B^+ 木における同時実行制御手法の統一的な再現実装及び性能検証

野原　健汰† 　鈴木　駿也† 　杉浦　健人† 　石川　佳治† 　陸　 可鏡†

† 東海国立大学機構名古屋大学大学院情報学研究科　〒 464–8603 愛知県名古屋市千種区不老町

Email: {nohara, ssuzuki, lu}@db.is.i.nagoya-u.ac.jp, {sugiura, ishikawa}@i.nagoya-u.ac.jp

あらまし　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，それを活かすための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これまでにいくつか提案されているが，それらを最近のメニーコア環境で詳細に実装比較した研究は存在しない．実際に既存研究の実験で行われた比較には，ページレイアウトの違いやレコードメタデータの有無のような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．そのため本研究では 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．実験の結果，悲観的ロックを使用する手法は読取り時に共有ロックを取得するためキャッシュミスを誘発しやすく，マルチスレッド環境において性能が向上しないことを確認した．一方，楽観的ロックを用いる手法はマルチスレッド環境においても性能が向上しやすく，単純なワークロードでは近年提案されているロックフリー索引よりも高い性能をもつことを確認した．

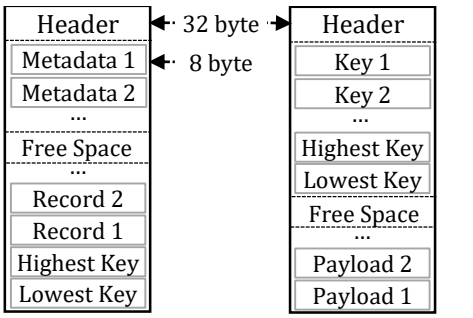
キーワード　B^+ 木，索引構造，同時実行制御．

1　はじめに

ハードウェア技術の進化によってメモリの大容量化や CPU のメニーコア化が進み，近年では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の台頭により従来のデータベースのボトルネックであった高価なディスクアクセスが解消され，索引性能がより重要視されるようになってきている．また CPU のメニーコア化に伴い，その性能を最大限活用するための高い同時実行制御性能が求められるようになってきている．

代表的な索引構造である B^+ 木 [1] に対する同時実行制御手法はこれまでにいくつか提案されている [2–6]．一方，これらを最近のメニーコア環境で詳細に実装比較した研究は存在しない．実際に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．例えばページレイアウトの違いは格納できるレコード数に影響を与え，レコードメタデータの有無はレコードへのアクセス方法に影響を与える．また，その他にも一部の命令が未実装であったり異なる方針のガベージコレクションを使用していたりするなど既存研究において公平に比較評価されているとは言い難い．

そこで本研究では B^+ 木における 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．まずロック方式および粒度の観点から B^+ 木における同時実行制御手法を 4 つに分類し，それぞれを統一的に再現実装する．ここではページレイアウトを統一させたり同じガベージコレクションを使用したりするなど同時実行制御以外の面で性能に影響を与えないようにする．その後各制御手法を近年提案されている Bw 木 [5] や Bz 木 [6] のようなロックフリー索引と比較し，その性質を検証する．



(a) 可変長キー対応　(b) 固定長キー対応

図 1　ノードレイアウト

2　準　　備

本章では，本研究で扱うノードレイアウトおよび同時実行制御の基本的な構成要素であるロック方式と構造変更時のロック粒度について説明する．

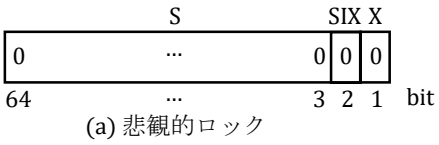
2.1　ノードレイアウト

本研究では同時実行制御手法以外の面で性能に影響を与えないようにするためノードレイアウトを統一する．本研究で扱うノードレイアウトは可変長キー対応したものと固定長キーのみに対応したものの 2 種類あり，それぞれ図 1 に示す．

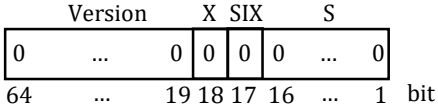
図 1(a) に示す可変長キー対応したノードは大きく分けてヘッダー，メタデータ配列，空き領域，レコード領域の 4 つの領域に分かれており，ページサイズは 1024 byte を基本としている．ヘッダーは 32 byte でレコード数や兄弟ノードのポインタなどのノード情報を管理する．メタデータ配列は各 8 byte でレコード位置やキーの長さなどのレコード情報を管理する．レコード領域では末尾でノードの最小キーおよび最大キーを管理し，末尾から順にレコードを管理する．

表 1　ロックの関係

	S	SIX	X
S	✓	✓	-
SIX	✓	-	-
X	-	-	-



(a) 悲観的ロック



(b) 楽観的ロック

図 2　ロック方式

可変長キー対応したノードでは，レコードへのアクセスは常にメタデータを経由して行われる．メタデータ配列は常にキーの昇順に整列されており，探索は二分探索を用いて行われる．また，葉ノードでは削除したレコードをメタデータのフラグを用いて管理し，中間ノードでは削除したレコードのメタデータを即時削除する．

図 1(b) に示す固定長キーのみに対応したノードは大きく分けてヘッダー，キー配列，空き領域，ペイロード領域の 4 つの領域に分かれており，こちらもページサイズは 1024 byte を基本としている．ヘッダーは可変長キー対応のノードと同様 32 byte でレコード数や兄弟ノードのポインタなどのノード情報を管理する．キー配列では各キーを昇順に管理し，末尾にはノードの最大キーおよび最小キーをもつ．ペイロード領域では末尾からキー配列の順番に合わせてペイロードを管理する．

2.2　ロック方式

本研究では悲観的（pessimistic）と楽観的（optimistic）の 2 種類のロックを使用し，それぞれ排他ロック（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）を用いて実装する．排他ロック，排他意図共有ロック，共有ロックの関係を表 1 に示す．

2.2.1　悲観的ロック

悲観的ロックは書込み時および読取り時の両方でロックを取得するロック方式である [7]．書込み時には排他ロックを取得し，読取り時には排他意図共有ロックまたは共有ロックを取得する．実装では図 2(a) に示すように，64bit 符号なし整数の最下位 bit 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一貫性を保証できる一方，メニーコア環境における性能向上は期待できない．レコードに一切変更を加えない読取り時にも共有ロックカウンタの増減が必要であり，読取りが主となるワークロードにおいてもキャッシュミスが頻発する．特に B^+ 木のような木構造においては，レコードへの変更が少ない索引層におけるキャッシュヒット率の向上が性能改善に直結するが，悲観的ロックでは索引層においてもロック更新によるキャッシュミスが頻発するため性能が悪化しやすい．

表 2　悲観的・楽観的ロックのデータ変更量（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	$2M+O(M)$	$2M+O(M)$
merge	$8+M+O(M)$	$8+M+O(M)$

2.2.2　楽観的ロック

楽観的ロックは書込み時のみロックを取得するロック方式である [4]．書込み時には排他ロックを取得して書込み後にバージョン値を更新し，読取り時には操作前後のバージョン値を比較して一致していれば操作を終了し，異なっていれば操作を再試行する．本研究では先行研究 [4] で提案されたアルゴリズムを拡張し，後述する同時実行制御において必要であるため排他意図共有ロックおよび共有ロックも使用する．実装では図 2(b) に示すように，64bit 符号なし整数の下位 16bit を共有ロックを表すカウンタ，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 をバージョン値として扱う．

楽観的ロックは悲観的ロックと比較してアルゴリズムがやや複雑になるが，メニーコア環境における性能向上が期待できる．単純な読取り時は操作前後のバージョン値を取得するのみでありロックの更新が行われないため，CPU キャッシュが汚染されない．つまり，単純な読取りであればロックフリーに実行できるためメニーコア環境においてキャッシュを有効活用でき，性能向上につながりやすい．

2.2.3　ロック方式による影響

悲観的・楽観的ロック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，ロック，メタデータをそれぞれ 8 byte，ページサイズを M byte，葉ノード数を N と仮定する．また，insert，update，delete はそれによって構造変更が起きないときの値を示す．

表 2 から基本的に悲観的ロックは楽観的ロックよりも多くデータを変更することが分かる．悲観的ロックは探索が必要な各命令（read，insert，update，delete）において中間ノードでもロックを取得するため，楽観的ロックを使用する場合と比較して $8\log_M N$ byte のデータ変更が追加で必要になる．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を更新するため，CPU キャッシュの汚染を招き性能悪化につながりやすい．また，悲観的ロックは read 命令であっても葉ノードにおいて共有ロックを取得するため楽観的ロックと比較して追加で 8 byte のデータ更新が必要になる．insert，update，delete は葉ノードにおいては悲観的ロックも楽観的ロック同様のデータ変更量である．insert 命令は葉ノードの新規レコード挿入によってノード内の

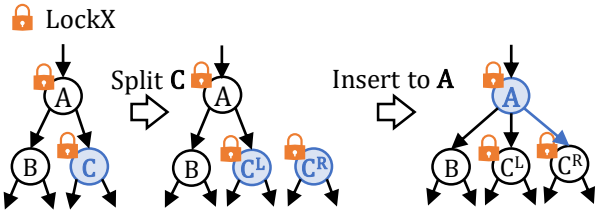


図 3　多層ロックにおける split

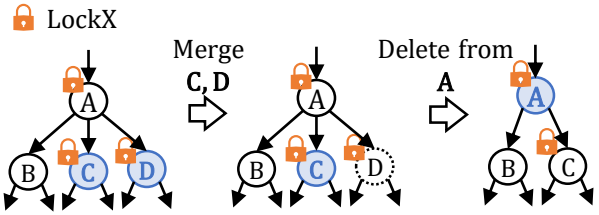


図 4　多層ロックにおける merge

各レコードの並び替えが起こるため $O(M)$ byte のデータ更新が必要になる．update 命令はペイロードとロックの更新によって合計 16 byte のデータ更新が必要になる．delete 命令はメタデータとロックの更新によって合計 16 byte のデータ更新が必要になる．

探索が不要な各命令（sort，split，merge）は悲観的ロックと楽観的ロックで同様のデータ変更量である．sort 命令はノード内の全レコードの並び替えが必要になるため，M byte のデータ更新が必要になる．split 命令は 2 つの子ノードへの分割に合計 $2M$ byte，更に親ノードへ分割キーを挿入するため追加で $O(M)$ byte のデータ更新が必要になる．merge 命令は右子ノードのロックに 8 byte，左子ノードへのレコードコピーに M byte，更に親ノードから分割キーを削除するため追加で $O(M)$ byte のデータ更新が必要になる．

2.3　ロック粒度

本研究では構造変更時に多層ロック（multi-layer lock）と単層ロック（single-layer lock）の 2 種類のロック粒度によって排他制御を行う．B^+ 木における構造変更は，空き領域が少なく新規レコードの挿入ができない場合に 1 つのノードを 2 つに分割する split 操作と空き領域が多すぎる場合に 2 つのノードを 1 つに統合する merge 操作がある．

2.3.1　多層ロック

多層ロックは構造変更時に親ノードと子ノードの排他ロックを同時に取得する方式である．子ノードに対して構造変更が必要な場合，親ノードに対してもその構造変更を反映する必要があるため，親ノードのロックを取得し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．再現実装では先行研究 [8] において紹介されているように探索時に構造変更が必要になりそうな場合は積極的に構造を変更することで最大で 2 層のみ排他ロックを同時に取得する．

多層ロックにおける split の動作例を図 3 に示す．ノード C の空き領域が少ないと仮定すると，その親ノード A と共に排他ロックを取得する．その状態でノード C に対して split を発行し，ノード C^L とノード C^R に分割する．その後ノード C^R の分割キーとノード C^R を指すポインタをノード A に挿入し，操作を終了する．

多層ロックにおける merge の動作例を図 4 に示す．ノード C の空き領域が多いと仮定すると，その親ノード A と兄弟ノード D と共に排他ロックを取得する．その状態でノード C に対して merge を発行し，ノード D のレコードをノード C と統合させる．その後ノード D の分割キーとノード D を指すポインタをノード A から削除し，操作を終了する．

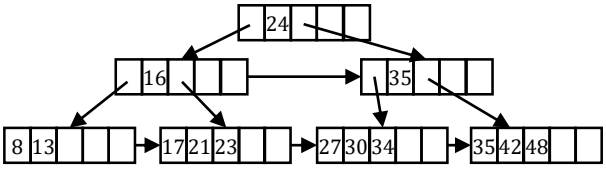


図 5　B^{link}　木

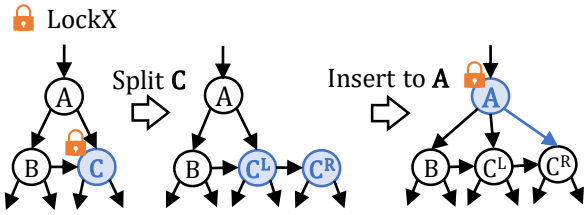


図 6　単層ロックにおける split

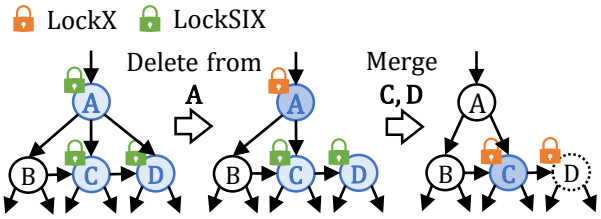


図 7　単層ロックにおける merge

2.3.2　単層ロック

単層ロックは構造変更時に親ノードと子ノードの排他ロックを別々に取得する方式である．これは B^{link} 木 [2] を用いることで実現できる．B^{link} 木は図 5 で示すように中間ノードにも兄弟リンクをもつことで，構造変更の途中状態を読んでも適切なノードに辿り着くことが可能である．

単層ロックにおける split の動作例を図 6 に示す．ノード C の空き領域が少ないと仮定すると，まずノード C の排他ロックを取得する．その状態でノード C に対して split を発行し，ノード C^L とノード C^R に分割する．ノード C^L とノード C^R の排他ロックを解放し，親ノード A の排他ロックを取得する．その後ノード C^R の分割キーとノード C^R を指すポインタをノード A に挿入し，操作を終了する．

単層ロックにおける merge の動作例を図 7 に示す．ノード C の空き領域が多いと仮定すると，その親ノード A と兄弟ノード D と共に排他意図共有ロックを取得する．その状態でノード C

とノード D が merge 可能であるかを確認する．merge 可能であれば親ノード A の排他意図共有ロックを排他ロックに更新し，そうでなければノード A，ノード C，ノード D の排他意図共有ロックを解放して操作を終了する．merge 可能である場合はまず親ノード A からノード D の分割キーとノード D へのポインタを削除する．その後親ノード A の排他ロックを解放し，ノード C およびノード D の排他意図共有ロックを排他ロックに更新する．その状態でノード C に対して merge を発行し，ノード D のレコードをノード C と統合させて操作を終了する．

2.3.3 ロック粒度による影響

単層ロックは多層ロックと比較して主に 3 つの利点をもつ．これらの利点は特にメニーコア環境において性能に影響を与えやすいと考えられる．

1 つ目は構造変更時のロックの占有期間を削減できることである．単層ロックは基本的に操作するノードのみロックを取得するため，多層ロックと比べて構造変更時のロックの占有期間が少ない．特に親ノードの排他ロックを取得しないため，複数スレッドからアクセスを受けやすい中間ノードのロック占有期間を削減でき，性能向上につながりやすい．

2 つ目はスレッド間の順序が固定されないことである．単層ロックは基本的に構造変更時の各層の更新を非同期に行うことができるため，スレッド間で各層の更新順序が入れ替わっても問題ない．これによって，特定のスレッドの処理遅延などの影響を受けない．

3 つ目はボトムアップに構造変更可能なことである．先述した通り単層ロックは基本的に構造変更時の各層の更新を非同期に行うことができるため，葉ノードに書込みをする際に構造変更が必要ならばボトムアップに構造変更できる．これによって探索時に無駄な分岐を減らすことができる．多層ロックでこれを実現するためには最大で根ノードから探索した全てのノードに対するロックを取得する必要があり，これは効率が悪い．このため本研究では探索時に構造変更が起こりそうな場合に積極的に構造を変更している．

3 B^+ 木における同時実行制御手法

本章では，ロック方式および粒度の観点から分類した 4 種類の同時実行制御手法について説明する．

3.1 Pessimistic Multi-Layer Locking（PML）

Pessimistic Multi-Layer Locking（PML）は悲観的ロックを用いて探索し，多層ロックを用いて構造を変更する同時実行制御手法である [8]．悲観的ロックを使用するため，基本的にノードに書込む場合は排他ロックを取得し，ノードからレコードを読む場合は共有ロックを取得する．また探索中に子ノードの変更が行われることを防ぐため，常に 2 層ずつロックを取りながら探索する．書込み時はその書込みによって構造変更が発生する可能性を考慮して，探索時にその可能性がある場合はその場で構造を変更する．このため書込み時は排他意図共有ロックを使用し，読取り時は共有ロックを使用して 2 層ずつロックを取りながら探索する．

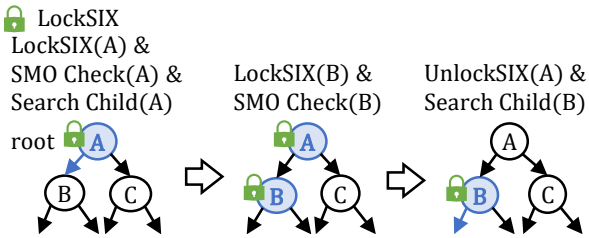


図 8　PML における書込み時の探索

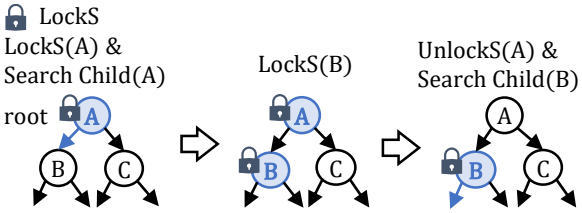


図 9　PML における読取り時の探索

PML における書込み時の探索の動作例を図 8 に示す．まず根ノードであるノード A の排他意図共有ロックを取得し，ノード A に構造変更の可能性があるかを確認する．ここでノード A に構造変更の可能性がある場合はノード A の排他意図共有ロックを排他ロックに更新して構造を変更する．次にノード A から適切な子ノードを探索する．ここでノード B が得られたとすると，ノード B の排他意図共有ロックを取得しノード B に構造変更の可能性があるかを確認する．同様にノード B に構造変更の可能性がある場合はノード A およびノード B の排他意図共有ロックを排他ロックに更新し，merge の場合は更にノード C の排他ロックを取得して構造を変更する．その後ノード A の排他意図共有ロックを解放し，ノード B から適切な子ノードを探索する．同様の操作を葉ノードに到達するまで繰り返す．

PML における読取り時の探索の動作例を図 9 に示す．まず根ノードであるノード A の共有ロックを取得し，ノード A から適切な子ノードを探索する．ここでノード B が得られたとするとノード B の共有ロックを取得する．その後ノード A の共有ロックを解放し，ノード B から適切な子ノードを探索する．同様の操作を葉ノードに到達するまで繰り返す．

3.2 Optimistic Multi-Layer Locking（OML）

Optimistic Multi-Layer Locking（OML）は楽観的ロックを用いて探索し，多層ロックを用いて構造を変更する同時実行制御手法である [9]．楽観的ロックを使用するため，基本的にノードに書込む場合は排他ロックを取得し，ノードからレコードを読む場合は前後のバージョン値を比較する．PML 同様，書込み時はその書込みによって構造変更が発生する可能性を考慮して，探索時にその可能性がある場合はその場で構造を変更する．ただし PML とは異なり，探索時は基本的にバージョン値の比較のみでロックを取らないため構造変更が必要な場合はその場で親ノードおよび子ノードの 2 層の排他ロックを取得する．また，他スレッドが行った構造変更によって探索中のキーがノードの範囲外になってしまった場合や対象ノードが削除予定であ

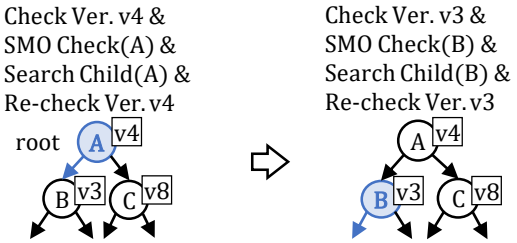


図 10　OML における書込み時の探索

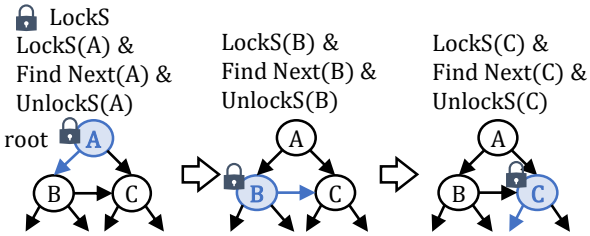


図 11　PSL における書込み時および読取り時の探索

る場合には根ノードから再試行する．

OML における書込み時の探索の動作例を図 10 に示す．まず根ノードであるノード A のバージョン値を取得し，ノード A に構造変更の可能性があるかを確認する．ここでノード A に構造変更の可能性がある場合はノード A の排他ロックを取得して構造を変更する．次にノード A から適切な子ノードを探索する．探索後にノード A のバージョン値を再度取得し，最初に取得したバージョン値と比較する．ここでバージョン値が異なる場合は再試行する．ノード A の操作前後のバージョン値が一致しており，探索によってノード B が得られたとすると，ノード B のバージョン値を取得しノード B に構造変更の可能性があるかを確認する．同様にノード B に構造変更の可能性がある場合はノード A およびノード B の排他ロックを取得し，merge の場合は更にノード C の排他ロックを取得して構造を変更する．その後ノード B から適切な子ノードを探索し，ノード B の操作前後のバージョン値が一致しているかを確認する．ここでも操作前後のバージョン値が異なる場合は再試行する．同様の操作を葉ノードに到達するまで繰り返す．OML における読取り時の探索は書込み時の探索とほとんど同様であり，違いは構造変更の確認が不要なことであるため，ここでは具体的な説明を省略する．

3.3　Pessimistic Single-Layer Locking（PSL）

Pessimistic Single-Layer Locking（PSL）は悲観的ロックを用いて探索し，単層ロックを用いて構造を変更する同時実行制御手法である [2]．PML 同様悲観的ロックを使用するため，基本的にノードに書込む場合は排他ロックを取得し，ノードからレコードを読む場合は共有ロックを取得する．PSL は B^{link} 木を使用するため，探索中に子ノードの変更が行われても兄弟リンクを辿って適切なノードに辿り着くことが可能である．また B^{link} 木はボトムアップに構造変更が可能であるため，PSL は探索中に構造を変更しない．このため書込み時や読取り時に関わらず 1 層ずつ共有ロックを取りながら探索する．

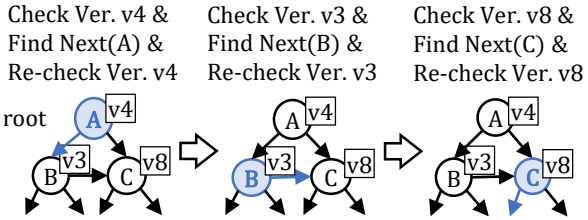


図 12　OSL における書込み時および読取り時の探索

PSL における書込み時および読取り時の探索の動作例を図 11 に示す．まず根ノードであるノード A の共有ロックを取得し，ノード A から適切な子ノードを探索した後共有ロックを解放する．ここでノード B が得られたとするとノード B の共有ロックを取得する．PSL では 1 層ずつロックを取得するため，辿り着いたノードが適切なノードではない可能性がある．例えばノード B の共有ロック取得前にノード B が構造を変更し，探索キー範囲がノード C に移動していたとする．この場合はノード B の共有ロックを解放後に兄弟リンクを使用してノード C に移動する．このように探索中は適切なノードに移動できるように常に最大キーを確認しながら兄弟リンクを辿るか子ノードに移動するかを判定する．同様の操作を葉ノードに到達するまで繰り返す．

3.4　Optimistic Single-Layer Locking（OSL）

Optimistic Single-Layer Locking（OSL）は楽観的ロックを用いて探索し，単層ロックを用いて構造を変更する同時実行制御手法である [4]．OML 同様楽観的ロックを使用するため，基本的にノードに書込む場合は排他ロックを取得し，ノードからレコードを読む場合は前後のバージョン値を比較する．OSL は PSL 同様，B^{link} 木を使用するため探索中に構造を変更しない．

OSL における書込み時および読取り時の探索の動作例を図 12 に示す．まず根ノードであるノード A のバージョン値を取得し，ノード A から適切な子ノードを探索した後再度バージョン値を確認する．ここでバージョン値が異なる場合は再試行する．探索によってノード B が得られたとするとノード B のバージョン値を取得する．OSL は PSL 同様 1 層ずつロックを取得するため，辿り着いたノードが適切なノードではない可能性がある．ノード B が構造を変更して探索キー範囲がノード C に移動していたとすると，ノード B のバージョン値を再度確認して，一致していればノード C に移動する．このときバージョン値が異なる場合はノード B のバージョン取得から再試行する．同様の操作を葉ノードに到達するまで繰り返す．

4　性能評価

本章では，先述した B^+ 木における同時実行制御手法の性能をロックフリー索引と比較して評価する．なお，本稿で紹介する実験は全て可変長キー対応したノードを使用する．

4.1　実験設定

本研究では 1 台のサーバを用いて複数のワークロードで実験する．本研究で用いるサーバの構成を表 3 に示す．また，

表 3　実験用サーバの構成

CPU	Intel(R) Xeon(R) Gold 6258R (two sockets)
RAM	DIMM DDR4 (Registered) 2933 MHz (16GB × 12)
OS	Ubuntu 20.04.5 LTS
Compiler	GCC 9.4.0

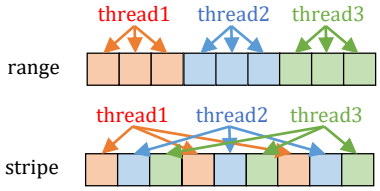


図 13　キー分割方法

本研究で扱うワークロードを表 4 に示す．YCSB-A，YCSB-B，YCSB-C は YCSB（Yahoo! Cloud Serving Benchmark） [10] のワークロード A，B，C をそれぞれ表し，一括挿入によって索引を構築した後に読み書きをする．YCSB ワークロードにおける書込みは全て update 操作である．Update-Scan Mixed は一括挿入によって索引を構築した後，update 操作と scan 操作を同時に発行する．scan 操作はランダムに選択した開始キーから 10^4 個のペイロードを取得する．Construction は書込みのみのワークロードであり，索引が空の状態から write 操作によって索引を構築する．各スレッドが書込みを担当するキーの分割方法として range および stripe があり，図 13 にその概要を示す．Destruction は削除のみのワークロードであり，一括挿入によって索引を構築した後，delete 操作によって全レコードを削除する．各ワークロードでは各スレッドが最大 10^7 回操作し，総実行時間を総実行回数で割ってスループットを計算する．

4.2　YCSB

図 14 に YCSB ワークロードにおいてキーの偏りがない状態でスレッド数を変化させた時のスループットを示す．各結果から OML，OSL，Bz 木，Bw 木は線形増加していることが分かる．YCSB ベンチマークでは書込み操作は update 操作のみであるが，PML または OML は空き領域がない場合は積極的に構造を変更するため，特に YCSB-A ワークロードにおいてスレッド数が増えると多層ロックと単層ロックの性能差が顕著になっていると考えられる．楽観的ロックを使用する OML および OSL が線形増加している一方，PML および PSL はスレッド数を増やしても性能は向上しないことが分かる．特に図 14(c) の YCSB-C は read 操作のみであるにも関わらず性能が向上しない．これは，探索時に共有ロックを取得することによってキャッシュミスを誘発しているためであると考えられる．

図 15 に YCSB ワークロードにおいてキーの偏りがある状態でスレッド数を変化させた時のスループットを示す．図 15(a) からキーに偏りがある場合の YCSB-A ワークロードでは OML や OSL のような楽観的ロックを用いる場合でもマルチスレッド環境において性能向上しないことが分かる．これはキーの偏りによって書込みと読取りが競合し，読取りの再試行回数が増加することが原因であると考えられる．また，Bz 木や Bw 木のようなロックフリー索引はキーに偏りがある状況では楽観的ロックを用いる手法を上回る性能をもつことが分かる．

図 16 に YCSB ワークロードにおいてキーの偏りがない状態でキーサイズを変化させた時のスループットをそれぞれ示す．各結果からキーサイズが大きくなるにつれて性能が悪化することが分かる．これはキーサイズが大きくなるにつれて 1 ノードに格納できるレコード数が減り，ノード分割が増え木の構造が高くなったためである．木が高くなることで索引層で経由しなければならないノードの数が増えるため，メモリアクセスおよびそれに伴うキャッシュミスも増加し，全体として性能が減少する．また，特に図 16(a) の YCSB-A ワークロードにおいてキーサイズが大きくなるにつれて OML の性能が大きく下がっていることが分かる．OML は楽観的ロックを使用しており探索時にロックを取得しないため，場合によっては根ノードから再試行する．キーサイズが大きくなると頻繁に構造変更が発生し，根ノードからの再試行回数が増えたことが原因と考えられる．

4.3　Update-Scan Mixed

図 17 に Update-Scan Mixed ワークロードにおいてスレッド数を変化させた時のスループットを示す．図 17 から PML を除く全索引がスレッド数の増加に伴い性能が向上していることが分かる．B^+ 木において scan 操作は開始点のみ中間ノードを辿って探索し，その後は兄弟リンクを辿るため探索性能に影響を受けにくいことが理由として考えられる．PSL や OSL のような単層ロックを用いる手法は書込みによって構造変更が必要になったときのみ構造を変更するため，書込みが update 操作のみの本ワークロードでは構造変更が発生せず高い性能を発揮していると考えられる．一方，PML や OML のような多層ロックを用いる手法は構造変更が必要になりそうな場合は構造を変更するため，一括挿入によって索引を構築した後に構造変更が発生しロックの競合が起きるため，性能が低下しやすいと考えられる．

4.4　Construction および Destruction

図 18(a)(b) に Construction ワークロードにおいて 112 スレッドで構築する索引サイズを変化させた時のスループットを示す．図 18(a)(b) から stripe は range よりもスレッド間のアクセス競合が起きやすいため，全体的に性能が下がっていることが分かる．Construction ワークロードでは頻繁に構造変更が起こるため，構造変更時のロック粒度が低い OSL は OML よりも高い性能をもつ．この性能差は構築する索引サイズが小さい場合にロックの競合が起きやすいため，より顕著になる．一方，悲観的ロックを使用する PML および PSL は探索時でも共有ロックを取得することによってロック競合が起きやすいため，性能が向上しない．ただし，PSL は PML よりも構造変更時のロック占有期間が少ないため，やや性能が高い．

また，図 18(c)(d) に Destruction ワークロードにおいて 112 スレッドで初期索引サイズを変化させた時のスループットを示す．図 18(c)(d) から Construction ワークロード同様に stripe は range よりもスレッド間のアクセス競合が起きやすいため，全体的に性能が下がっていることが分かる．OML は range では OSL と

表 4　ワークロード

Workload	Read/Write	Initial Keys	Partition	Access Pattern	Payload Size
YCSB-A	0.5/0.5	1e8	none	random	8 byte
YCSB-B	0.95/0.05	1e8	none	random	8 byte
YCSB-C	1.0/0.0	1e8	none	random	8 byte
Update-Scan Mixed	0.5/0.5	1e8	none	random	8 byte
Construction	0.0/1.0	0	range/stripe	ascending	8 byte
Destruction	0.0/1.0	[1e4, 1e9]	range/stripe	ascending	8 byte

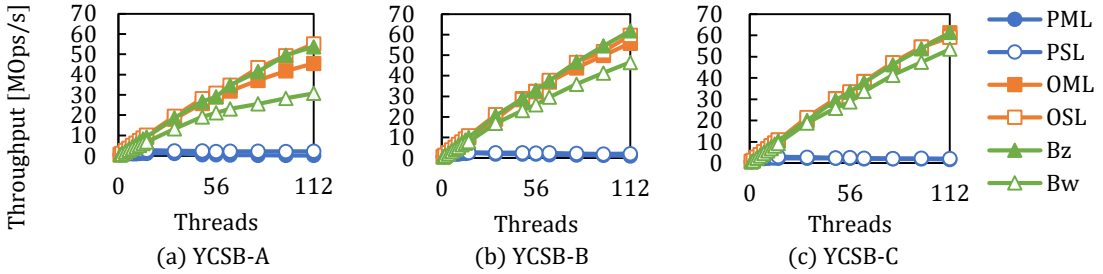


図 14　YCSB ワークロードにおけるスレッド数毎のスループット変化（skew=0.0，key size=8 byte）

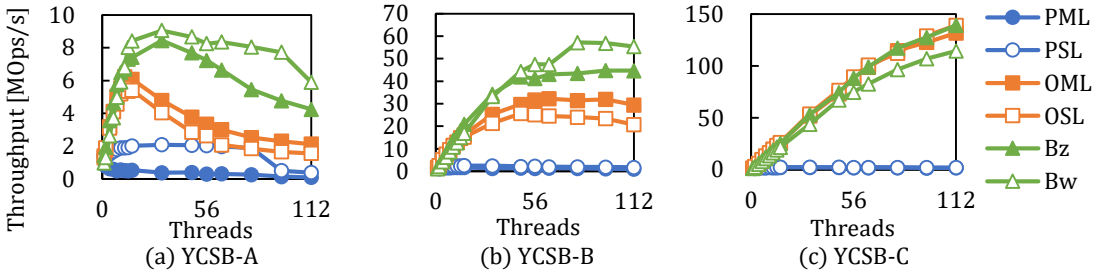


図 15　YCSB ワークロードにおけるスレッド数毎のスループット変化（skew=1.0，key size=8 byte）

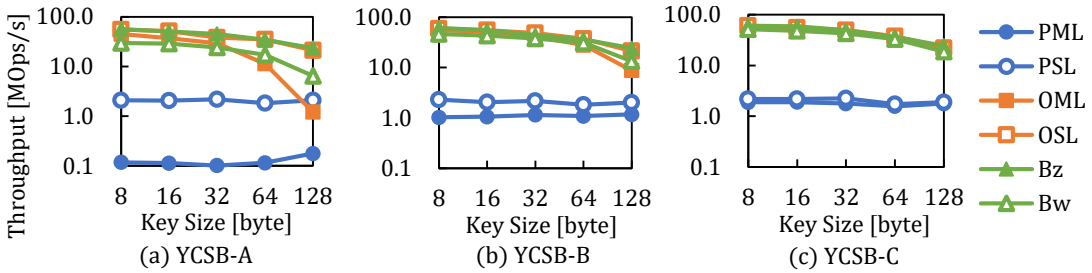


図 16　YCSB ワークロードにおけるキーサイズ毎のスループット変化（skew=0.0，thread=112）

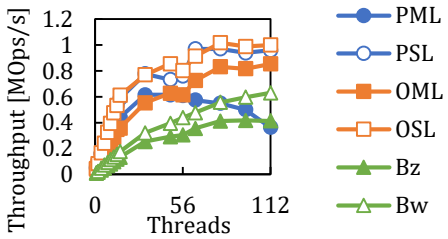


図 17　Update-Scan Mixed

同程度でロックフリー索引を上回る性能をもつ一方，stripe では特に初期索引サイズが 10^8 や 10^9 のような大きい場合にロックフリー索引を下回っていることが分かる．これは葉ノードにおけるレコードの削除をメタデータのフラグ更新によって管理している影響で隣接ノードの連続した merge 操作を誘発しやすいことが原因であると考えられる．

4.5　考　　察

実験の結果から悲観的ロックを用いる PML および PSL は少数のコア環境で使う場合に適していると考えられる．PML および PSL はメニーコア環境において性能向上しないことが実験の結果明らかになった．スレッド数の増加に伴う性能向上は 16 スレッド程度が限界であるため，メニーコア環境で悲観的ロックを用いるのは推奨されない．一方で悲観的ロックを用いる手法は実装の複雑さが少なく維持管理しやすいため少数のコア環境で安定した性能が求められる場合に適していると考えられる．

楽観的ロックを用いる OML および OSL はメニーコア環境でキー分布が一様分布に近い場合に適していると考えられる．OML および OSL はキーに偏りがない場合はメニーコア環境に

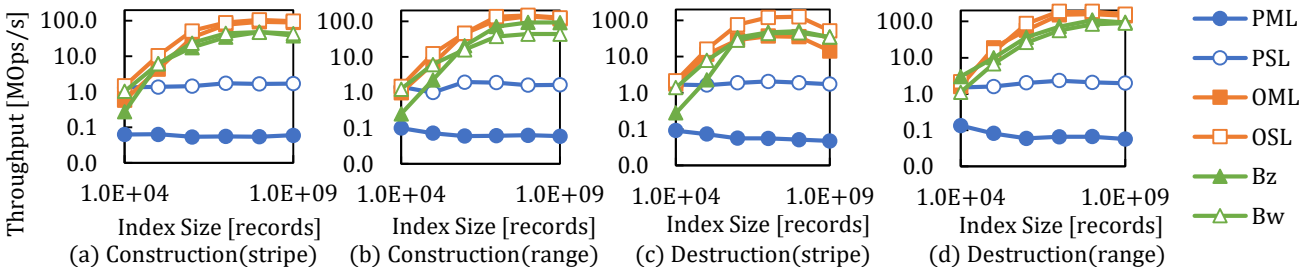


図 18　Construction および Destruction

おいて性能向上することが実験の結果明らかになった．一方でキーに偏りがある場合には性能低下しやすいなどワークロードによる影響を受けやすいことが分かった．このため楽観的ロックを用いる手法はキー分布が一様分布に近い場合や読取り割合が高い場合に適していると考えられる．

5　関 連 研 究

本章では，近年注目されているロックフリー索引として Bw 木および Bz 木について概説する．

Bw 木は各ノードに差分レコードをアトミックに追記していくことによって書込みおよび構造を変更するロックフリーな索引構造である．Bw 木は基本的には B^{link} 木と同様の構造をしているが，ノード間の物理ポインタは持たずマッピングテーブルによって仮想化している．Bw 木における書込みは差分レコードで表し，各ノードの先頭に追記する．この操作はマッピングテーブル上の CAS 命令で行われるため，ロックフリーに更新が可能である．差分レコードはある閾値を超えた場合に統合され，更新を反映したノードを生成する．

Bz 木は Multi-Word Compare-And-Swap（MwCAS）を不揮発性メモリ向けに拡張した Persistent Multi-Word Compare-And-Swap（PMwCAS）命令を用いることで不揮発性メモリ上でも動作するロックフリーな索引構造である．Bz 木は基本的には B^+ 木と同様の構造をしているが，葉ノードにおいて兄弟リンクを持たない．B^+ 木をロックによる排他制御なしで更新するためには，通常は複数ワードの更新をアトミックに行えなければならない．ここで Bz 木は MwCAS 命令を使用することによりロックフリーな排他制御を実現する．書込み時は MwCAS 命令を用いて葉ノードの未ソート領域に追記し，適宜ソート領域に移動する．読取り時には未ソート領域に最新レコードが格納されているため，まずは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．

6　お わ り に

本稿では B^+ 木における 4 種類の同時実行制御手法を紹介し，Bw 木や Bz 木といったロックフリー索引と共に性能評価を実施した．実験結果から悲観的ロックを使用する PML や PSL は読取り時に共有ロックを取得するためキャッシュミスを誘発しやすく，マルチスレッド環境において性能が向上しないことを確認した．一方，楽観的ロックを用いる OML や OSL はマルチスレッド環境においても性能が向上しやすく，単純なワークロードではロックフリー索引よりも高い性能をもつことを確認した．また，ロックフリー索引は単純なワークロードでは楽観的ロックを使用する手法にやや劣るが，キーに偏りがある場合に高い性能をもつことを確認した．今後の課題として，Adaptive Radix Tree（ART） [11] や Mass 木 [12] といったトライ木ベースの索引や ALEX [13] のような学習索引との性能比較が挙げられる．

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．また，本研究の実装にあたり助力いただいた名古屋大学石川研究室の平野匠真氏に感謝する．

文　　献

[1] 北川 博之, **データベースシステム**. 昭晃堂, 1996.
[2] P. L. Lehman and S. B. Yao, “Efficient locking for concurrent operations on B-Trees,” *ACM TODS*, vol. 6, no. 4, pp. 650–670, 1981.
[3] J. Rao and K. A. Ross, “Making B^+-Trees cache conscious in main memory,” in *Proc. SIGMOD*, pp. 475–486, 2000.
[4] S. K. Cha, S. Hwang, K. Kim, and K. Kwon, “Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems,” in *Proc. VLDB*, pp. 181–190, 2001.
[5] J. Levandoski, D. B. Lomet, and S. Sengupta, “The Bw-tree: A B-tree for new hardware platforms,” in *Proc. ICDE*, pp. 302–313, 2013.
[6] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson, “BzTree: A high-performance latch-free range index for non-volatile memory,” *PVLDB*, vol. 11, no. 5, pp. 553–565, 2018.
[7] Goetz Graefe, *On Transactional Concurrency Control*. Morgan & Claypool Publishers, 2019.
[8] V. Srinivasan and M. J. Carey, “Performance of B-Tree concurrency control algorithms,” in *Proc. SIGMOD*, pp. 416–425, 1991.
[9] V. Leis, F. Scheibner, A. Kemper, and T. Neumann, “The ART of practical synchronization,” in *Proc. DaMoN*, pp. 1–8, 2016.
[10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with YCSB,” in *Proc. SoCC*, pp. 143–154, ACM, 2010.
[11] V. Leis, A. Kemper, and T. Neumann, “The adaptive radix tree: ARTful indexing for main-memory databases,” in *Proc. ICDE*, pp. 38–49, 2013.
[12] R. T. M. Yandong Mao, Eddie Kohler, “Cache craftiness for fast multicore key-value storage,” in *EuroSys ’12*, pp. 183–196, 2012.
[13] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann, *et al.*, “Alex: an updatable adaptive learned index,” in *Proc. SIGMOD*, pp. 969–984, 2020.