Vol.2022-DBS-175 No.13
Vol.2022-IFAT-148 No.13
2022/9/9

B^+ 木における同時実行制御手法の性能検証

野原 健汰[†1,a)]　杉浦 健人[†1,b)]　石川 佳治[†1,c)]

概要：ハードウェア技術の進化によりメニーコア環境が主流となった現在は，より効率的な同時実行制御が求められている．代表的な索引構造である B^+ 木に対する同時実行制御手法はいくつか存在するが，それらを最近のメニーコア環境で公平に比較評価した研究は存在しない．具体的には異なるページレイアウトの使用や固定長データに対する最適化の有無，異なる方針のガベージコレクションの使用など，既存研究の実験で行われた比較には同時実行制御以外の面で性能に大きく影響を与える要素が含まれている．そこで本研究では B^+ 木における悲観的・楽観的ロックを用いた 4 種類の同時実行制御手法を統一的に再現実装し，それらの性能比較を通して各制御手法の性質を検証する．

1. はじめに

ハードウェア技術の進化によって，CPU のメニーコア化，メモリの大容量化が進む現在は，その性能を最大限に発揮するマルチスレッドソフトウェアの需要が高まっている．データベースもその 1 つであり，メモリの大容量化によって近年では全てのデータをメモリ上で管理するインメモリデータベースが台頭している．インメモリデータベースでは，従来のデータベースでボトルネックになっていた高価なディスクアクセスを避けることができ，索引性能がより重要になる．

メニーコア環境で性能を向上させるために，索引上では適切な同時実行制御によって並列処理性能を向上させることが求められている．代表的な索引構造である B^+ 木 [9] に対する同時実行制御手法はいくつか提案されている [2, 7]．しかし，これらを最近のメニーコア環境で公平に比較評価した研究は存在しない．既存研究においてこれらの手法が性能評価に用いられる場合があるが，異なるページレイアウトの使用や固定長データによる最適化の有無などにより公平に比較評価されているとは言い難い．

本研究ではまずロックの種類と粒度の観点から B^+ 木における同時実行制御手法を 4 つに分類し，それぞれ統一的に再現実装する．そして，それらを近年提案されている Bw 木 [6] や Bz 木 [1] のようなロックフリー索引と比較し，各制御手法の性質を検証する．

†1 名古屋大学大学院情報学研究科
Graduate School of Informatics, Nagoya University
a) nohara@db.is.i.nagoya-u.ac.jp
b) sugiura@i.nagoya-u.ac.jp
c) ishikawa@i.nagoya-u.ac.jp

表 1 ロックの関係

	S	SIX	X
S	✓	✓	-
SIX	✓	-	-
X	-	-	-

2. 準備

本節では，B^+ 木における同時実行制御の基本的な概念としてロック方式と構造変更時のロックの粒度について説明する．

2.1 ロック方式

ロック方式は悲観的（pessimistic）と楽観的（optimistic）の 2 種類を用いる．本節で使用する排他ロック（exclusive lock，X），排他意図共有ロック（shared with intent exclusive lock，SIX），共有ロック（shared lock，S）の関係を表 1 に示す．

2.1.1 悲観的ロック

悲観的ロックは，書き込み時に排他ロックを，読み込み時に排他意図共有ロックと共有ロックを取得する手法である [3]．実装では図 1 に示すように 64bit 符号なし整数の LSB を排他ロック，2bit 目を排他意図共有ロック，その他の bit を共有ロックを表すカウンタとして扱う．

悲観的ロックは単純なアルゴリズムで並列処理における一貫性を保証できるが，性能向上は期待できない．特に，共有ロックの取得ないし解放時のカウンタ増減は CPU キャッシュの汚染を招き，性能悪化につながることが報告されている [2]．B^+ 木のような木構造では探索時に根ノー

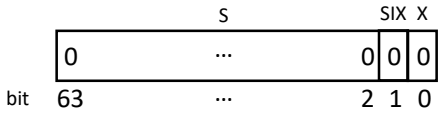

図 1 悲観的ロック

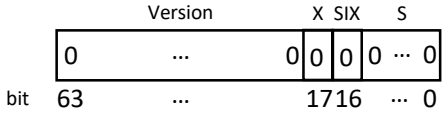

図 2 楽観的ロック

ドから複数の共通する中間ノードを辿らねばならないが，実際にはノード内のレコードを一切変更していないにも関わらず，共有ロックの取得・解放のために探索のたびにそれら中間ノードでキャッシュミスが発生してしまう．

2.1.2 楽観的ロック

楽観的ロックは単純な読み取り操作をロックフリーに実現するための手法である [2]．書き込み時には悲観的ロックと同様に排他ロックを取得し，排他ロックの解放時に内部のバージョン値を増加する．読み込み時はまず操作前に現在のバージョンの値を取得し，操作後もバージョンの値が一致するか確認する．操作後に確認したバージョンの値が異なっている場合や排他ロックが取得されている場合，ロック領域に対して別スレッドによる変更が加えられているため，現在のバージョン値の取得から操作を再試行する．ここで重要となるのは，読み取り時にロックの内部状態を一切変更せず，バージョン値の取得のみが行われる点である．つまり，悲観的ロックのように CPU キャッシュを汚染することなく，読み取りのみであればロックフリーな操作を実現できる．

本研究では後述する同時実行制御において必要となる排他意図共有ロック，共有ロックも使用する．実装では図 2 に示すように 64bit 符号なし整数の下位 16bit を範囲検索用の共有ロックカウンタ，17bit 目を排他意図共有ロック，18bit 目を排他ロック，その他の bit をバージョンとして扱う．

本研究では先行研究 [2] で提案されたアルゴリズムを拡張して使用する．排他ロックを取得する場合は，排他ロックを表す 18bit 目を 1 にするため，CAS（compare-and-swap）命令を用いて $+2^{17}$ に更新する．排他ロックを解放する場合は，排他ロックを表す 18bit 目を 0 にし，更にバージョン番号をインクリメントする必要があるため，取得時と同様 CAS 命令を用いて $+2^{17}$ に更新する．読み込み時には排他ロックが取得されていないこととバージョン情報が更新されていないことを確認する必要がある．そのため，操作前は 18bit 目が 0 になるまで待ち，18bit 目が 0 になったらバージョンを取得する．操作後は 18bit 目が 0 であることと，操作前とバージョンが一致していることを確認する．

表 2 B^+ 木における各ロック方式のデータ変更量（byte）

命令	悲観的	楽観的
read	$8+8\log_M N$	0
insert	$O(M)+8\log_M N$	$O(M)$
update	$16+8\log_M N$	16
delete	$16+8\log_M N$	16
sort	M	M
split	$O(M)+2M$	$O(M)+2M$
merge	$8+O(M)+M$	$8+O(M)+M$

2.1.3 ロック方式の与える影響

B^+ 木における各ロック方式のデータ変更量を表 2 に示す．ここではキーおよびペイロードとロック用変数，レコード位置を表すメタデータを 8 byte，ページサイズを M byte，葉ノード数を N byte とする．また，insert，update，delete では構造変更が起きないと仮定する．

探索が必要な各命令（read，insert，update，delete）では，悲観的ロックを使用する場合は楽観的ロックを使用する場合と比較して，探索時に中間ノードもロック用変数を更新する必要があるため $8\log_M N$ byte の更新が加わる．read は，悲観的ロックを用いる場合は葉ノードにおいてもロック用変数を更新する必要があるため 8 byte の更新が必要になるが，楽観的ロックではデータ変更は発生しない．insert は，レコード挿入時に並べ替えを必要とするため $O(M)$ byte の更新が必要となる．update は，ロック用変数とペイロードの更新が必要になるため，16 byte の更新が必要となる．delete は，ロック用変数とメタデータの更新が必要になるため，16 byte の更新が必要となる．sort は，ページ内全レコードの並び替えが必要になるため，M byte の更新が必要となる．split は，親ノードへのレコードの挿入に $O(M)$ byte，2 つの子ノードの sort に $2M$ byte の更新が必要となる．merge は，merge 元である右子ノードのロックに 8 byte，親ノードのレコードの削除に $O(M)$ byte，merge 先である左子ノードの sort に M byte の更新が必要となる．

表 2 から悲観的ロックは楽観的ロックよりも多くのデータ変更を必要とすることが分かる．特に多くのスレッドからアクセスされやすい中間ノードのヘッダを変更するため，量以上に性能に対して与える悪影響が大きい．

2.2 ロックの粒度

構造変更時は多層ロック（multi-layer lock）と単層ロック（single-layer lock）の 2 種類を用いる．

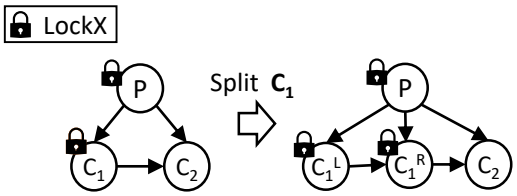

図 3 多層ロックでの split 例

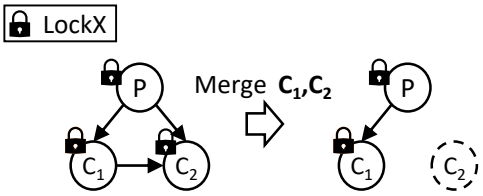

図 4 多層ロックでの merge 例

2.2.1 多層ロック

多層ロックは構造変更のために常に親ノードと子ノードの両方のロックを取得する方法である．子ノードにおいて split/merge を発行した場合，親ノードへのレコードの挿入ないし削除が発生するため，親ノードの排他ロックを取った状態で子ノードの構造を変更することで一貫性を保証する．先行研究 [8] において紹介されているように 2 層のみロックを取得する方法と複数層でロックを取得する方法があるが，本研究では 2 層のみロックを取得する方法を用いる．

多層ロックでの split の例を図 3 に示す．この例においてノード C_1 は split する可能性があるためノード P の排他ロックを保持したまま，ノード C_1 をノード C_1^L とノード C_1^R に split する．ノード P の排他ロックはノード C_1^R のポインタおよび対応する分割キーの挿入後に解放する．

多層ロックでの merge の例を図 4 に示す．この例においてノード C_1 は merge する可能性があるためノード P の排他ロックを保持したままノード C_2 の排他ロックを取得して merge する．その後，ノード P においてノード C_1 の分割キーを持つレコードを削除し，ノード C_2 の分割キーを持つレコードがノード C_1 を指すように更新する．

2.2.2 単層ロック

単層ロックは多層ロックとは違い，操作するノードのみロックを取得する方法である．B^{link} 木 [4] で提案されたように各ノードに最大キーと兄弟リンクを付与することで親ノードのロックを取得せず一貫性を保証できる．B^{link} 木の概観を図 5 に示す．単層ロックでは子ノードにおいて split/merge を発行した場合に親ノードはロックを取得していないため，兄弟リンクを使用して挿入ないし削除すべき適切な親ノードを探す必要がある．

単層ロックでの split の例を図 6 に示す．この例においてノード C_1 は split する可能性があるためノード C_1 をノード

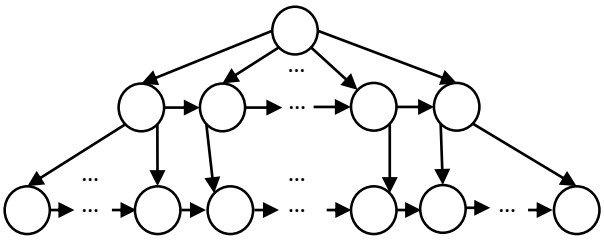

図 5 B^{link} 木概観

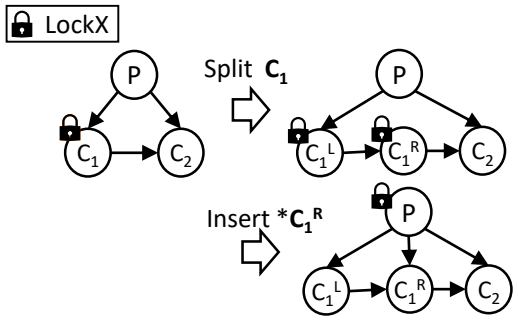

図 6 単層ロックでの split 例

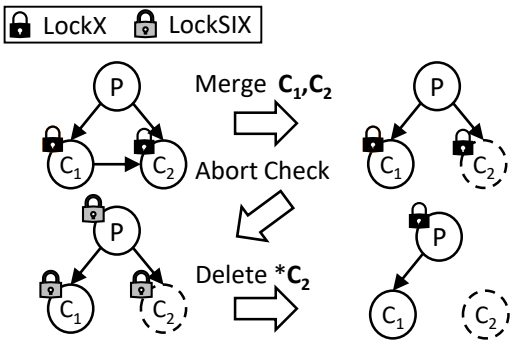

図 7 単層ロックでの merge 例

C_1^L とノード C_1^R に split し，排他ロックを解放する．その後ノード P の排他ロックを取得し，ノード C_1^R のポインタを挿入する．ただし，ノード C_1 を split した際にノード P の排他ロックを取得していなかったため，ノード P が split ないし merge された可能性を考慮する必要がある．ノード P が split/merge されていた場合はノード P の兄弟リンクを使用して適切なノードを探索する．

単層ロックでの merge の例を図 7 に示す．この例においてノード C_1 は merge する可能性があるためノード C_2 の排他ロックを取得し merge する．この時，ノード P が split しており，ノード C_1 を指すレコードとノード C_2 を指すレコードが異なる親ノードに存在している場合は merge をアボートする必要がある．そのため，ノード C_1 とノード C_2 の排他ロックを排他意図共有ロックに更新し，ノード P の排他意図共有ロックを取得する．ただし split 時同様，この時点でノード P は split/merge されている可能性がある

ため，その場合は兄弟リンクを使用して適切なノードを探索する．またこの例ではアボートする必要はないと仮定すると，ノード P の排他意図共有ロックを排他ロックに更新し，ノード C_1 とノード C_2 の排他意図共有ロックを解放する．その後，ノード P においてノード C_2 を指していたレコードをノード C_1 を指すように更新し，ノード C_1 を指していたレコードを削除する．

2.2.3 ロックの粒度が与える影響

単層ロックは多層ロックと比較して主に 3 つの利点を持つ．先述したように適切なノードを探索する必要はあるが，中間ノードにおいてこの探索回数は少なく，影響は限定的である．

1 つ目はロックの占有期間を削減できることである．多層ロックが常に親ノードと子ノードのロックを必要とするのに対し，単層ロックでは基本的に操作するノードのみロックを必要とする．これによりロックの占有による待ち時間を減らすことができる．

2 つ目は根ノードから再試行する必要がないことである．楽観的ロックを用いる場合，多層ロックでは同時書込みによって到達したノードのキー範囲が検索キーを含まない可能性があり，その際は適切なノードへ到達するために根ノードからの再試行が必要となる．しかし，単層ロックでは兄弟リンクを持つため，到達したノードのキー範囲が異なる時でも兄弟リンクを使用して適切なノードへ遷移できる．

3 つ目はスレッド間順序が固定されないことである．先述したように，単層ロックでは構造変更時の親ノードへの反映を非同期に実行できる．このため，スレッド間で構造変更の順序が固定されず，特定のスレッドの処理遅延などの影響を受けない．

3. B+ 木における同時実行制御手法

本節では，前節で説明したロックの種類と粒度の観点から 4 つに分類した B+ 木における同時実行制御手法を説明する．

3.1 PML（Pessimistic Multi-layer Locking）

PML は悲観的ロックを用いて多層ロックで同時実行制御をする手法である [8]．書き込みの探索時に構造変更が起きる可能性のある場合は常に構造変更する．書き込みは探索時に葉ノードの排他ロックを取得し，操作後に排他ロックを解放する．読み込みは探索時に葉ノードの共有ロックを取得し，操作後に共有ロックを解放する．

PML における書き込み時の探索例を図 8(a) に示す．まず，ノード A の排他意図共有ロックを取得し，子ノードを探索する．次にノード A の子ノードであるノード B の排他意図共有ロックを取得し，ノード B の split/merge の可能性を確認する．split/merge の可能性がある場合はノード A とノード B の排他意図共有ロックを排他ロックに更新し，その時点で split/merge を発行する．続いてノード A のロックを解放し，ノード B の子ノードを探索する．ノード B の子ノードであるノード C の排他意図共有ロックを取得し，ノード C の split/merge の可能性を確認する．ノード B 同様に split/merge の可能性がある場合はノード B とノード C の排他意図共有ロックを排他ロックに更新し，その時点で split/merge を発行する．最後にノード B のロックを解放し，ノード C は葉ノードであるためノード C の排他意図共有ロックを排他ロックに更新し，ノード C を返す．このように書き込み時の探索は排他意図共有ロックを使用するため読み取りとロックの競合は発生しない．

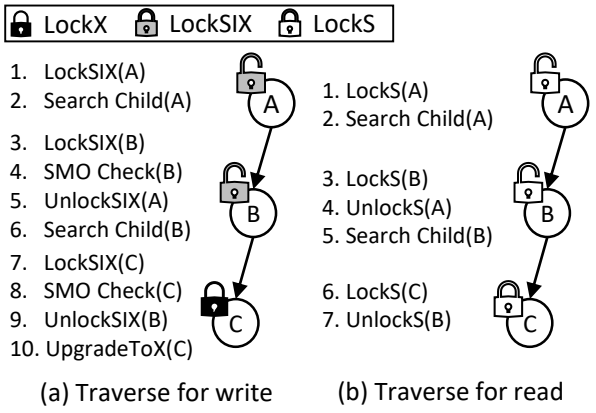

図 8 PML の書き込み時/読み込み時の探索処理

PML における読み込み時の探索例を図 8(b) に示す．まず，ノード A の共有ロックを取得し，子ノードを取得する．次にノード A の子ノードであるノード B の共有ロックを取得し，ノード A の共有ロックを解放する．最後にノード B の子ノードであるノード C の共有ロックを取得し，ノード B の共有ロックを解放する．ノード C は葉ノードであるためノード C を返す．

3.2 OML（Optimistic Multi-layer Locking）

OML は楽観的ロックを用いて多層ロックで同時実行制御をする手法である [5]．書き込みの探索時に構造変更が起きる可能性のある場合は常に構造変更する．書き込みは探索時に葉ノードの排他ロックを取得し，操作後に排他ロックを解放する．読み込みは探索時に葉ノードのバージョンを取得し，操作後に再度バージョンを確認する．ここでバージョンが一致している場合は処理を終え，異なっている場合は再試行する．ただし，ノードのキー範囲が検索キーと異なっているもしくはマージによりノードが削除されている場合は根ノードから再試行する．OML では削除されたページにアクセスする可能性があるため，GC が必要になる．

OML における書き込み時の探索例を図 9(a) に示す．まず，ノード A のバージョンを取得し，子ノードを取得した

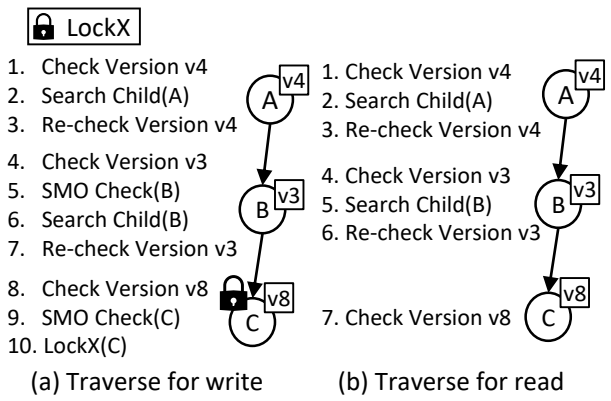

図 9 OML の書き込み時/読み込み時の探索処理

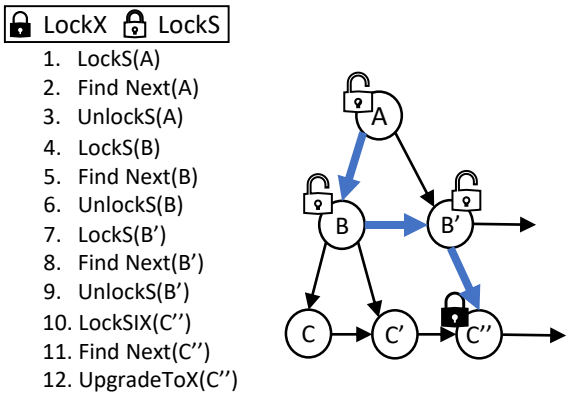

図 10 PSL の書き込み時の探索処理

らノード A のバージョンを再チェックする．ここで，ノード A のバージョンが変更されていたら根ノードから再試行する．次にノード A の子ノードであるノード B のバージョンを取得する．ノード B が split/merge の可能性がある場合はノード A とノード B の排他ロックを取得し，その時点で split/merge を発行する．この時にノード A やノード B のバージョンが変更されていたら根ノードから再試行する．その後，ノード B の子ノードを取得し，ノード B のバージョンを再チェックする．ここで，ノード A 同様ノード B のバージョンが変更されていたら根ノードから再試行する．続いて，ノード B の子ノードであるノード C のバージョンを取得する．ノード B 同様 split/merge の可能性がある場合はノード B とノード C の排他ロックを取得し，その時点で split/merge を発行する．この時にノード B やノード C のバージョンが変更されていたら根ノードから再試行する．最後にノード C は葉ノードであるため，ノード C の排他ロックを取得してノード C を返す．ここでもノード C のバージョンが変更されていたら根ノードから再試行する．

OML における読み込み時の探索例を図 9(b) に示す．読み込み時の探索は書き込み時とは異なり構造変更のチェックが不要であり，葉ノードではロックを取得しない．そのため，葉ノードであるノード C に到達した後はノード C とノード C のバージョンを返す．

3.3 PSL（Pessimistic Single-layer Locking）

PSL は悲観的ロックを用いて単層ロックで同時実行制御をする手法である [4]．構造変更は葉ノードからボトムアップに行う．書き込みは探索時に葉ノードの排他ロックを取得し，操作後に排他ロックを解放する．読み込みは探索時に葉ノードの共有ロックを取得し，操作後に共有ロックを解放する．PSL では削除されたページにアクセスする可能性があるため，GC が必要になる．

PSL における書き込み時の探索例を図 10 に示す．まず，ノード A の共有ロックを取得し，子ノードを取得した後ロックを解放する．次に，ノード A の子ノードであるノード B の共有ロックを取得する．ただし，ノード A のロック解放からノード B の共有ロック取得までにノード B は split/merge している可能性があるため，兄弟リンクを辿って適切なノードを探索する必要がある．この例ではノード B が split して探索キーがノード B′ に存在していると仮定する．ノード B に探索キーが存在しないため，ノード B のロックを解放し，ノード B′ の共有ロックを取得する．ノード B′ に探索キーが存在しているため，子ノードを取得し，ノード B′ のロックを解放する．最後に，ノード B′ の子ノードであるノード C″ は葉ノードであるため排他意図共有ロックを取得し，先ほどと同様に適切なノードを探索する．この例ではノード C″ に探索キーが存在すると仮定すると，ノード C″ の排他意図共有ロックを排他ロックに変更し，ノード C″ を返す．

PSL における読み込み時の探索は，PSL における書き込み時の探索処理とほとんど同様であり，違いは葉ノードの排他意図共有ロックを共有ロックに変更するのみである．

3.4 OSL（Optimistic Single-layer Locking）

OSL は楽観的ロックを用いて単層ロックで同時実行制御をする手法である [2]．構造変更は葉ノードからボトムアップに行う．書き込みは探索時に葉ノードの排他ロックを取得し，操作後に排他ロックを解放する．読み込みは探索時に葉ノードのバージョンを取得し，操作後に再度バージョンを確認する．ここでバージョンが一致している場合は処理を終え，異なっている場合は兄弟リンクを使用して適切なノードを探索する．OSL では削除されたページにアクセスする可能性があるため，GC が必要になる．

OSL の書き込みの探索処理例を図 11 に示す．まず，ノード A のバージョンを取得し，子ノードを取得した後バージョンを再チェックする．ここで，バージョンが異なる場合，再びノード A のバージョンと子ノードを取得する．次に，ノード A の子ノードであるノード B のバージョンを取得する．ただし，ノード B は split してノード B とノード B′ に分かれており，その分割キーがノード A を読んだ

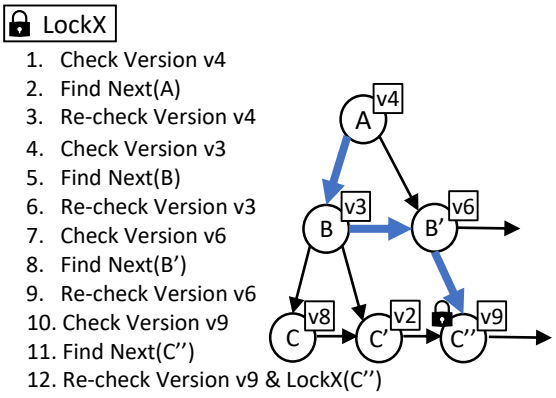

図 11 OSL の書き込みの探索処理

時点でノード A に入っていなかった可能性がある．この例ではノード B が split して探索キーがノード B′ に存在していると仮定する．ノード B に探索キーが存在しないため，兄弟リンクを辿ってノード B′ を取得する．ここで，ノード B のバージョンを再チェックし，バージョンが異なる場合再びノード B のバージョンと兄弟リンクを取得する．続いて，ノード B′ のバージョンを取得し，子ノードを取得する．この後ノード B′ のバージョンを再チェックし，バージョンが異なる場合再びノード B′ のバージョンと子ノードを取得する．最後に，ノード B′ の子ノードであるノード C″ のバージョンを取得し，先ほどと同様に適切なノードを探索する．この例では C″ に探索キーが存在すると仮定すると，ノード C″ は葉ノードであるためノード C″ のバージョンの再チェックと排他ロックの取得を CAS 命令を用いて同時に行い，ノード C″ を返す．

OSL における読み込み時の探索は，OSL における書き込み時の探索処理とほとんど同様であり，違いは葉ノードの排他ロックを取得しないのみである．

4. 関連研究

近年ロックによる占有期間を減らすロックフリー索引に注目が集まっている．本節では，ロックフリー索引として Bw 木と Bz 木について説明する．

Bw 木は B^+ 木の上書き更新におけるキャッシュミスを減らすために差分更新によって書き込みおよび構造変更するロックフリーな索引構造である．Bw 木はキャッシュミスを減らすためにベースノードに対する更新を差分レコードの単方向連結リストとして保持する．差分レコードは挿入や更新，削除だけでなく構造変更操作も表すことができ，ある閾値以上の長さになると統合される．また，差分レコードを挿入する際に発生する親ノードの更新コストを減らすためにマッピングテーブルを作成する．これは木構造を仮想化したものであり，これにより更新をマッピングテーブル上の CAS 命令のみで行うことができる．

Bz 木は B^+ 木を拡張する形で提案された不揮発性メモリ上でも動作するロックフリー索引である．B^+ 木をロックフリーで更新するためには，通常複数ワードの更新をアトミックに行う必要がある．Bz 木はこれに対し，MwCAS（multi-word compare-and-swap）命令を不揮発性メモリ向けに拡張した PMwCAS（persistent multi-word compare-and-swap）命令を使用する．Bz 木は中間ノードでは上書き更新するが，葉ノードでは追記更新することで高い書き込み性能を持つ．葉ノードではソート領域と未ソート領域を持ち，未ソート領域に追記されるレコードを適宜ソート領域に移動する．読み取り時にはまず未ソート領域を線形探索し，探索キーが存在しなければソート領域を二分探索する．

5. まとめと今後の方針

本稿では，ロックの種類と粒度の観点から 4 つに分類した B^+ 木における同時実行制御手法を紹介した．今後は，紹介した PML，OML，PSL，OSL を実装し，各同時実行制御の性能を検証する．また，これらを近年提案されている Bw 木や Bz 木のようなロックフリー索引と比較し，その性質を議論する．

謝辞 本研究は JSPS 科研費 JP20K19804，JP21H03555，JP22H03594 の助成，および国立研究開発法人新エネルギー・産業技術総合開発機構（NEDO）の委託業務（JPNP16007）の結果得られたものである．

参考文献

[1] Arulraj, J., Levandoski, J., Minhas, U. F. and Larson, P.-A.: BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory, *PVLDB*, Vol. 11, No. 5, pp. 553–565 (2018).
[2] Cha, S. K., Hwang, S., Kim, K. and Kwon, K.: Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems, *Proc. VLDB*, pp. 181–190 (2001).
[3] Goetz Graefe: *On Transactional Concurrency Control*, Morgan & Claypool Publishers (2019).
[4] Lehman, P. L. and Yao, S. B.: Efficient Locking for Concurrent Operations on B-Trees, *ACM TODS*, Vol. 6, No. 4, pp. 650–670 (1981).
[5] Leis, V., Scheibner, F., Kemper, A. and Neumann, T.: The ART of Practical Synchronization, *Proc. DaMoN*, pp. 1–8 (2016).
[6] Levandoski, J., Lomet, D. B. and Sengupta, S.: The Bw-Tree: A B-tree for New Hardware Platforms, *Proc. ICDE*, pp. 302–313 (2013).
[7] Rao, J. and Ross, K. A.: Making B^+-Trees Cache Conscious in Main Memory, *Proc. SIGMOD*, pp. 475–486 (2000).
[8] Srinivasan, V. and Carey, M. J.: Performance of B-Tree Concurrency Control Algorithms, *Proc. SIGMOD*, pp. 416–425 (1991).
[9] 北川 博之：データベースシステム，昭晃堂 (1996).