DEIM2022 K34-2

ロックフリー索引構造Bw木の再現実装及び性能評価

牧田　直樹† 　杉浦　健人† 　石川　佳治† 　陸　 可鏡†

† 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: makita@db.is.i.nagoya-u.ac.jp, sugiura@i.nagoya-u.ac.jp, ishikawa@i.nagoya-u.ac.jp,
lu@db.is.i.nagoya-u.ac.jp

あらまし　近年，ムーアの法則の終焉によるメニーコア環境への転換に伴い，マルチスレッド向けの索引に対する需要が増加している．ATM や電子商取引に用いられる OLTP システムにおいてその性能は特に重要であり，索引構造の更新にロックフリー技術を用いるロックフリー索引は並列動作時の干渉の少なさとそれによる同時実行性の高さから注目されている．一方で，Bw 木や Bz 木などの既存のロックフリー索引に関して，統一的な環境・実装での比較実験は不足している．そこで，本研究ではロックフリー索引の性能検証を目的とし，基礎的なベンチマークの作成及び Bw 木の再現実装を行い，統一的な環境下での比較によってその性能特徴を検証する．

キーワード　ロックフリー索引，再現実装，性能評価

1 はじめに

ムーアの法則の終焉によりマシンの単一コアのクロック数を高めていく時代は終わり，より多くのスレッドを動かしてパフォーマンスを高めるメニーコアの時代となった．データベース技術も例に漏れず，日々やり取りされる膨大なデータを処理するためその対応に迫られている．例えば ATM や電子商取引に用いられる OLTP システムではリアルタイムに大量のデータベース処理を行う必要があり，その性能は特に重要である．マルチスレッドを主眼に置いたものとして，トランザクションエンジンでは Silo [1]，Hyper [2]，索引では ART [3]，Mass 木 [4]，P 木 [5] などが提案された．

データ構造を並列操作する場合の最も単純なアプローチとして，ロックの取得による排他処理が挙げられる．構造への破壊的な操作するスレッドが排他処理を行うことにより競合操作による不整合を防ぐ．しかし，ロックの占有により並列化によるスケーラビリティが小さくなるというデメリットがある．

そこで，索引構造の更新にロックフリー技術を用いるロックフリー索引は並列動作時の干渉の少なさとそれによる同時実行性の高さから注目されている．ロックフリー索引は CPU 命令であるコンペア・アンド・スワップ命令（Compare-And-Swap, CAS）を利用したロックを使わないアルゴリズムを用いる．CAS は不可分操作として，あるメモリ位置の内容と指定された値を比較し，等しければそのメモリ位置に別で指定された値を格納する．並列で CAS が動作した場合でも値の比較により，競合を検知することが可能となる．

一方で，Bw 木 [6] や Bz 木 [7] などの既存のロックフリー索引に関して，統一的な環境・実装での比較実験は不足している．具体的な実装方法についても解釈の余地が広く，適切な比較実験がなされているとは言い難い．

そこで，本研究ではロックフリー索引の性能検証を目的とし，基礎的なベンチマークの作成および Bw 木の再現実装を行う．また，統一的な環境下での比較によってその性能特徴を検証する．

2 関連研究

本章では木型の索引に関する先行研究を取り上げる．木型索引の基礎となる B^+ 木と並列環境下での一貫性制御について述べる．次に，Bw 木の比較対象となる B^+ 木のその他のバリエーションを紹介する．

2.1 B^+ 木

B^+ 木 [8] は挿入・削除・検索を効率に行うためのバランス木構造であり，様々な DBMS で索引構造として用いられている．

B^+ 木ではレコードは葉ノードに格納され，葉でないノードには分割キーが格納される．分割キーによって下位の木を分割するため，ある特定のキーが属する部分木を効率的に探索できる．特定のキーの読み取りや書き込みの操作は，この分割キーを用いた葉ノードの探索によって実現される．効率的な範囲検索のために葉ノードは兄弟ノードへのポインタを持つ．

特定のノードにレコードの偏りが発生した場合，B^+ 木は探索効率の悪化を抑制するためノードの分割やマージ等の構造変更操作を実行する．特定のノードにレコードの偏りが発生した場合，B^+ 木は探索効率の悪化を抑制するため構造変更操作をする．レコードがしきい値を超えた場合，B^+ 木は図 1 のように分割処理を実行する．ノードを 2 つに分割し，親ノードに新しくエントリを追加する．レコードがしきい値を下回った場合，B^+ 木は図 2 のようなリバランス処理を実行することがある．ノードを左側のものと統合し，親ノードからエントリを削除する．

並列環境下で B^+ 木を操作する場合，同時の書き込みが発生すると工夫がなければ構造の一貫性を損なう恐れがある．しかしながら，並列実行によるパフォーマンス向上のため，並列環境下における B^+ 木の一貫性制御は重要な課題である．

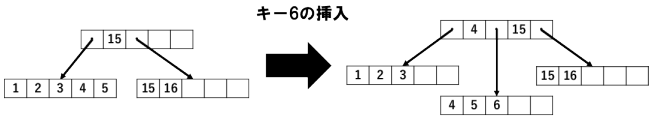


図 1　B^+ 木のノード分割処理

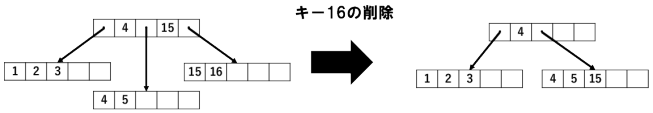


図 2　B^+ 木のノードマージ処理

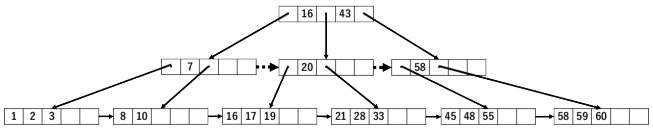


図 3　B^{link}　　木

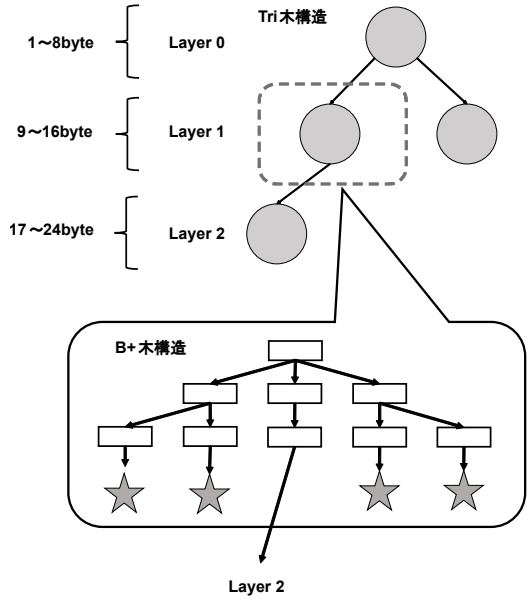


図 4　Mass　　木

2.2　B^{link}　　木

Lehman ら [9] は B^+ 木の一貫性制御の方法の一つとして B^{link} 木を提案した．

B^{link} 木は B^+ 木の各ノードに対して，葉ノードだけでなく内部ノードにも図 3 のように同じ高さの兄弟ノードへのリンクとハイキーを追加した構造を持つ．B^{link} 木ではノードへの書き込みの際にノード単位でロックを占有し，読み取りの際にロックは取らない．書き込みと読み取りが同時実行される場合，ノードの分割と読み取りが競合する場合がある．このとき，読み取り操作は読むべきノードの左側のノードを読む可能性がある．B^{link} 木の場合，探索キーとハイキーの比較によって作業スレッドはそうした場合でも右側ノードにリンクを用いて移動することができる．

2.3　Bz　　木

Bz 木は Arulraj ら [7] によって提案された不揮発性メモリを想定した B^+ 木ベースの索引構造である．不揮発性メモリ上の動作による更新の永続化や迅速なリカバリが利点として挙げられる一方で，揮発性メモリ上においても動作が可能である．

Bz 木では Persistence Multi-word CAS（PMwCAS）を使用してロックフリーな構造操作を実現する．CAS 命令とは，値の比較と置換を不可分に行うアトミック命令であり並列操作時の競合検知に用いられる．PMwCAS は，CAS（Compare and Swap）命令を単一ワードから複数ワードに拡張した命令である Multi-Word CAS [10] を不揮発性メモリ上で実行するものである．各操作において PMwCAS を用いることでアトミックな更新を実現する．

ノードの操作は，PMwCAS を用いたノードのメタデータの更新による書き込み領域の予約やレコードの可視状態の変更によって行われる．ノードへの書き込みは非ソート領域に行われ，非ソートレコードの数がしきい値を超えるとノードの再編成が行われる．

分割やマージなどの構造変更操作では，ノードを凍結状態にすることにより進行途中の更新操作の完了を防ぎ一貫性を保つ．凍結状態のノードを観測したスレッドはノードの構造変更操作を後追いする．最終的に，最も早く操作を終えたスレッドが PMwCAS によってノードをスワップし構造変更操作を完了させる．

Arulraj らは様々なワークロードにおいて Bw 木より高いパフォーマンスを発揮したことを報告している．

2.4　OpenBw 木

OpenBw 木は Wang ら [11] が提案した Bw 木を改良したオープンソースの Bw 木実装である．非ユニークキーのサポートと探索効率の改善のためのアルゴリズムを提案した．本研究ではこの OpenBw 木との比較も行う．

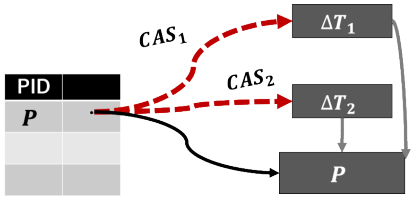
2.5　Mass　　木

Mass 木は Mao ら [4] が提案した，図 4 のようにトライ木の各ノードに楽観的同時実行制御を用いた B^+ 木を配置した構造を持つ．キーの接頭辞部分をトライ木で探索し，短く分割したキーをさらに B^+ 木で探索を行う．長いキーをそのまま扱うよりキャッシュ効率がよいという利点がある．ホームページアドレス等の prefix を共有しやすく長いキーに対して有用である．

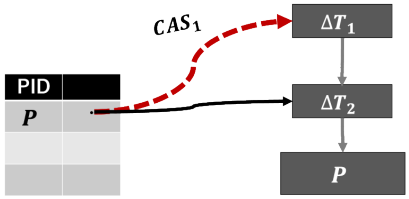
楽観的同時実行制御を用いた B^+ 木では，書き込みに対してはロックを取得するが，読み取りに対してはロックを取らず書き込みと読み取りの競合を許す．ノードにバージョンカウンタを持たせ，書き込みを行う際は同時に更新する．読み取り操作側でノードのバージョンカウンタを常に比較することによって中間状態の観測を防ぐ．

3　Bw　　木

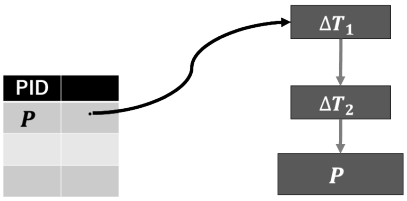
Bw 木は Levandoski ら [6] が提案したロックフリー索引である．基本的な構造は B^+ 木と共通しており，B^{link} 木のように各レベルのノードは右兄弟へのリンクポインタを保持する．独自



（a）ΔT_1とΔT_2の挿入の同時実行



（b）ΔT_2のCAS挿入の成功とΔT_1の挿入の再試行



（c）ΔT_1 の挿入が成功

図 5　Bw 木への書き込み

の点として，ノード間のリンクを仮想化するマッピングテーブルを持ち，差分ノードと CAS 命令を組み合わせてアトミックな更新を行う点がある．

マッピングテーブルはノード間のリンクを仮想化し，ストレージ内のノード領域の柔軟な変更を可能にする．一般的な B^+ 木と異なり，ノード間は物理的なポインタで結ばれていない．

Bw 木における書き込みは図 5 のようなデルタレコードの挿入操作に代替される．デルタレコードはノードの変更差分を表し，ノードの先頭に次々と挿入するため図 5(c) のようにデルタチェーンを生成する．デルタレコードの挿入操作はマッピングテーブルへの CAS 命令によって行われ，ロックを取得することなく競合する更新に対して正しく失敗する．失敗した挿入操作は挿入先を再設定しリトライされる．デルタチェーンはしきい値を超えると統合され，B^+ 木と同様のソートされたキーを持つベースノードを生成する．

ノードを探索する際は，デルタチェーンに対しては単方向リストと同様の探索を行い，ベースノードに対しては B^+ 木と同様の二分探索による効率的な読み取りを実行する．

4　Bw 木の再現実装

この章では Bw 木の各操作アルゴリズムおよびレコードの増減に伴なう構造変更操作について述べる．

```
1  Function Read(key)
2      node_stack ← SearchLeafNode(key)
3      leaf_node ← ValidateNode(key, node_stack)
4      existence, ptr ← キーの存在とペイロードへのポインタを
         取得
5      if existence = NotExist then
6          return KeyNotExist
7      else
8          return Success, payload(ptr)
9  Function SearchLeafNode(key)
10     begin
11         cur_head ← root_node
12         node_stack.push(cur_head)
13         while cur_head is not a leaf node do
14             if cur_headisDeltaNode then
15                 cur_head ← cur_head.next_node
16             else
17                 cur_head ← BinarySearch (cur_head,key)
18                 node_stack.push(cur_head)
19         return node_stack
```

図 6　Read アルゴリズム

4.1　葉ノードの操作

4.1.1　読み取り

読み取り操作は葉ノードから指定のキーを探索し，その値を返す．キーの検索のために，通常の B^+ 木と同じ方法でキーが属する葉ノードを探索する．

点読み取りである Read と範囲読み取りである Scan の実装が存在する．

a）　Read

Read 操作のアルゴリズムについて，図 6 に示す．

初めに *key* が属する葉ノードを取得する SearchLeafNode を呼び出し，取得したノードから目的のキーを探索する．SearchLeafNode は木の根からスタートし，*key* が属する葉ノードを探索する．葉ノードにたどり着くまでの道筋をスタックとして記憶し，スタックを返す．デルタチェーンは単方向連結リストの構造を持つのでネクストポインタを辿り，ベースノードはソートされた分割キーを持つので二分探索を行う．葉ノードまで到達するとスタックを返す．

Bw 木の読み取りと書き込みはロックフリーで行われ，B^{link} 木 [9] のようにノードにハイキーと右側ノードへのリンクを持つ．B^{link} 木と同様に，Split 操作と同時に実行される操作は辿るべきノードの左側のノードを読む可能性がある．こうしたことが起こりうるため，辿りついた対象ノードが検索キーの範囲内かどうかを検証する必要がある．ValidateNode は辿りついた葉ノードが検索キーの範囲内か検証するとともに必要に応じてサイドリンクを辿る．目的のキーが存在すればペイロードを返す．

b）　Scan

Scan 操作は，与えられた *key* を左端として右側方向に移動

```
1  Function Scan(key)
2      all_results ← Empty
3      node_stack ← SearchLeafNode(key)
4      target_node ← node_stack.top
5      while target_node is exist do
6          node_snapshot ← ConsolidateCopy (target_node)
7          all_results ←
             all_results + node_snapshot.records
8          target_node ← node_snapshot.right_node
9      return all_results
```

図 7　Scan アルゴリズム

する前向きイテレータを返す．図 7 は与えられた *key* より大きいものすべてを取得する際のイテレータの動きを再現したものである．

イテレータはサイドリンクを辿りながら右側へ進み，葉ノードを Consolidate したもののスナップショットを保持しながら進む．スナップショットの内容をすべて読み切った場合，葉ノードのサイドリンクを辿り右側兄弟のスナップショットを新たに取得する．

4.1.2　書 き 込 み

書き込み操作は葉ノードにキーと値のペアの情報を持つデルタレコードを挿入する．

キーが属する葉ノードを検索し，CAS によってΔ Insert の挿入を試行する．CAS が失敗した際は操作をやり直す．

無条件の書き込み操作である Write，キーの不在を条件とする Insert，キーの存在を前提とする Update，キーの削除を実行する Delete の実装がある．

基本となる Write 操作のアルゴリズムについて，図 8 に示す．初めに *key* が属する葉ノードを取得する SearchLeafNode を呼び出し，書き込み先のノードを特定する．ValidateNode により現在のノードが検索キーの範囲内かどうかを検証し，そうでなければサイドリンクを辿る．挿入するデルタレコードを作成し，挿入先の葉ノードの先頭要素と CAS を行う．書き込みの競合がなければ CAS は成功し Write の処理は終了する．CAS が失敗した場合，葉ノードの先頭要素を再び設定し CAS を再試行する．

Insert，Update，Delete の各操作については，キーの存在確認を行うフェイズが増える点以外では Write 操作と同様である．

4.2　構造変更操作

4.2.1　Consolidate

Consolidate はあるノードでデルタレコードの数がしきい値を超えたときにトリガされ，デルタレコードとベースノードを統合した新しいベースノードを作成する操作である．デルタレコードが意味する各種の操作をベースノードに再帰的に反映したベースノードが作られ，スレッドは CAS によってノードの置き換えを試行する．CAS が失敗した場合，スレッドは再試行を行わない．

各作業スレッドがノードを訪れた際にデルタチェーンの数をカウントし，しきい値を超えていればそのスレッドが Consolidate 操作を実行する．

```
1  Function Write(key, payload)
2      node_stack ← SearchLeafNode(key)
3      new_delta_record ←
         MakeInsertDeltaRecord(key, payload)
4      while true do
5          head ← ValidateNode(key, stack)
6          new_delta_record.next ← head
7          rc ← CompareAndSwap
             (head, stack.top, new_delta_record)
8          if rc = Success then
9              break
10     return Success
```

図 8　Write アルゴリズム

4.2.2　Split

Split は，Consolidate の結果により作成されたベースノードのレコード数がしきい値を超えた際にトリガされ，ノードの半分を右側の兄弟として分割する．ノード内のレコードが密になった際に検索効率の悪化を防ぐために実行される．

実際の操作は，図 9 のように**子ノードの分割**を行うフェーズ（ａ），（ｂ）と**親ノードの更新**を行うフェーズ（ｃ）に分かれる．

子ノードの分割

（1）　分割対象ノード P の右兄弟となるノード Q を作成．

（2）　Q をマッピングテーブルに登録する．

（3）　Δ Split を P に CAS を用いて挿入する．失敗時はノード P の Consolidate 操作をトリガする．

Q は P を適切に分割する右半分の要素を持つ．

Δ Split は Q へのポインタおよび P を分割する分割キー K_P を持ち，K_P より大きいレコードの探索を Q へ誘導する．

親ノードの更新

（1）　P の親ノードにΔ Insert を CAS を用いて挿入する．Δ Insert は K_P および Q のハイキーを持ち，Q に属するレコードへのキー検索を Q へ誘導する．

Δ Split，Δ Insert は Consolidate 時に統合される．Δ Split は右側兄弟のリンクの変更を反映し，Δ Insert は分割キーを更新する．

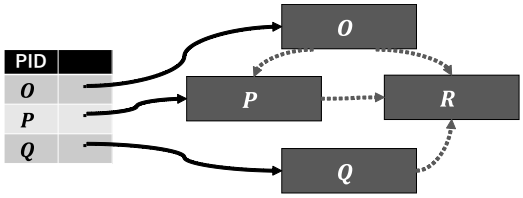
4.2.3　Merge

Merge は，対象ノードとその左兄弟のレコード内容を統合する操作である．Consolidate の結果により作成されたベースノードのレコード数がしきい値を下回った際にトリガされる．木のレコードが疎になった際の検索効率の低下を抑制するために実行される．

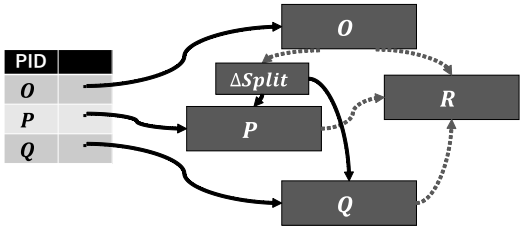
Merge は図 10 のように**削除対象のマーク**を行うフェーズ（ａ）と**子ノードのマージ**を行う（ｂ），**親ノードの更新**を実行する（ｃ）の 3 つの段階で実行される．

削除対象のマーク

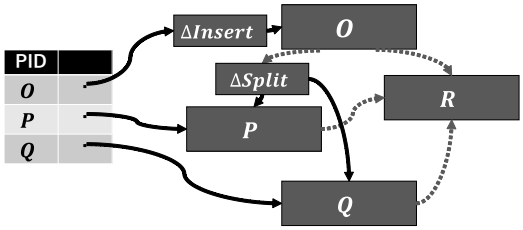
（1）　Merge 対象ノード R にΔ remove を CAS で挿入する．



（ａ）ノードQの作成

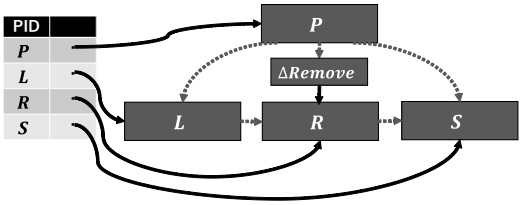


（ｂ）Δ$Split$の挿入

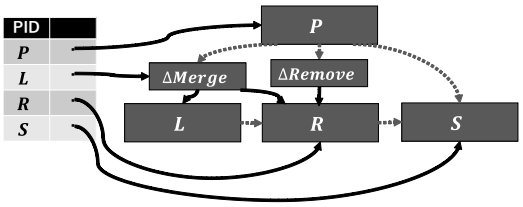


（ｃ）Δ$Insert$の挿入

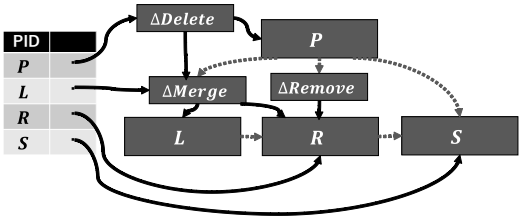
図 9　Bw 木の Split 操作



（ａ）Δ$Remove$の挿入



（ｂ）Δ$Merge$の挿入



（ｃ）Δ$Delete$の挿入

図 10　Bw 木の Merge 操作

失敗した場合，再試行は行わない．

Δ remove は R へのアクセスをブロックする．作業スレッドがΔ remove を見つけた場合，根から作業をやり直す．

子のマージ

（1） R の左兄弟 L にΔ Merge を CAS を用いて挿入する．失敗した場合，再試行する．

Δ Merge は R へのポインタおよび L のハイキーを持つ．Δ Merge は R に属するレコードへのキー検索を R へ誘導する．

親ノードの更新

（1） R の親ノード P にΔ Delete を CAS を用いて挿入する．失敗した場合，再試行する．

Δ Delete はΔ Merge へのポインタおよび L のローキー，R のハイキーを持つ．Δ Delete は L または R に属するレコードへのキー検索をΔ Merge へ誘導する．

Δ Delete，Δ Merge は Consolidate 時に統合される．Δ Delete は分割キーを更新し，Δ Merge はノードの統合を反映する．

4.2.4　構造変更操作の線形化と協調

Split と Merge はいくつかのステップに分けて実行されるため，これらの構造変更操作が進行中かつ未完了であるような中間状態が存在する．構造の一貫性を保つため，構造変更操作のスケジュールはある規則で線形化されている必要がある．例えば，図 11 のように複数の Merge 操作が同時に実行されている場合がある．図 11（ａ）のように S の Merge を行うスレッドが Δ $Merge_2$ の挿入先に別で進行している構造変更操作を示す Δ $Remove_1$ を見つけることがある．Δ $Remove_1$ が挿入されているノードとマージすることはできないため，S の Merge 作業の完了は左側のノード R の Merge の完了に依存する．Bw 木ではこのような競合を発見したとき，ノード S の Merge 作業を行っているスレッドは左側のノード R の構造変更操作を先に終わらせるよう協力する．この場合，図 11（ｂ）のように Δ $Delete_1$ の挿入を実行する．このように構造変更操作を行うスレッドが他の進行中の構造変更操作を観測した場合，協調操作を行うことがある．こうした構造変更操作の代行はスタックされ，再帰的に呼び出される．

5　性 能 評 価

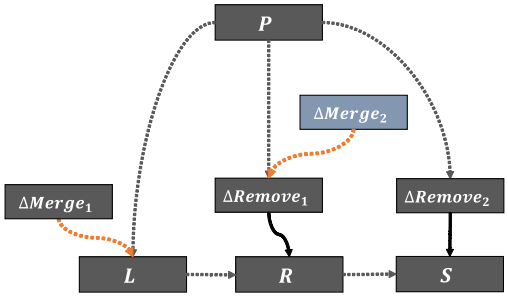
本章では作成した基礎ベンチマークの概要と評価項目について述べる．

5.1　ベンチマークの実装

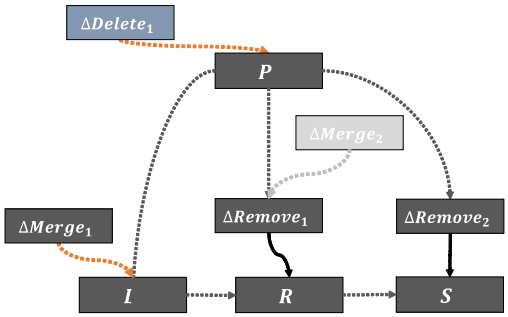
本ベンチマークはマルチスレッドをサポートし，ユーザの指定したワークロード下での索引のスループットおよびパーセンタイルの遅延を測定する．ベンチマークがサポートする可変なパラメータとして以下を設定している．

a）　木の初期サイズ

ベンチマーク実行前の初期化時に行う挿入操作の回数を設定する．read 操作のみのワークロードの実行や木の高さが各操作に与える影響を計測する際に有用である．



（a） $\Delta Merge_2$の挿入手順時に$\Delta Remove_1$を発見



（b） $\Delta Merge_2$の挿入をスタック，$\Delta Delete_1$を挿入

図 11　Merge 操作における協調動作

b）　キーサイズ

8, 16, 32, 64, 128 バイトのキーサイズをサポートする．

c）　キーの最大数及び偏り

式（1）の分布に従う Zipf の法則 [12] に基づき，パラメータを変更することでキーの偏りを調整する．式（1）において N はキーの最大数，k はキーの出現頻度の順位，s は偏りを調整するパラメータである．s を増加させるとキーの偏りは大きくなる．

$$f(k;s,N) = \frac{1/k^s}{\sum_{n=1}^{N} 1/n^s} \qquad (1)$$

d）　各操作の割合

索引への操作として read，scan，upsert（write），insert（write if not exist），update（write if exist），delete（write if exist）の 6 つを想定する．これらの操作の割合をワークロードとして指定し，ベンチマークとして実行する．

5.2　評 価 項 目

再現実装した Bw 木といくつかの索引構造について性能測定を行い比較する．Bw 木と比較する索引構造は以下の 4 つである．

- Bz 木 (append)
- Bz 木 (in-place)
- Mass 木
- OpenBw 木

Bz 木は再現実装したものを用いた．Bz 木は同一キーへの Write 操作時の処理方針の違いから 2 種類に分けている．Bz 木 (in-place) は Write 時に同一キーの存在を探索し，存在していればその領域のペイロードのみを置き換える．Bz 木 (append) は Write 時に同一キーの存在を確認せず非ソート領域への挿入を試みる．Mass 木および OpenBw 木は公開されている実装を用いた．

実験に用いるパラメータは 表 5.2 の通りである．実験に用いたマシンの構成を 表 2 に記載する．

以上の設定で索引操作のパーセンタイルの遅延およびスループットをベンチマークを用いて測定する．スループットは 5 回測定したうちの平均値を取る．遅延は，1 パーセンタイルから 99 パーセンタイルまでを 5 パーセンタイルの刻みで計測する．

表 1　測定に用いるパラメータ

ワークロード (Read:Write)	{(0:100) , (50:50) , (100:0) }
キーサイズ (byte)	{8 , 32 , 64 }
並列数	{1 , 8 , 16 , 32 , 48 , 64 , 80 , 96 , 112 }
初期レコード数	{1M}
操作発行回数	{10M}
キーの数	{1M}

表 2　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 20.04.2 LTS

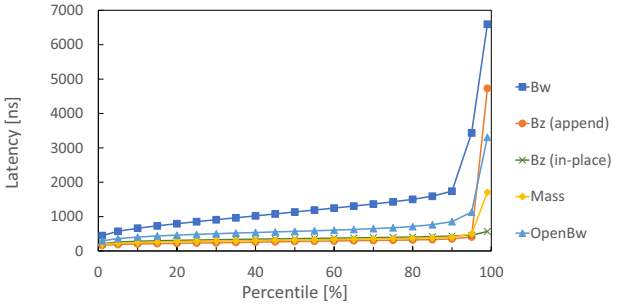


図 12　(R,W)=(0,100)，並列数 1 の条件下での各索引の遅延

5.3　実 験 結 果

5.3.1　遅　　延

各索引の遅延性能の測定を行った．特徴的な実験結果について取り上げる．

並列数 1，キーサイズ 8byte として Write-Only のワークロードの遅延を測定対象の索引全てに対して測定した結果，図 12 のようになった．Bw 木はデルタレコードでの更新や Consolidate 操作など，シングルスレッド動作時のオーバーヘッドの大きさから遅延が大きくなっていると考えられる．

並列数 112 とした同条件で測定した結果，図 13 が得られた．書き込みに対してロックを取得する Mass 木は並列数の増大に伴なって待ち状態のスレッドが多くなり，遅延も伴なって増大することが考えられる．また，ロックフリー索引は並列数の増加に対して遅延の増加が緩やかであることが確認できる．

並列数 1，キーサイズ 8byte として Read-Only のワークロードの遅延を測定対象の索引全てに対して測定した結果，図 14 のようになった．また，並列数 112 とした同条件で測定した結

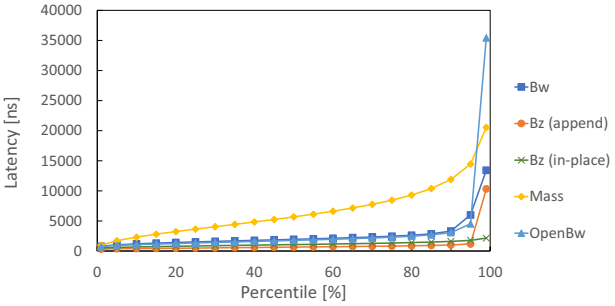


図 13　(R,W)=(0,100)，並列数 112 の条件下での各索引の遅延

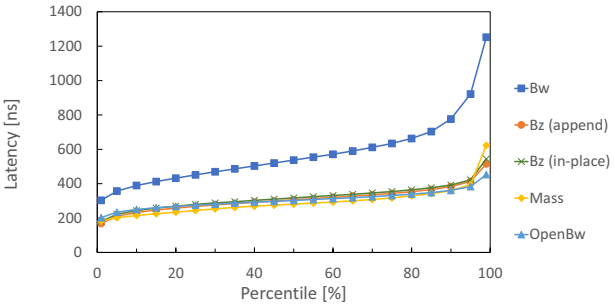


図 14　(R,W)=(100,0)，並列数 1 の条件下での各索引の遅延

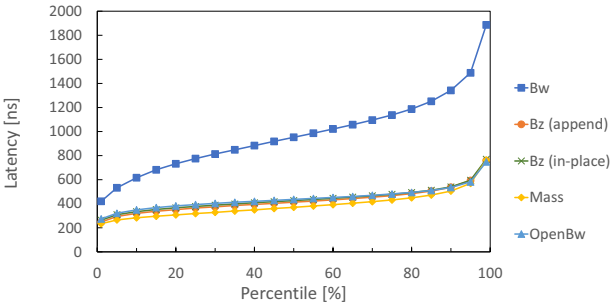


図 15　(R,W)=(100,0)，並列数 112 の条件下での各索引の遅延

果，図 15 が得られた．どちらも Bw 木の遅延が一つ抜けて大きくなっている．一方で OpenBw 木は他の索引と変わらない位置についているため，Read に対するチューニングの差が存在すると考えられる．また，Bw 木はデルタチェーンの種類によっての処理分岐が多く，かつ分岐先の偏りが小さい．そのため，CPU の分岐命令に対する分岐予測による効率化の恩恵が小さくなっていると考えられる．

5.3.2　スループット

キーサイズ 8byte として並列数を変化させ 3 つのワークロードのスループットを各索引について測定した結果，図 16，図 17，図 18 のようになった．

Write-Only のワークロードではロックを用いる Mass 木の性能が低く出ているが，並列数が大きくなるとロックフリー索引のスループットの増加率も小さくなっている．この原因として CAS 更新の失敗による索引操作の実質的な直列化や，構造変更操作の競合による遅延等が原因として考えられる．

Read-Only のワークロードに関して，どの索引もロックを取らないためスループットは直線的な増加が見られる．キーサイズ 8byte の条件下では単純な B^+ 木である Mass 木は他のロックフリー索引と比べてノード取得のオーバーヘッドが小さいため有利であると考えられる．

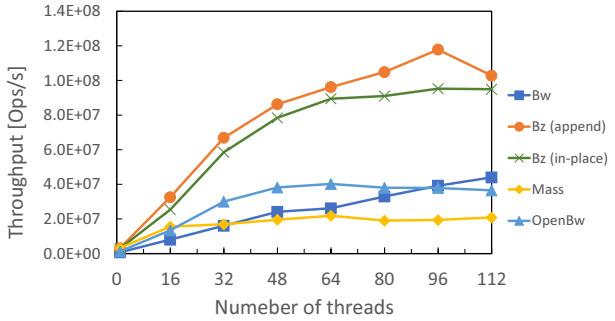


図 16　(R,W)=(0,100)，キーサイズ 8byte の条件下での各索引のスループット

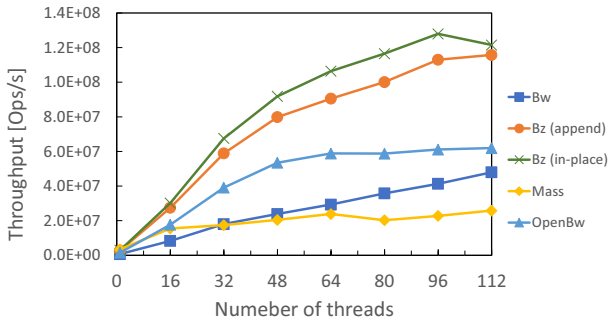


図 17　(R,W)=(50,50)，キーサイズ 8byte の条件下での各索引のスループット

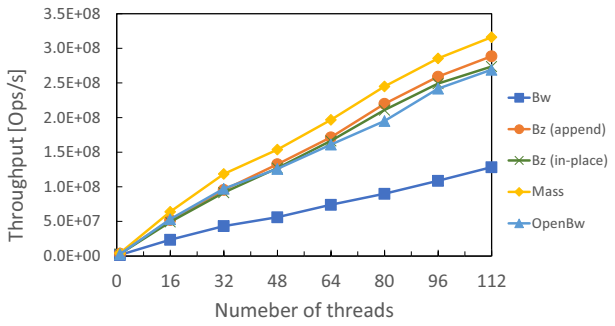


図 18　(R,W)=(100,0)，キーサイズ 8byte の条件下での各索引のスループット

5.4　考　　察

実験の結果，並列動作時のロックフリー索引の有効性を確認できた．また，実装した Bw 木は他の索引と比べて性能が出ていないことが明らかになった．この章では Bw 木の問題点と改善案について考察する．

a）　Consolidate の連続的な失敗

Consolidate 操作が連続的に失敗することにより操作のオーバーヘッドが大きくなっていることが考えられる．同一ノードに操作を行ったスレッドが全て Consolidate 操作を試行する場合，成功するのは多くとも一つのスレッドのみである．これらの操作が成功しなければ，後続するスレッドも Consolidate 操

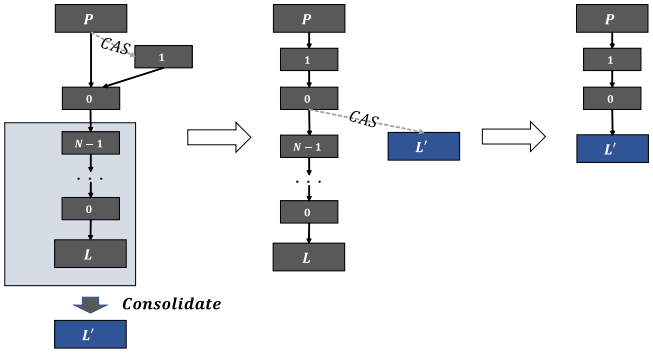


図 19　Consolidate 操作の改善案

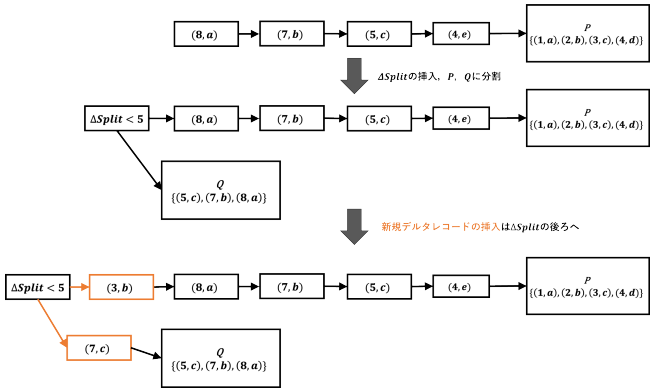


図 20　Split 操作の改善案

作がトリガされる．デルタチェーンが大きくなれば比例して操作の負荷は大きくなり，別の操作が挟まって失敗する確率はより大きくなる．さらに，デルタチェーンの増大により探索の効率も低下する．特に，特定のキーに書き込みが集中するようなワークロードで大きな遅延を引き起こすと考えられる．

この問題の改善策として，Consolidate ノードのスワップ位置をマッピングテーブルへのスワップではなくデルタチェーンのネクストポインタとのスワップにする方法が考えられる．図 19 のようにデルタチェーンに番号を振っておく．番号がしきい値 $N(modN)$ に到達したデルタノードを挿入したスレッドは，その子のデルタチェーンからベースノードまでの Consolidate を実行する．Consolidate 操作を行ったノード L' を番号 0 のノードのネクストポインタに対して CAS を行う．この CAS は確実に成功させることができるため，Consolidate を再試行するスレッドは 1 つのみになる．

b） 並行 Split 検知のためのノード走査

Bw 木は B^{link} 木と同様の仕組みで Split 操作と読み取り時が並行した際に起こる問題を回避しているが，これはキャッシュミスを増加させる要因になっていると考えられる．並行する Split 操作の検知のためにノードを探索するスレッドは $\Delta Split$ の存在を確認するためにデルタチェーンを走査し，到着した葉ノードが目的のノードであるかを常に確認する必要がある．つまり，ノードの書き込みや読み取りのためには対象葉ノードのデルタノードを全て検査する必要がある．並行した書き込みが増える分だけデルタノードのキャッシュヒット率は低下し遅延が大きくなると考えられる．

この問題の改善策として，図 20 のように $\Delta Split$ を常にデルタチェーンの先頭に持つ実装が考えられる．$\Delta Split$ が存在するノードへの更新は常に $\Delta Split$ のネクストポインタへの CAS で行う．$\Delta Split$ が存在するならば必ずデルタチェーンの先頭であるので，$\Delta Split$ の存在を確認するためにデルタチェーン全てを読む必要がなくなる．その結果，デルタチェーンを読む数が減るため，キャッシュミス率の低下を期待できる．

6　おわりに

メニーコア環境への転換に伴い，マルチスレッド向けの索引構造の需要は高まっている．ロックを必要とせず並列操作のスケーラビリティを失いにくいとされるロックフリー索引構造への注目が高まる一方で，Bw 木や Bz 木などの既存のロックフリー索引に関して統一的な環境・実装での比較実験は不足しているという課題が存在する．本研究ではその課題解決の一歩として基礎的なベンチマークの作成および Bw 木の再現実装を行った．

今後の課題として，他の従来索引との比較，Bw 木の操作アルゴリズムの改善案の実装とその性能測定が挙げられる．

謝　辞

本研究は JSPS 科研費（16H01722，20K19804, 21H03555）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．

文　献

[1] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, “Speedy transactions in multicore in-memory databases,” in *In Proc. SOSP*, pp. 18–32, 2013.

[2] A. Kemper and T. Neumann, “HyPer: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots,” in *In Proc. ICDE*, pp. 195–206, 2011.

[3] V. Leis, A. Kemper, and T. Neumann, “The adaptive radix tree: Artful indexing for main-memory databases,” in *In Proc. ICDE*, pp. 38–49, 2013.

[4] R. T. M. Yandong Mao, Eddie Kohler, “Cache craftiness for fast multicore key-value storage,” in *EuroSys ’12*, pp. 183—196, 2012.

[5] Y. Sun, G. E. Blelloch, W. S. Lim, and A. Pavlo, “On supporting efficient snapshot isolation for hybrid workloads with multi-versioned indexes,” *PVLDB*, vol. 13, no. 2, pp. 211–225, 2019.

[6] 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.

[7] 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.

[8] 北川 博之, “データベースシステム,” 情報系教科書シリーズ第 14 巻, pp. 115—119, 昭晃堂, 1996.

[9] s. B. Y. Philip L. Lehman, “Efficient locking for concurrent operations on B-trees,” *ACM*, vol. 6, no. 4, pp. 650–670, 1981.

[10] T. L. Harris, K. Fraser, and I. A. Pratt, “A practical multi-word compare-and-swap operation,” in *In Proc. DISC*, pp. 265–279, 2002.

[11] Z. Wang, A. Pavlo, H. Lim, V. Leis, H. Zhang, M. Kaminsky, and D. G. Andersen, “Building a Bw-tree takes more than just buzz words,” in *Proc. SIGMOD*, pp. 473–488, 2018.

[12] 梅沢 克之, Neil Ruberns, 松田 健, 三川 健太, 水野 信也, and 山本 健司, **情報検索 検索エンジンの実装と評価**. 森北出版株式会社, 2020.