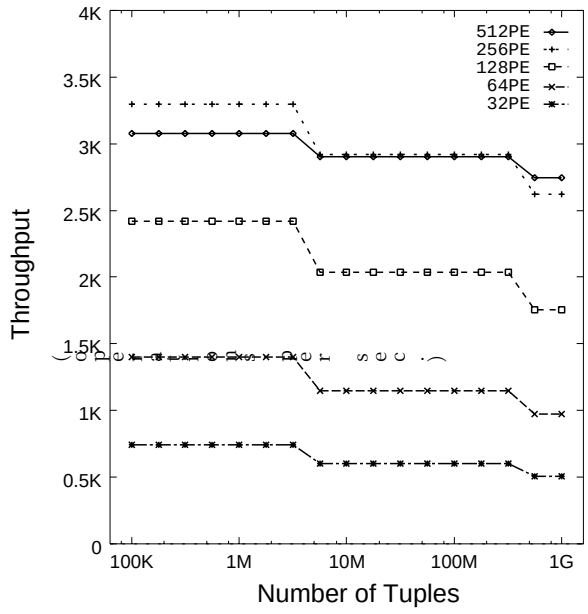


図 4.5: キャッシュ効果 0 の B-tree 全コピー方式のスループット
[キャッシュメモリページ数 : 0, WRITE 割合 : 0.10]

4.2.3 WRITE の割合の変化の影響

メモリにキャッシュできるページ数を 100 とし、タプル数を 1M に固定して、PE 数が 32、128、512 の 3 通りの場合について、処理に占める WRITE の割合を変化させた時の Fat-Btree と B-tree 全コピー方式のスループットを比較した (図 4.6, 図 4.7, 図 4.8)。

B-tree 全コピー方式では、PE 数が多くなるほど、WRITE の割合の増加に伴ってスループットが大きく低下している。これは、全 PE にディレクトリ全体が置かれているので、WRITE によるディレクトリ更新時に全 PE をその処理に巻き込むからである。一方、Fat-Btree の方は WRITE の割合が増加してもスループットは緩やかにしか低下しない。これは、Fat-Btree では WRITE によってディレクトリの更新が行なわれても、書き換えられるインデックスページのコピーが少数の PE だけにのみに置かれている場合が多くて、少ない PE だけで WRITE の処理を行なえるので、B-tree 全コピー方式に比べて WRITE の処理が軽いからである。

なお、WRITE の割合が 0 のときも Fat-Btree の方が B-tree 全コピー方式よりもスループットが高いのは、キャッシュメモリのヒット率の差である (節 3.2.3参照)。メモリにキャッ

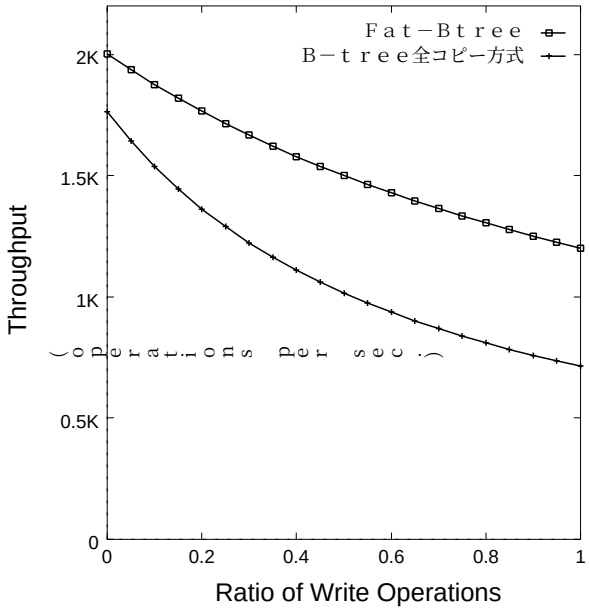


図 4.6: WRITE の割合の変化によるスループットの推移 1
[キャッシュメモリページ数 : 100, タプル数 : 1M, PE 数 : 32]

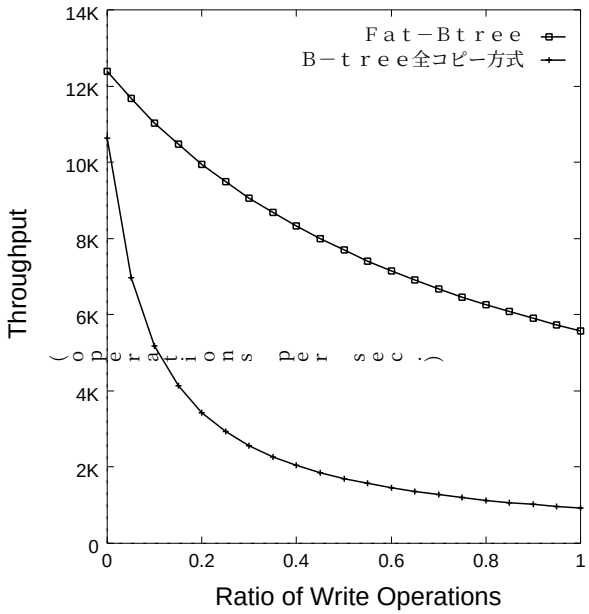


図 4.7: WRITE の割合の変化によるスループットの推移 2
[キャッシュメモリページ数 : 100, タプル数 : 1M, PE 数 : 128]

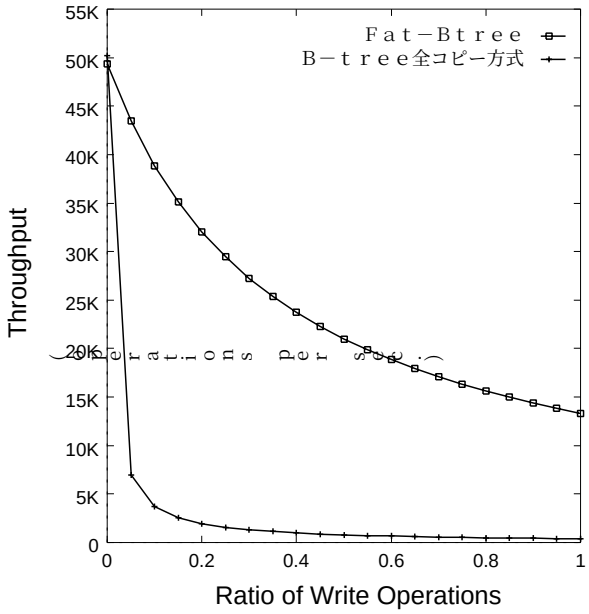


図 4.8: WRITE の割合の変化によるスループットの推移 3
[キャッシュメモリページ数 : 100, タプル数 : 1M, PE 数 : 512]

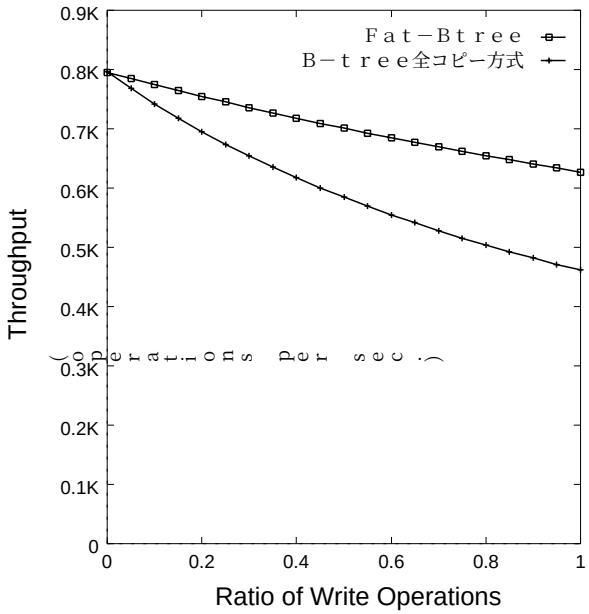


図 4.9: キャッシュ効果 0 の場合の WRITE の割合の変化によるスループットの推移
[キャッシュメモリページ数 : 0, タプル数 : 1M, PE 数 : 32]

シュできるページ数が 0 のときは、WRITE の割合が 0 であると、Fat-Btree よりも B-tree 全コピー方式の方がスループットが少し高くなる (図 4.9)。しかし、その差は僅かであり、WRITE の割合が増えるとすぐに逆転して、その後 Fat-Btree 方式のスループットが大きく上回ることになる。

4.2.4 レスポンスタイムの比較

次に、Fat-Btree 方式と B-tree 全コピー方式のレスポンスタイムの解析結果を示す (図 4.10, 図 4.11)。

当初、対象となるインデックスページが問い合わせを受け付けた PE にない場合に、別の PE と通信して処理を引き継ぐ必要のある Fat-Btree の方が、要求を受け付けた 1 つの PE 内で直接葉ページの格納場所を特定できる B-tree 全コピー方式に比べて、レスポンスタイムが悪くなると予想していた。しかし、図 4.10と図 4.11を比べると、Fat-Btree の方がレスポンスタイムにおいても優れているということが分かる。 これは、Fat-Btree では

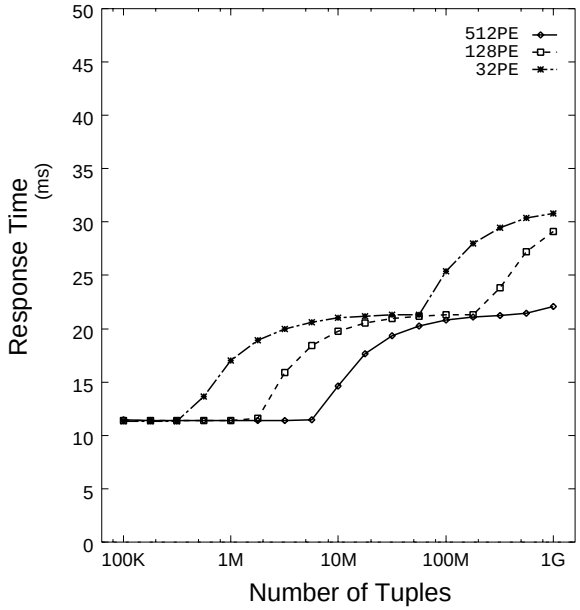


図 4.10: Fat-Btree のレスポンスタイム
[キャッシュメモリページ数 : 100, WRITE 割合 : 0.10]

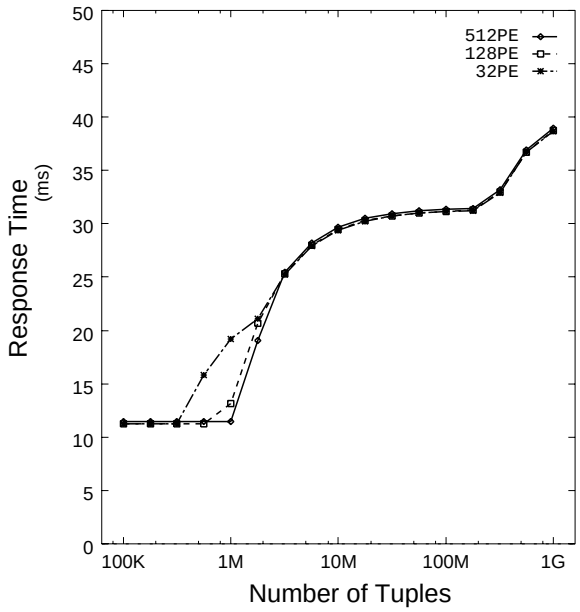


図 4.11: B-tree 全コピー方式のレスポンスタイム
[キャッシュメモリページ数 : 100, WRITE 割合 : 0.10]

各 PE にディレクトリの一部しか置かないので、各 PE に置くページ数が少なくて済み、キャッシュメモリの容量を一定だとするとページのメモリに載る割合が大きくなり、ディスクをアクセスする回数が減ることから通信コストを相殺できるためと考えられる。つまり、Fat-Btree 方式のキャッシュメモリの相対的なヒット率の高さの効果であると言える。

4.3 解析のまとめ

本章では、新しい並列ディレクトリ構成法である Fat-Btree の有効性を検証するために、計算による性能解析を行ない、従来の手法である B-tree 全コピー方式との比較を行なった。

その結果、メモリをキャッシュとして用いず、PE 数が少なく、ディレクトリの更新が殆んど無い場合を除いて、大抵の場合、Fat-Btree 方式の方がスループットが優れていることが確かめられた。特に、PE の数、タプル数が多くなり、WRITE の割合が増えるにつれて、Fat-Btree 方式は B-tree 全コピー方式よりもスループットが大幅に向上する。これは、PE の数が増加すると、B-tree 全コピー方式ではディレクトリ更新時の PE 間の同

期処理に時間が非常にかかり、そのためにシステム全体のスループットが低下してしまうのだが、Fat-Btree は一部の限られた PE のみで更新処理を行なえるので、多数の PE による同期処理が少なくて済み、PE の数の増加にしたがってスループットがほぼ線形に向上するからである。

また Fat-Btree では、1 つの PE に置くディレクトリのページ数が少ないので、キャッシュメモリのヒット率が高くなり、それによって B-tree 全コピー方式よりもレスポンスタイムも向上することが分かった。

第 5 章

Fat-Btree の実現

本章では、Fat-Btree の通常の B-tree との違いに起因して、Fat-Btree を実際に実装する際に生じる選択肢について考察する。

5.1 インデックスページの構造

通常の B-tree と異なり、Fat-Btree では問合せ処理を実行する PE 上にディレクトリが一部しか存在しない。それゆえに、もし、ある PE 上で問合せ処理におけるデータ検索の為にアクセスパスを辿っていて、必要なインデックスページがその PE 上に存在しなかった場合は、その PE は問合せ処理を必要なインデックスページを格納している PE に任せることになる。このときに、最初に問合せを処理していた PE では、それまでにアクセスパス上のいくつかのインデックスページを辿ってきているはずである。できれば次にその問合せを任された PE では、アクセスパス上で続きとなるインデックスページから探索を再開したい。そのためには、インデックスページ内の子ページへのポインタ部の構造が非常に重要となる。

通常の B-tree では、インデックスページ内には子ページへのポインタとして子ページの物理ページ番号が記載される。一方 Fat-Btree の場合には、子ページへのポインタ部の構造として、次のようなものが考えられる。

1. **全コピーページの物理ページ番号** これは、子ページの全てのコピーの物理ページ番号 (PE 番号 + ディスク番号 + セクタ番号) をポインタ部に記載するという方法である (図 5.1(1))。しかし、Fat-Btree ではインデックスページに多数のコピーが存在す

る場合がある。それゆえに、もし子ページの全てのコピーの物理ページ番号を記載できるようにしようとすると、そのポインタの格納の為に必要となるスペースは非常に大きなものとなり[1]、インデックスページのスペース利用効率が非常に低下してしまう。また、PE 数によりインデックスページの形態が変化してしまう。

2. **他 PE のコピーの物理ページ番号を別ページに保管** これは上の方法の改良で、同一 PE 内にある子ページの物理ページ番号はインデックスページ内に記載するが、他の PE に置かれているコピーの物理ページ番号は別ページに保管して、インデックスページからはその物理ページ番号保管場所へのポインタだけを張っておく (図 5.1(2))。こうすることによって、インデックスページの形態を PE 数に依存させることなく、インデックスページ自体のスペース利用効率の問題も解決できる。

3. **物理ページ番号 + コピーのある最初と最後の PE の PE 番号** これは、子ページが同一 PE 内にある場合にはその物理ページ番号をポインタとして記載し、それとは別に子ページのコピーが置いてある PE の PE 番号も記載するという方法である (図 5.1(3))。この方法では、子ページが同一 PE 内に無い場合には子ページの格納されている PE が分かるだけで、データの検索はその子ページの格納されている PE においてルートページからやり直さなければならない。これは、アクセスパスが長くなるのと同じことで、レスポンスタイムやスループットの低下を招く危険性がある。また、異なる PE に存在するコピーページの為の識別子[2] が無いので、他の PE に存在するコピーページのロックを獲得する際などに、求めるページを識別するのに大きなコストがかかる。

4. **論理ページ ID + コピーのある最初と最後の PE の PE 番号** この方式では、まず各インデックスページに PE 間で共通して使用できる論理ページ ID を付ける。このとき、異なる PE に置かれている同一ページのコピー同士は、同一の論理ページ ID になるようにする。そして、インデックスページ内のポインタ部にはこの論理ページ ID のみを記載し、物理ページ番号を得るには別に用意されている論理ページ ID と物理ページ番号の対応表を使用するようにする (図 5.1(4))。この方式の場合には、PE 間で共通して使用できる論理ページ ID を設けるので、異なる PE 間でページを

[1] Fat-Btree では 1 つのページに数十、数百のコピーが存在することがある。

[2] 物理的な識別子 (物理ページ番号) 又は論理的な識別子 (論理ページ ID)

認識し合うことが極めて容易となり、複数 PE 間に渡る問合せ処理においては非常に好都合である。しかしこの方式では、対象ディスク領域が大きい場合、論理ページ ID と物理ページ番号の対応表の為に非常に大きなメモリスペースが必要となり、また対応表を多層化した場合には、表検索にかかる時間も問題となる。

これらの候補のうち、現在のところ有力と考えられているのは 2. の方法である。3. と 4. の併用、つまり PE 内のページへのアクセスには物理ページ番号を使用し、PE 外のコピーページへのアクセスには論理ページ ID を使うという方法は、かなり良い性能をもたらすものと思われるが、その実現にはメモリスペースの制約が伴う。

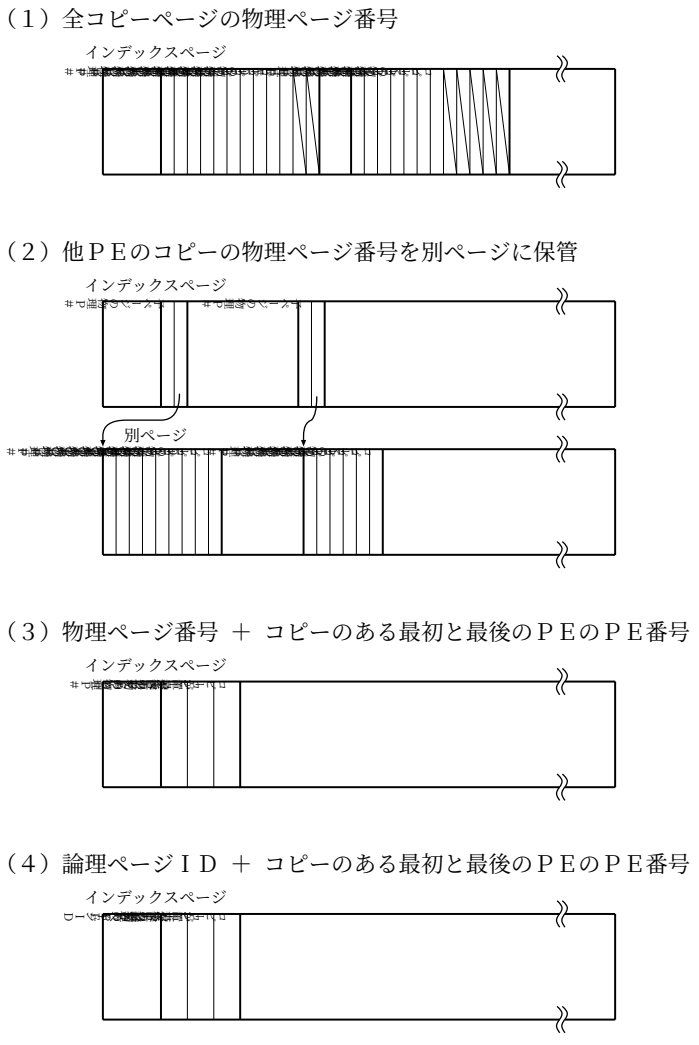


図 5.1: インデックスページのポインタ部の構造

5.2 初期状態における Fat-Btree の構築法

初期状態において、ある程度のタプル数を持っているリレーションから Fat-Btree を構築する際には、どのようにして Fat-Btree の各 PE で異なるディレクトリ構造を構築するかという点が問題となる。特に、インデックスページ内の子ページへのポインタ部が物理ページ番号で示されている場合には、子ページの格納されている物理ページ番号が異なるとコピーページ同士でページの内容が異なることになり、Fat-Btree の構築に工夫が必要となる。以下では、考えられる Fat-Btree の構築法の選択肢を挙げる。

初期状態における Fat-Btree の構築法は、大きく分けて 3 つある。

1. データ偏り制御を行ないながら、通常のタプルの挿入操作を繰り返す
2. 1 つの PE で B-tree を構築しておき、それを他の PE へコピーする
3. 各 PE でそれぞれ同時に同じ Fat-Btree を作る

1. の方法では、初期構築の為の特別の手続きは何も必要ない。タプル 0 個の Fat-Btree から始めて、通常のタプル挿入操作を繰り返すのみである。もちろん、小さな Fat-Btree にタプルを多量に挿入すればデータ配置が偏るので、頻繁にデータ偏り制御を行なうことは必要である。タプルの挿入操作は、木が小さい時にはロックの競合を避ける為に 1 つの PE のみで行なう方が効率が良いが、木の大きさがある程度大きくなれば複数の PE で同時に行なう方が効率が良くなる。この方法の欠点は、頻繁に行なうデータ偏り制御で生じる多量のページ移動である。特に木が小さい時には、上位インデックスページを巻き込む大規模なページ移動が頻繁に生じ、その処理には非常に大きなコストがかかる。

2. は、1 つの PE のみで逐次的に B-tree を構築して、その完成後に必要なページだけを他の PE にコピーするという方法である。こうすることによって、ページを書き換える際に一切ロックを獲得する必要がなくなり、特に木が小さい間は効率的に構築が行なえる。しかしこの方法では、PE 間でページがコピーされる時にページが格納される物理ページ番号が変わるので、インデックスページ内の物理ポインタを書き換える必要が生じることが問題である。特に、親ページが子ページを物理ページ番号だけで識別しているような場合、子ページの物理ページ番号が変わってしまうと子ページの識別が難しくなり、親ページの物理ポインタの書き換えに工夫が必要となる。この問題は、インデックスページ内の子ページへのポインタを、論理的なページ ID にできれば解決する。

3. は、各 PE でそれぞれ同時に同じリレーションから同じ Fat-Btree を作り、その後に各 PE で不要なページを削除するという方法である。この方法は、2. の方法と比べてページを PE 間で転送する必要が無くなることと、インデックスページ内の物理ページ番号の書き換えが同一 PE 内の子ページの分は必要なくなるという長所がある。しかし、この方法でもやはり、インデックスページ内の子ページへのポインタとして、他の PE にあるページの物理ページ番号を記載する場合は、他 PE の物理ページ番号情報の獲得とインデックスページへのその情報の書き込みが必要となる。

これら 3 つの方法の良否は、5.1節のインデックスページのポインタ部の構造にも依存するので一概には言えないが、最初は 3. の方法で Fat-Btree を構築し、木がある程度の大きさになったら 1. の方法に切替えるのが、現在のところの最善の選択であると考えられる。

5.3 ページロックプロトコル

通常の B-tree における READ の際のページロックプロトコルは、次のようなものである。まずルートページを S ロック[3]し、それから子ページの S ロックを獲得しては親ページの S ロックを解放するということを繰り返し、葉ページまでアクセスパスをたどる。このとき、親ページのロックの解放を子ページのロックが獲得できた後で行なうのは、子ページ以下のアクセスパスが変更されていないことを保証するためである。

同じ方法を Fat-Btree で行なうことを考えると、アクセスパスが PE 間に跨っていることが問題となる。ある PE で親ページのロックを獲得していて、そのロックを確保したまま他の PE に置いてある子ページのロックを要求して、その子ページのロックが獲得できたら親ページのロックを解放する、ということを行なうためには、子ページのロックが獲得できたことを親ページのある PE へ知らせるための余分なメッセージ通信が必要になるし、また、その通信を受けとるまで親ページのロックを確保し続けなければならない。メッセージ通信型並列計算機ではメッセージ通信のコストは小さくはないので、可能な限りメッセージ通信は削減したいし、また、ページをロックしておく時間は、他の問い合わせとのロック競合を起こさないためにも可能な限り短くしたい。ゆえに、子ページのロックが獲得できるまで親ページのロックを保持しておくより、できることなら子ページのあ

[3]本節で議論されている各ロックモード間の適合性を表 5.1に示す。

表 5.1: 各ロックモード間の適合性

mode	S	IX	SIX	X
S	○	○	○	
IX	○	○		
SIX	○			
X				

る PE に問い合わせを引き継ぐ前に親ページのロックを解放するのが望ましい。

PE 間での問い合わせの引き継ぎの際に、子ページのロックを獲得する前に親ページのロックの解放を行なう場合、問い合わせを引き継いだ先の PE で子ページのロックを獲得した時点で既にその子ページが更新されていて、そのページの下にアクセスパスが無くなっていることが考えられ、その場合への対応策が必要となる。対応策としては、次の 2 通りの方法が考えられる。

1. ルートページからの検索のやり直し

2. B-link の使用

1. の方法では、検索を続けた結果たどり着いた葉ページに、検索しているデータがなかった場合には、ルートページからもう一度検索をやり直す。2 回目の検索は 1 つの PE の中で閉じて行なえるので、通常の B-tree における検索と同様で、アクセスパスが保証され、確実にデータにたどり着くことができる。B-tree では、ディレクトリの更新はそう頻繁に発生する訳ではないので、この楽観的な方法は好結果をおさめるかもしれない。

一方 2. の方法では、B-tree を B-link tree 化する [6]。B-link tree は、B^+-tree の葉ページのように、B-tree の同じレベルのページをリンクで結んだものである。B-link tree の各ページには、通常の B-tree のページに記載されている内容の他に、そのページの下のサブツリー内で最も大きなキーの値 (最大キー値) の情報と、右隣のページへのリンクが追加されている。この B-link tree におけるページ検索では、親ページのロックは子ページのロックを獲得する前に解放することができる。各ページにおけるパスの検索では、検索しているキー値をそのページの最大キー値と比べて、検索しているキー値の方が大きい場合は、リンクをたどって右ページへ検索を進める。こうして、アクセスパスのページ

がすでに更新されていた場合においても、適切な次ページ (子ページ又は右ページ) へと進むことができる。この方法の長所は、言うまでもなく検索のやり直しが必要なくなることである[4]。ただ、この 2. の方法で問題なのは、ページ分割によって新しいページを作る時に、最大キー値の情報を新たに得る必要があり、そのために子ページへの 1 回の余分なページアクセスが必要となることである。

以上の 2 通りの方法のどちらが良い結果をもたらすかは、ディレクトリの更新頻度に依存するので一概には言えない。ディレクトリの更新頻度が高い場合には、1. の楽観的な方法では検索のやり直しが多くなり、レスポンスタイムもシステムのスループットも低下する。逆に、ディレクトリの更新頻度が低い場合には、1. の方法でも検索のやり直しは発生せず、2. の方法ではページ分割時に入る余分なページアクセスの分だけ性能が低下する。もっとも、ページ分割の頻度が低い場合には、増加する余分なページアクセスの回数も多くはなく、この性能低下は問題とならないかもしれない。

次に、WRITE 時のページロックプロトコルについて述べる。B-tree の基本的なページロックプロトコル (B-X 法) における WRITE は、次のように行なわれる [6]。まず、ルートページの X ロックを取る。それから、アクセスパスをたどって、次々と子ページの X ロックを取っていく。このとき、親ページのロックは、もしそのページが更新処理に巻き込まれる可能性がないならば[5]、子ページのロック獲得後に解放される。こうしていって、葉ページの X ロックを獲得できたら、それから WRITE の処理を行なう。この時点で、その WRITE 処理によって更新される全てのページには、X ロックが掛けられていることが保証される。

しかし上の方法では、多くのページの X ロックが要求されるので、ロックの衝突が生じやすくなる。そこで、それを解消するために、X ロックの代わりに SIX ロックを用いる方法 (B-SIX 法) がある [6]。この方法では、SIX ロックを用いてルートページからアクセスパスをたどり、葉ページまでたどり着いた時点で、更新範囲のページの SIX ロックを上から順に X ロックに変更する。他に、ロックの衝突をより避けるために、次のような楽観的な方法 (B-OPT 法) もある [6]。この方法では、まずルートページから IX ロックを用

[4] 本来 B-link tree は、ページ分割時に分割する子ページと親ページの非同期な更新処理を可能にし、更新時の同時ロックページ数を少数化して、ロックの衝突を減少させるために考案された手法であるが、Fat-Btree ではコピーページのある他 PE との同期した更新処理を行なう必要があり、ページ間での非同期な更新処理は行なうことができない。

[5] WRITE がタプルの挿入の場合は子ページのエントリが一杯でないときで、WRITE がタプル削除の場合は子ページのエントリが最少数でないとき。

いてアクセスパスをたどっていく。このとき、子ページのロックが確保でき次第、親ページのロックは解放する。そうして、葉ページまでたどりついたら、葉ページでは X ロックを獲得する。もし、葉ページが WRITE 処理によってディレクトリの更新を起こさないならば、WRITE 処理を行なって処理は終了である。一方、WRITE 処理によってディレクトリの更新が生じる場合は、葉ページの X ロックを一度解放して、ルートページから B-SIX 法を用いて WRITE をやり直す。

これらの B-tree の WRITE 時のページロックプロトコルのうち、Fat-Btree に一番適しているのは B-OPT 法である。なぜなら Fat-Btree では、データまでのアクセスパスが PE 間に跨っている場合があり、そのときはデータの置いてある PE ではアクセスパスの一部のページしか参照しないかもしれないからである。B-X 法や B-SIX 法では、葉ページのロックを獲得した時点で更新範囲の全てのページのロックが確保されている必要があるのだが、Fat-Btree においてはこれが保証できないのである。一方 B-OPT 法は、アクセスパスをたどる際に次々とロックを解放していくので、そのまま Fat-Btree に用いることが可能である。B-OPT 法でも、親ページのロックは子ページのロックが獲得できるまでは保持しておくのであるが、Fat-Btree においてアクセスパスが PE 間に跨っている場合には、そこで親ページのロックを解放した後に、子ページの置いてある PE で子ページのロックを獲得することになる。これを行なう際に生じる問題点は前述の READ の場合と同じであり、その解決策もまた READ の場合と同じである。

第 6 章

結論

6.1 まとめ

本研究では、無共有並列計算機上のデータベースシステムにおいて、処理負荷をできるだけ均等に分散させながら、絞り込みによるデータへの高速アクセスを実現する、新しい並列ディレクトリ構成法である Fat-Btree に関して、その特性に関する検討を行なうとともに、確率的な解析による B-tree 全コピー方式との性能比較を行なった。

解析の結果、メモリをキャッシュとして用いず、PE 数が少なく、ディレクトリの更新が殆んど無い場合を除いて、大抵の場合 Fat-Btree 方式の方がスループットが優れていることが確かめられた。特に、PE 数、タプル数が多くなり、WRITE の割合が増えるにつれて、Fat-Btree 方式は B-tree 全コピー方式よりもスループットが大幅に向上する。これは、ページのコピーをアクセス頻度は高いが更新頻度は低い B-tree の上位部分に絞って、処理負荷の集中を避けながらも、ディレクトリの更新時のコピーの同期更新をできるだけ抑えたことによる効果である。また、レスポンスタイムに関しても、Fat-Btree 方式はその効率的なメモリ利用の効果によって、B-tree 全コピー方式よりも優れた結果を示した。

以上の Fat-Btree の性能解析については、電子情報通信学会データ工学研究会にて発表している [9]。

本研究では更に、Fat-Btree 実装法に関する考察も行なった。

Fat-Btree のインデックスページの構造に関しては、同一 PE 内にある子ページの物理ページ番号はインデックスページ内に記載し、他の PE に置かれているコピーの物理ページ番号は別ページに保管して、インデックスページからはその物理ページ番号保管場所

へのポインタだけを張っておくという方法が適していることを示した。また、初期状態における Fat-Btree 構築法については、各 PE でそれぞれ同時に同じリレーションから同じ Fat-Btree を作り、その後に各 PE で不要なページを削除するという方法が適していることを示した。さらに、ページロックプロトコルに関しては、READ については Fat-Btree を B-link tree 化 [6] することによって、PE 間での問い合わせの引き継ぎの際に、子ページのロック獲得前の親ページのロック解放を行なうという方法が適し、WRITE については IX ロックを用いてアクセスパスをたどり、もし葉ページでの WRITE 処理に伴ってディレクトリが更新される場合には、ルートページから SIX ロックを用いてもう一度 WRITE をやり直すという楽観的な手法が適していることを示した。

この Fat-Btree 実装に関する考察は、データ工学ワークショップ (DEWS'98) で発表する予定である。

6.2 今後の研究課題

本研究における計算による性能解析では、簡略化のためにいくつかの仮定を用いた。ロックの待ち時間を考慮しないというのもその一つであった。並列 B-tree において、複数 PE にコピーを持つページを同時更新する際のロック獲得では、待ち時間が生じることが考えられる。このため、ロック獲得の待ち時間を考慮すると、B-tree 全コピー方式が全 PE でインデックスページのコピーのためにロックを獲得しなければならないのに対して、Fat-Btree はインデックスページがコピーされている限られた PE でのみロックを獲得すればよいので、より有利であると考えられる。

また、節 4.1.1において Fat-Btree のインデックスページのコピーの数の期待値を導出する過程で、コピーの数は考え得る最大の値を仮定した (式 4.1参照)。実際にはインデックスページのコピーの数はこれより少なくなると考えられ、その場合には、Fat-Btree の性能はスループット、レスポンスタイム共にさらに向上することが予測される。

しかし、計算による性能解析という方法では、上に述べたような Fat-Btree のより詳細な性能解析は困難であると考えられる。Fat-Btree の詳細な性能解析を行なうためには、Fat-Btree を実際に実装し、性能を評価する必要があると考えられる。

また本研究では、同じページ分配型並列 B-tree の中の B-tree 全コピー方式との性能比較を行なったが、レコード分配型並列 B-tree においてレコードの分配に値域分割を用いる方法も、データ配置に全く偏りが生じなければ良い特性を示すものと考えられる。この

方法に対してデータ配置の偏りが及ぼす影響を考慮した上で、この方法と Fat-Btree との性能比較も行なう必要があると考えられる。

謝辞

本研究を進めるにあたり、終始熱心な御指導を賜わりました横田治夫助教授に心から感謝致します。

また、適切な御助言をして頂きました日比野靖教授に深く感謝致します。

さらに、貴重な御意見、御討論を頂きました杉野栄二助手、宮崎純助手と横田研究室の学生の皆様に心から御礼を申し上げます。

最後に、多くの方々の御援助によって本研究を行なうことができましたことを厚く御礼申し上げます。

付録 A

解析式の詳細について

本付録では、第 4章における Fat-Btree の性能解析に用いた解析式の詳細について示す。B-tree 全コピー方式の解析式については特に示さないが、Fat-Btree と同様にして解析式を求めている。

A.1 スループットの解析

スループットの解析では、まず 1 回の READ 操作と 1 回の WRITE 操作それぞれに要する延べ処理時間を求めて、それを全オペレーションに占める WRITE の割合に応じて掛け合わせ、1 回のオペレーションあたりの平均延べ処理時間を求め、その平均延べ処理時間の逆数をとってスループットを求めている。ここで言う延べ処理時間とは、PE1 台が処理に費やした時間と、処理に必要とした PE 台数の積である。以下で、1 回の READ 操作に要する延べ処理時間と 1 回の WRITE 操作に要する延べ処理時間の求め方について述べる。

A.1.1 READ 操作の処理時間

Fat-Btree のある PE に、検索しているデータへのアクセスパスにあたる第 i 段インデックスページが存在する確率は、式 4.3より $(IP(i) + (N_{PE} - 1))/(N_{PE} \times IP(i))$ なので、ある PE にアクセスパスが i 段目までしかない確率、つまりアクセスパス上の第 i 段イン

デックスページは存在するが第 $(i+1)$ 段インデックスページは存在しない確率は、

$$\frac{IP(i)+(N_{PE}-1)}{N_{PE}\times IP(i)} - \frac{IP(i+1)+(N_{PE}-1)}{N_{PE}\times IP(i+1)} \tag{A.1}$$

となる。

Fat-Btree において、最初に検索を行なっていた PE で、アクセスパス上の第 j段目のインデックスページが存在せず、問い合わせを別の PE に引き継いだ場合、問い合わせが引き継がれた先の PE には第 j段インデックスページが存在することが保証されている。この場合において、問い合わせが引き継がれた先の PE にアクセスパスが j段目から i 段目までしかない確率 $P_{path}(j,i)$ は、式 A.1から次のようになる。

$$P_{path}(j,i) = \frac{1}{\frac{IP(j)+(N_{PE}-1)}{N_{PE}\times IP(j)}} \cdot \left\{ \frac{IP(i)+(N_{PE}-1)}{N_{PE}\times IP(i)} - \frac{IP(i+1)+(N_{PE}-1)}{N_{PE}\times IP(i+1)} \right\}$$

ただし、$i = h-1, h$ のときは、葉ページにコピーが存在しないので、以下のようになる。

$$P_{path}(j,h-1) = \frac{1}{\frac{IP(j)+(N_{PE}-1)}{N_{PE}\times IP(j)}} \cdot \left\{ \frac{IP(h-1)+(N_{PE}-1)}{N_{PE}\times IP(h-1)} - \frac{1}{N_{PE}} \right\}$$

$$P_{path}(j,h) = \frac{1}{\frac{IP(j)+(N_{PE}-1)}{N_{PE}\times IP(j)}} \cdot \frac{1}{N_{PE}}$$

よって、第 j段インデックスページから問い合わせが引き継がれた時に、それ以降その問い合わせに必要な処理時間 $READ(j)$ は、その PE で必要な第 j段から第 i 段までのページアクセスの時間と、次の PE へ問い合わせを引き継ぐためのメッセージセットアップの時間[1]と、次の PE で必要な処理時間の合計で、次のようになる。

$$\begin{aligned} READ(j) = & \sum_{i=j}^{h-2} \{P_{path}(j,i) \times (T_{PA}(j,i) + T_{MS} + READ(i+1))\} \\ & + P_{path}(j,h-1) \times \{T_{PA}(j,h-1) + T_{MS} + T_{PA}(h,h+1)\} \\ & + P_{path}(j,h) \times T_{PA}(j,h+1) \end{aligned} \tag{A.2}$$

ここで、$T_{PA}(j,i)$ は、アクセスパスの j段目から i 段目までのページアクセスに必要な時間で、各段のページのキャッシュヒット率 (式 4.5参照) によって決まる。なお、この場合において、$h+1$ 段目はデータページを意味する。T_{MS}は、PE 間通信のメッセージセットアップに要する時間である。

式 A.2において $j=0$ の場合、つまり $READ(0)$ が、Fat-Btree において READ 操作 1 回に要する延べ処理時間となる。

[1] メッセージの準備と受け取りにかかる時間の合計

A.1.2 WRITE 操作の処理時間

Fat-Btree において、WRITE 操作 (ここでは挿入操作) に要する処理時間は、データを挿入する場所を検索する時間と、挿入するデータを書き込む時間と、コピー間で同期してコミット処理を行なう時間の合計である。データを挿入する場所を検索する時間は READ の場合と同様で、ただ READ の場合よりもデータページ読み込みの為の 1 回分のディスクアクセスが少ないだけである。挿入するデータを書き込む時間と、コピー間で同期してコミット処理を行なう時間は、その WRITE 処理に巻き込まれる PE の数に大きく依存する。

WRITE 処理に巻き込まれる PE の数は、その WRITE 処理によって更新されるページに存在するコピーの数で決まる。その WRITE 処理によってページ分割が生じる場合は、更新されるページは複数となるが、Fat-Btree ではその構造上、子ページにコピーが存在すると親ページにも同じだけコピーが発生するので、更新される内で最も上位のページのコピーの数が WRITE 処理に巻き込まれる PE の数となる。よって、WRITE 処理によってページ分割が Sp 段 $(0 \leq Sp \leq h)$ 生じるとすると、WRITE 処理に巻き込まれる PE の数の期待値 Invは、式 4.1より、

$$\begin{aligned} Inv &= \frac{N_{PE}-1}{IP(h-Sp)} && (Sp < h) \\ Inv &= \frac{N_{PE}-1}{IP(1)} = N_{PE}-1 && (Sp = h) \end{aligned}$$

となる[2]。

WRITE の処理の流れを図 A.1に示す。WRITE の処理は大きく分けて、ディレクトリを辿ってデータを挿入する場所を検索するデータ挿入場所検索部と、更新されるページのコピーがある PE に対してディレクトリ更新の要求のためのメッセージを発送するディレクトリ更新要求発生部と、データのある PE でのデータの挿入とコピーのある各 PE でのディレクトリ更新を実際に行なうディレクトリ更新処理部と、更新処理に関わった全 PE で同期してコミット処理をする 2 相コミット部に分かれる。データ挿入場所検索部に必要な時間に関しては本節の最初で述べた通りであるので、以下ではそれ以外の各部分に必要な処理時間の求め方について述べる。なお、図 A.1中の斜線部については、その処理時間が全体に与える影響が小さいとして、以下の解析式では考慮に入れていない。

[2] この数には元の PE の分は含まれていない。

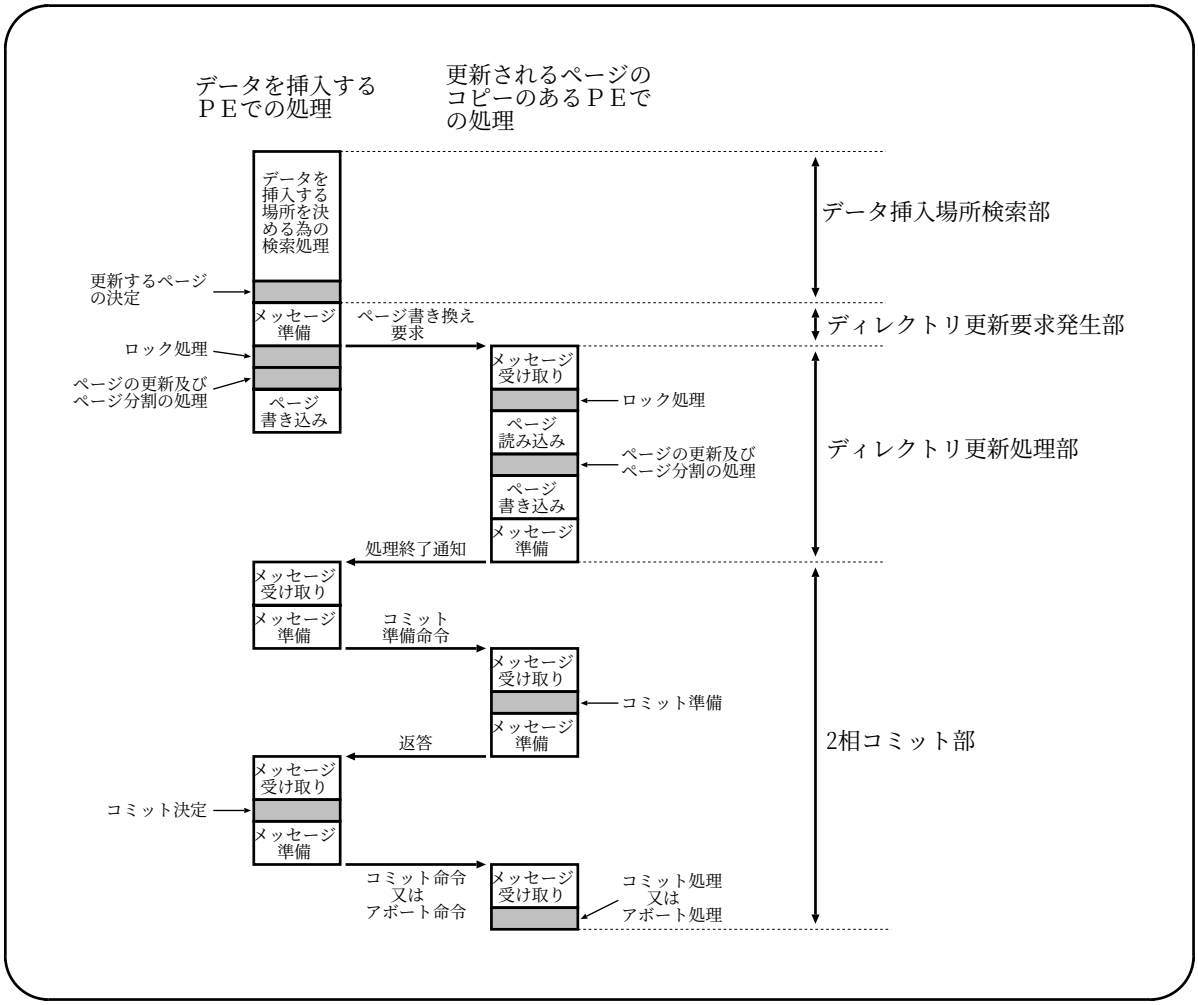


図 A.1: WRITE 処理の流れ ($Inv = 1$ の場合)

ディレクトリ更新要求発生部

ここでは他の更新に関わる PE に対する更新要求のメッセージを作成する。ここで必要な処理時間は、更新に関わる PE の数の分だけのメッセージ準備にかかる時間である。しかし、最初のメッセージを送り出せば、それを受け取った PE ではディレクトリ更新処理を始められるので、2 番目以降のメッセージ準備はディレクトリ更新処理部とオーバラップさせることができる。

ディレクトリ更新処理部

ディレクトリ更新処理部において、データを挿入する PE で必要な処理は、書き換えたページの書き込みである。本解析では、ページの書き込みは、データの永続性を保証するために、メモリではなくディスクに対して行なうとしている。この場合に必要となるディスクアクセスの回数は、分割したインデックスページの分の $(Sp \times 2)$ 回と、分割するページの 1 つ上位で新しいエントリが挿入されるインデックスページの分と、挿入されるデータのデータページへの書き込みの分で、合計 $(2Sp + 2)$ 回である。

一方、更新されるページのコピーのある PE で必要となる処理は、書き換えられるページの読み込みと、書き換え及びページ分割後の新しいページの書き込みである。ここで必要となるページアクセス数は、更新されるページの内の何ページがその PE にコピーされているかに依存する。WRITE 処理によってページ分割が Sp 段生じるとすると、更新されるインデックスページは、第 $(h - Sp)$ 段インデックスページから第 h 段インデックスページ (葉ページ) までの $(Sp + 1)$ ページである。式 4.1より、第 i 段インデックスページ 1 ページあたりのコピーの数の期待値は $(N_{PE} - 1)/IP(i)$ である。この値が 1 より小さい時には、その値がそのまま第 i 段インデックスページがコピーを持つ確率になり、また 1 を上回る場合には、第 i 段インデックスページの全てが 1 つ以上のコピーを持つことになる[3]。したがって、WRITE 処理によって更新される $(Sp+1)$ のインデックスページのうち、第 $(h - Sp)$ 段インデックスページから第 i 段インデックスページ $(h - Sp \leq i \leq h - 1$ [4]$)$ までが他の PE にコピーを持つ確率は、

$$\min\left(1,\ \frac{N_{PE} - 1}{IP(i)}\right) - \min\left(1,\ \frac{N_{PE} - 1}{IP(i+1)}\right)$$

となる。この場合において、コピーの置かれている PE で必要となるページアクセス数は、第 $(h - Sp)$ 段インデックスページから第 i 段インデックスページまでの読み込みの分と、それらのページ分割後の書き込みの分である。ページの読み込みは、もしそのページがメモリ上にキャッシュされていればメモリに対して行なうことができ、そのページアクセスに必要な時間はキャッシュのヒット率 (式 4.5参照) によって決まるが、この場合も書き込みは毎回ディスクに対して行なうので、書き込むページ数分だけのディスクアクセスが必要となる。

[3] これは、均等な Fat-Btree では、同レベル内の各インデックスページが持つコピーの数の差が、高々1 にしかならない為である。

[4] Fat-Btree では葉ページはコピーされない。

コピーの置かれている PE が複数ある場合、更新されるページの内でその PE にコピーされているものの数は PE 毎に異なるので、それら各 PE におけるディレクトリ更新処理に要する時間は異なる。また、データを挿入する PE とコピーのある PE でも、ディレクトリ更新処理に要する時間が異なる。しかし、全ての PE はコミット時に同期を取る必要があるので、結局は一番時間のかかる PE に全ての PE が合わせることになる。よって、システム全体でディレクトリ更新処理に要する総処理時間は、

$$\max(\text{各 PE における処理時間}) \times (Inv + 1)$$

となる。

2 相コミット部

Fat-Btree において、ディレクトリ更新によって更新されるページにコピーがある場合は、2 つ以上の PE がそのディレクトリ更新に巻き込まれることになる。その場合、各コピー間の内容の一貫性を保つために、2 相コミットをする必要がある。この 2 相コミットにおいては、データを挿入する PE 側が調停者 (coordinator)、その他のコピーの置かれている PE の方が参加者 (participant) となる。

調停者と参加者の間では 2 回のメッセージのやりとりを行なうわけだが、各参加者は調停者 1 つと通信すればよいのに対して、調停者は複数 (1 つの場合もあるが) の参加者全てと通信しなければならない。よって多くの場合、調停者がボトルネックになり、参加者側が調停者からのメッセージを待つことになる。本解析では、この際の参加者側のメッセージ待ちの時間も考慮に入れている。また参加者が少ない時には、調停者側でもメッセージ待ちが生じることがあり、本解析はそれも考慮している。

調停者は 2 回のメッセージ通信[5]を Inv個の参加者と行ない、参加者数 Invが少ない時にはそれにメッセージ待ちの時間が加わるので、2 相コミット処理に要する時間は次のようになる。

$$T_{MS} \times 2 \times Inv + (\text{メッセージ待ち時間})$$

一方参加者は、調停者がコミット命令を全ての参加者に送信し終るまで待つ必要はなく、自分のところにコミット命令が届いたらコミットすることができるので、調停者で必要な

[5] ここでは送信と受信を合わせて 1 回の通信と数え、その 1 回のメッセージ通信のセットアップに要する時間を T_{MS} としている。

最後の 0.5 回分[6]の通信 ($T_{MS} \times 0.5 \times Inv$) は必要ない。その代わりに自分に対する調停者からのメッセージ通信の分を加えて、必要な処理時間は以下のようになる。

$$T_{MS} \times 1.5 \times Inv + (\text{調停者側のメッセージ待ちの時間}) + T_{MS}$$

A.2 レスポンスタイムの解析

レスポンスタイムの解析は、スループットの解析と同じ手法で行なっている。両者の違いは、スループットの解析では複数 PE の延べ処理時間を求めているのに対して、レスポンスタイムの解析では PE 間での処理の重複分を除いた時間を求めているというだけである。レスポンスタイムの解析の詳細は、前節と同じ話の繰り返しになるので、ここでは省略する。

[6] 本解析では、メッセージの送信 (準備) に要する時間と受信 (受け取り) に要する時間は同じで、それぞれ $0.5T_{MS}$ であるとしている。

参考文献

[1] G. Copeland and W. Alexander and E. Boughter and T Keller, "Data Placement In Bubba", Proc. of ACM SIGMOD Conf., pp.99–108 (1988).

[2] D. DeWitt and J. Gray, "Parallel Database Systems: The Future of High Performance Database Systems", CACM, Vol. 35, No. 6, pp.85–98 (1992).

[3] S. Ghandeharizadeh and D. J. DeWitt, "Hybrid-Range Partitioning Strategy: A New Declustering Strategy for Multiprocessor Database Machines", Proc. of 16th VLDB Conf., pp.481–492 (1990).

[4] B. Seeger and P. Larson, "Multi-Disk B-trees", Proc. of ACM SIGMOD Conf., pp.436–445 (1991).

[5] C. Xu and F. C. M. Lau, "LOAD BALANCING IN PARALLEL COMPUTERS", Kluwer Academic Publishers (1997).

[6] V. Srinivasan and M. J. Carey, "Performance of B-Tree Concurrency Control Algorithms", Proc. of ACM SIGMOD Conf., pp.416–425 (1991).

[7] 横田 治夫, 宮崎 純, 土屋 由美子, "並列アクティブデータベースエンジン Parade の並列処理", 平成 8 年度科学研究費重点領域研究「高度データベース」松江ワークショップ講演論文集, pp426–433 (1996).

[8] 白砂 丈太郎, 鈴木 博貴, "パーティショニング技術の VLDB への適用", 信学技報 DE97–73 (AI97–40), 電子情報通信学会 (1997).

[9] 金政 泰彦, 宮崎 純, 横田 治夫, "並列データベースシステムにおける更新を考慮したディレクトリ構成", 信学技報 DE97–77 (AI97–44), 電子情報通信学会 (1997).