ロックフリー索引のトライ木化による高速化に関する研究

井戸 佑[1,a)]　杉浦 健人[1,b)]　石川 佳治[1,c)]　陸 可鏡[1,d)]

概要：代表的な索引構造である B+木は様々なデータを格納可能な汎用性の高い索引である．一方で，扱うキーに制限を加え，索引構造を最適化させることで性能を向上させる研究も行われてきた．本研究では，著者らの研究室で開発しているロックフリー B^+ 木（B^c 木）に対し同様の拡張および性能改善を行う．具体的には，扱うキーをバイナリ比較可能なものに制限することでトライ木の構造を適用し，空間利用効率の向上およびそれに伴う検索性能の向上を図る．

1. はじめに

現在のコンピュータ技術は複数のコアを用いて処理を行うマルチスレッド処理が主流であり，近年では単体で 200 コアを超えるマシンも登場し始めている．データベース分野においても，メニーコアなどを前提としたインメモリデータベースの研究が進んでいる．データベースの構成要素の 1 つである索引技術も同様に，メニーコア化した CPU の活用が求められている．

代表的な索引構造である B^+ 木 [7] においても，メニーコアを有効利用するための同時実行制御手法が提案されている．伝統的なデータベースでは B^+ 木の制御にロックが使用されるが，マルチスレッド処理においてロックによる同時実行制御は多数の待ちスレッドが発生しうるため，スケーラビリティを悪化させる．そのため，B^+ 木をロックフリー化させた索引として Bw 木 [4] や Bz 木 [2] が提案された．更に，それらを凌ぐ書き込み性能を達成する索引として著者らの研究室でも同時実行 B^+ 木（concurrent-B^+ 木, B^c 木 [6]）を開発している．

B^c 木を含め，B^+ 木を基にした索引は可変長データを始めとする任意のデータ型を格納可能な汎用的な索引構造であるが，一方で特定のデータ型への特化による性能改善も可能である．例えば，キーをバイナリ比較可能なデータ型に限定することで性能を改善したものとして Mass 木 [5] がある．Mass 木では，キーがバイナリ比較可能であるため，キーを先頭から 8 byte 単位で分割し各部分キーで B^+ 木を構成する．これにより B^+ 木内で管理されるキーが固定長となるため，キャッシュ効率に優れたノード構成を適用できる．更に，部分キーによる階層構造はトライ木の構造と同様であり，先頭で一致したキーの共有によるキャッシュ効率の改善も可能である．

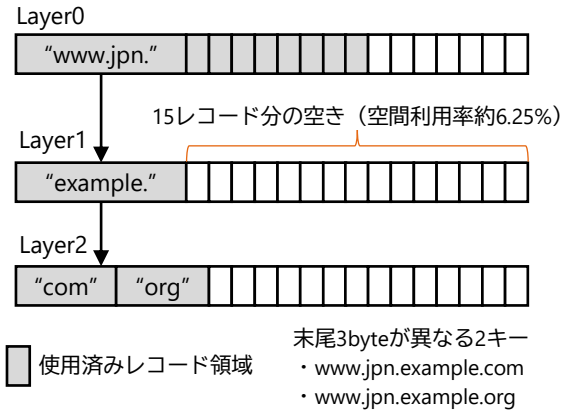

図 1　Mass 木の空間利用率

しかし，Mass 木はロックに基づく同時実行制御を用いている他，データセットによっては空間利用効率が極端に悪化しうるという問題がある．図 1 は末尾 3 byte が異なる 19 byte のキー 2 つと一意な 8 byte のキー 8 つを格納した Mass 木の索引構造である．先頭 8 byte が共通するため，layer 1 を作成する．同様に 9–16 byte 目が共通するため，layer 2 を作成する．Mass 木では本来，1 つのノードに最大 16 個のレコードを格納できる．しかし layer 1 では，1 つのレコードしか格納されていない状態で layer 2 を作成しており，layer 1 にのみ注目すると空間利用率は約 6.25 %しかない．Mass 木はトライ木の構造を持つため，接尾辞がわずかに異なるキーが多数存在すると空間効率が悪化しやすい．

そこで本研究では，B^c 木に対し扱うキーをバイナリ比較可能な型に限定することで性能を改善するとともに，Mass

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

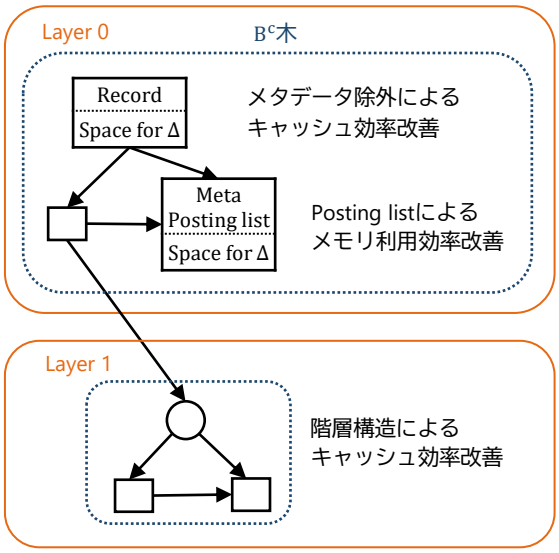

図 2 B^c-forest の概形

木のようなメモリ効率の悪化を回避した索引構造である B^c-forest を提案する．B^c-forest の概形を図 2 に示す．Mass 木と同様にキーを 8 byte の部分キーに区切り，各部分キーから B^c 木を階層的に作成する．各 B^c 木に対する操作の大半はロックに依らず動作するため，Mass 木に比べ書き込み性能を改善できる．更に，B^c 木内の中間ノードのノード構成を固定長レコードに最適化させ，キャッシュ効率を改善する．最後に，葉ノードでは posting list を利用し，共通の部分キーを持つレコードが一定サイズを超えるまで下層の生成を抑制する．つまり，下層で生成される B^c 木が極端に疎にならないよう制御し，Mass 木において発生した空間利用率の悪化を防ぐ．

本稿の構成は以下の通りである．2 章では，関連研究としてロックフリー索引やバイナリ比較可能なキーに対し最適化した索引について概説する．次に，3 章で B^c-forest の構造について説明し，4 章および 5 章で B^c-forest の操作について述べる．最後に，6 章で本稿のまとめと今後の方針を述べる．

2. 関連研究

関連の深い索引構造として，同時実行制御においてロックを取得しない B^c 木，および B^+ 木にトライ木の構造を組み合わせた Mass 木について紹介する．

2.1 B^c 木

B^c 木はマッピングテーブル・ノード内バッファという構造上の特徴を持ち，これらの構造および CAS 命令を利用することで大部分の操作をロックフリー化した索引である．B^c 木の概形を図 3 に示す．

2.1.1 データ構造の概観

B^+ 木と同様に，B^c 木は索引層およびデータ層によって

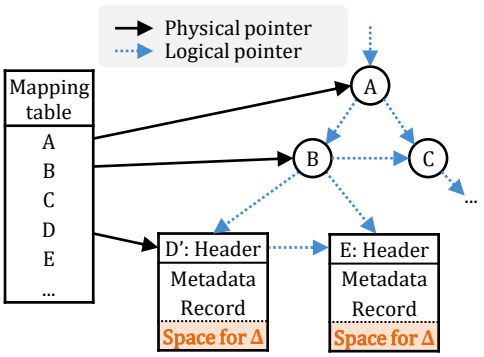

図 3 B^c 木の概形

構成される．索引層のノード（中間ノード）は分割キーと子ノードへのポインタの組を格納し，木の下方への検索を補助する．構造は B^{link} 木 [3] に則っており，各ノードが同じ階層の右兄弟への参照リンクを持つ．

ノード間の繋がりはマッピングテーブルにより仮想化する．各ノードは自身の子ノードや兄弟ノードへのポインタを直接持つ代わりにマッピングテーブル上の ID を持つ．各ノードへの参照はマッピングテーブルを用いた間接参照を採用し，マッピングテーブル内の物理ポインタを差し替えることでそのノードへの参照を一括で変更する．

各ノードの領域は不変領域と可変領域（ノード内バッファ）に分けられる．不変領域はノードヘッダおよびソート済みのレコードを格納する．ヘッダは不変領域の情報を管理し，構造変更時のみその値が変更される．可変領域はステータスワードの格納と差分レコードを挿入するための書き込みバッファの役割を果たす．ステータスワードは可変領域の情報を管理し，ノードの現在の状態や残容量などを管理する．

2.1.2 レコード操作の概観

ステータスワードを CAS 命令で更新することによって，ロックフリーな書き込みを実現している．書き込み操作は差分レコード領域の予約とレコード挿入および可視化の 2 ステップで行われる．図 4 に B^c 木における差分レコードの挿入を示す．

ノードフッタには可変領域の状態を表すステータスワードを用意し，この中のレコード数と使用済みブロックサイズを加算することで差分レコード用の領域を予約する．また，B^c 木ではノードの生成時に可変領域をゼロ埋めしている．B^c 木では，メタデータがゼロ埋めされている場合を処理途中として表すため，差分レコード用の領域を予約した時点ではレコードが可視化されていないことを認識できる．確保した領域へ差分レコードを書き込み，対応するレコードメタデータの値を更新することでレコードを可視化し，挿入処理を完了する．書き込み同士の競合はステータスワードで解決され，差分レコード用領域の予約，つまり差分レコードの書き込み順は CAS 命令の成功順によっ

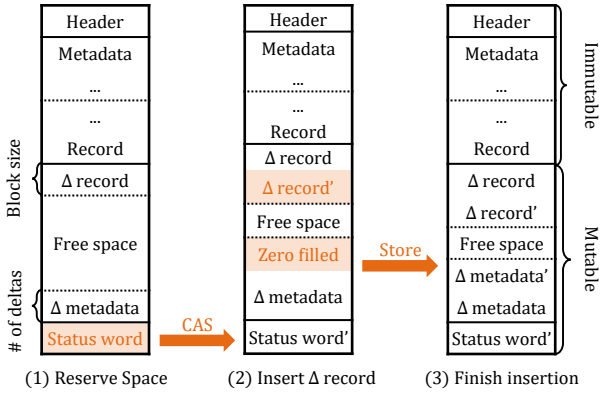

図 4　B^c 木における差分レコードの挿入

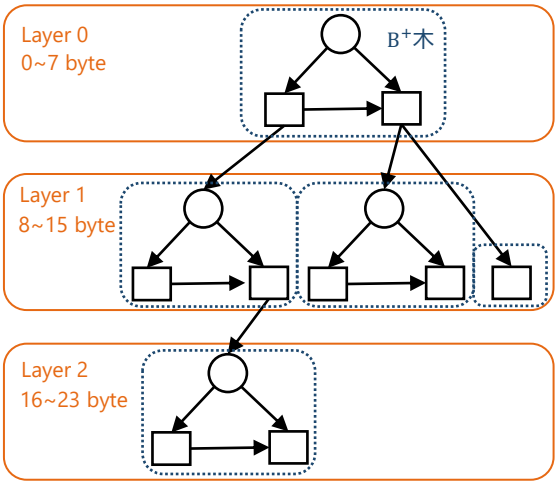

図 5　Mass 木の概形

て順序付けられる．

2.2　Mass 木

Mass 木は B^+ 木を基本単位とした階層構造やレコードメタデータの削除により，キャッシュ効率を改善した索引構造である．Mass 木の概形を図 5 に示す．

Mass 木は複数の B^+ 木と layer 構造から構成される．Layer 0 はキーの先頭 0〜7 byte で構成される B^+ 木である．先頭 8 byte で一意性が確保できる場合には，Layer 0 で完結する．先頭 8 byte で一意性が確保出来ない場合，Layer 1（キーの 8〜15 byte 目で構成される B^+ 木）を作成し，Layer 0 から Layer 1 への物理リンクを張る．同様にして，複数の B^+ 木や Layer を作成し，トライ木に似た構造を持つのが Mass 木の特徴である．Mass 木は，整数型や文字列型など先頭からのバイナリ比較で大小判定可能なキーのみを対象とすることで，上記に示す階層化（共通部分の集約）を可能にしている．

また，Mass 木は固定長キーおよび固定長ペイロードに特化したノードレイアウトを利用している．B^c 木のような可変長キーおよび可変長ペイロードに対応する索引構造

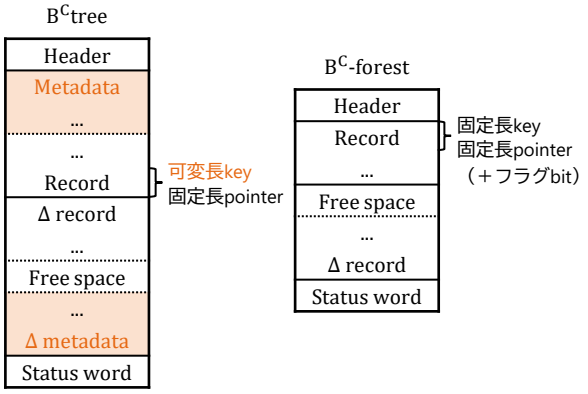

図 6　B^c-forest 中間ノードにおけるレコードメタデータの除外

では，各レコードに対応するレコードメタデータを利用することでノード内のレコードの配置等を管理している．一方，Mass 木ではキーを 8 byte 毎に分割するため各階層の B^+ 木で管理されるキーが 8 byte 固定長となる．ペイロードに関しても，可変長のペイロードなどは動的に確保した領域に保持し索引内にはそのポインタのみを格納することで，索引内では全てのペイロードを同じく 8 byte 固定長で扱う．つまりレコードの個数などから一意に参照先を決定でき，レコードメタデータの除外とそれによる CPU キャッシュ効率の改善が実現されている．

3.　B^c-forest の構造

B^c-forest は B^c 木をバイナリ比較可能なキーに最適化した索引構造である．Mass 木のように 8 byte 単位でキーを分割，階層分けし，各階層でのレコード管理には Bc 木を利用する．つまり，各 Bc 木の中間ノードでは固定長の部分キーおよび子ノードへのポインタのみを管理することとなり，レコードメタデータの除外によるキャッシュ効率の改善が可能となる．一方で，葉ノードでは posting list を用いて共通する部分キーを持つレコードを管理し，少数のレコードのみからなる下層の生成を抑制する．

3.1　中間ノードにおけるレコードメタデータの除外

B^c-forest では Mass 木同様，中間ノードにおいてレコードメタデータを除外できる．図 6 に，B^c 木および B^c-forest の中間ノードを示す．B^c-forest ではキーを 8 byte で分割しているため、中間ノード中では固定長キーとして扱える．また，ペイロードは子ノードへのポインタであるため固定長である．つまり中間ノード内のレコードはメタデータに依らずアクセス可能であり，キャッシュ効率の改善およびそれに伴う索引層の探索性能改善が可能である．

レコードメタデータの除外に伴い，各レコードの可視および削除済みフラグを子ノードポインタ用の領域で管理する．メタデータはレコードの配置管理以外に，差分レコードが挿入ないし削除済みであるかどうか判別する役割を

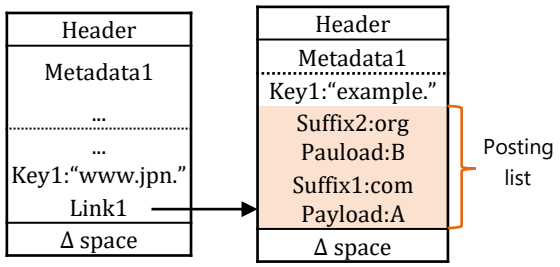

図 7 posting list の導入

担っている．そのため，メタデータを除外するには各フラグを何らかの方法で管理しなければならない．そこで，主要 OS の仮想メモリの管理において，ユーザ領域の上位 bit が未使用であることを利用する [1]．つまり，子ノードへのポインタをレコード中に格納するとき，最上位 bit を visibility フラグ，2 番目のビットを deleted フラグとして利用する．

3.2 葉ノードにおける posting list の導入

B^c-forest では空間利用率の改善として，posting list を導入する．図 7 は posting list を導入した際の索引構造を示したものである．

posting list では，1 つのキーに対して複数のペイロードを保持できる．layer 1 において posting list 作成することで，layer 2 の無駄な階層化と空間利用率の悪化を防ぐ．なお，posting list は後述する統合時に不変領域に生成され，可変領域には生成しない．

4. B^c-forest のノード操作

B^c-forest は読み取り操作（read，scan）および書き込み操作（upsert，insert，update，delete）をサポートする．本節では，読み取り操作（read）および書き込み操作（upsert）について説明する．

ノード操作は対象ノードの特定とノード内操作の 2 段階に分けられる．対象ノードの特定では，与えられた対象キーをもとに操作対象のノードを特定する．まず，対象キーの先頭 0〜7 byte をもとに属するの葉ノードを根ノードから二分探索により特定する（Layer 0）．次に，葉ノード内の不変領域を二分探索し，接頭辞 8 byte が共通するキーについて下層へのポインタの有無を確認する．ポインタが無い場合，本葉ノードを挿入先の葉ノードとして特定する．ポインタがある場合，下層の根ノードへ移動し，対象キーの 8〜15 byte 目をもとに属する葉ノードを二分探索により特定する（Layer 1）．同様に下層ポインタの有無を確認し，本操作を下層ポインタがなくなるまで繰り返し，葉ノードを特定する．特定した葉ノード不変領域内に対象キーが一致するレコードがある場合には，その位置を保持し，後述するノード内操作時に利用する．

4.1 読み取り（read）

読み取り（read）は与えられた対象キーのペイロードを返す操作である．操作対象の葉ノード特定後，葉ノード内の可変領域の線形探索と不変領域の 2 ステップで読み取り操作を行う．最新の値は差分レコード領域に書き込まれるため，まず可変領域を線形探索し，一致するキーがあるか確認する．このとき，差分レコードの数はステータスワードから読み取り，差分レコードが可視化されていなければスキップする．

差分レコード中に対象のレコードが存在しなければ，対象ノードの特定時に保持した位置をもとに，不変領域の読み取りを行う．接頭辞 8 byte が共通するキーについて posting list が存在する場合には，posting list 内を線形探索し，接尾辞が一致するキーがあるか確認する．先頭ノードが構造変更操作の途中である場合には，差分レコードを読み終わった時点で物理ポインタをたどり古いノードへ移動し，同様の 2 ステップを古いノード上で行う．

読み取り処理は wait-free に動作する．上述したとおり読み取り命令は一切ノードの状態を変更せず，読み取った状態に応じて適切な手続きを選択する．そのため，読み取り命令においてリトライなどは発生せず，有限時間内で必ず処理が終了する．

4.2 書き込み（upsert）

書き込み（upsert）は与えられた対象キーおよびペイロードを挿入する操作である．操作対象の葉ノード特定後，posting list チェックとレコード挿入の 2 ステップで書き込み操作を行う．B^c-forest における書き込み操作では，差分レコードのメタデータに posting list のサイズを持たせる．これは後述する下層生成操作と書き込み操作の衝突時に，書き込み先ノードを特定するためである．

Posting list チェックでは，キーの一意性を確認する．読み取り操作と同様に可変領域，不変領域の順に探索し，挿入するレコードを含めた posting list のサイズを計算する．以上の操作にて，取得した posting list のサイズをメタデータに入れ，図 4 に示す B^c 木と同様の操作で挿入する．

以上の書き込み操作によって差分レコードを挿入していくが，挿入後の差分レコードの総数またはノード容量のしきい値を越える場合，構造変更操作が行われる．しきい値の確認はステータスワードの更新時に行われ，しきい値を越えた場合はレコードの書き込み後にいずれの構造変更操作を行うか判定する．書き込み処理の大部分はロックフリーに動作するが，後述する構造変更操作の一部手続きとのみロックに基づく制御を利用する．

5. B^c-forest の構造変更操作

B^c-forest は構造変更操作として差分レコードのノードへの統合（consolidate）とノードの分割（split），併合（merge）

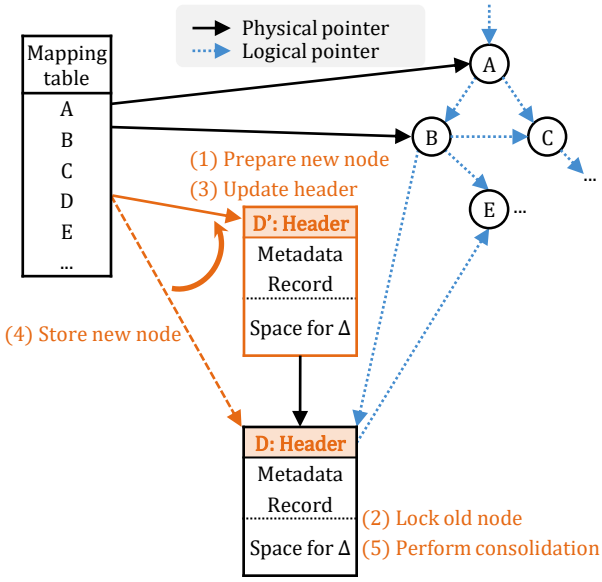

図 8 B^c-forest における統合操作

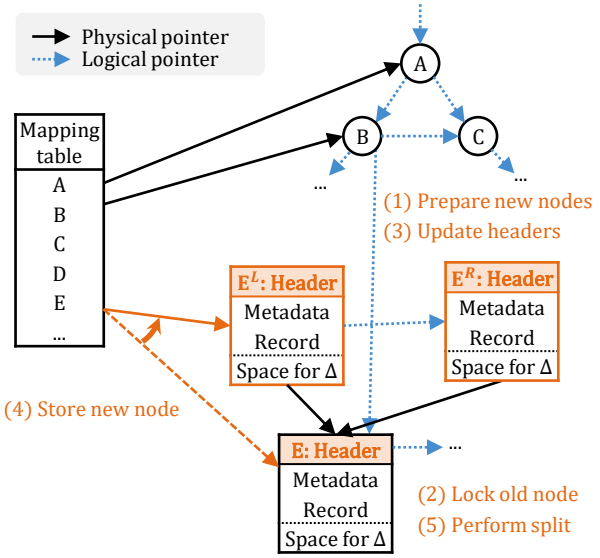

図 9 B^c-forest における分割操作

および下層生成，下層削除操作を持つ．本節では統合操作，分割操作および下層生成操作について述べる．

構造変更操作を始める前に，新たなノードをマッピングテーブルに挿入する．構造変更中に行われる書き込みについては，用意した新たなノード領域に書き込むことで，他のスレッドの待ち時間を削減する．

5.1 統合

統合操作は差分レコードの数またはノードの容量がそれぞれのしきい値を越えた場合，差分レコードと不変レコードをソートして不変領域に反映する操作である．本操作時に接頭辞 8 byte が共通するキーを集約し，不変領域に posting list（3.2 節）を作成する．統合操作を行うことで，可変領域を確保し新たな差分レコードの挿入を受け付ける．

図 8 に B^c-forest における統合操作を示す．統合が必要になった場合，新たなノード領域を用意する．そして，古いノードのステータスワードを更新後，ノードヘッダの情報から統合後に必要な不変領域を計算する．なお，このステータスワードの更新により古いノードは全体が不変となり，新たな差分レコードは挿入できなくなる．次に，新たなノードにおいてステータスワードを含むヘッダ情報のみを更新した後，新規ノードとしてマッピングテーブル上の参照を更新する．その後，古いノード上の差分レコードを不変レコードへ反映させつつ，新たなノードへレコードをコピーする．最後に，統合後の状態でノードヘッダを更新し，新しいノードにおける不変領域を可視化する．この際に，後述する分割操作で用いられる分割キーを，統合後のノードの不変領域から計算する．

統合操作を行っている際に，発生した書き込み操作は新規ノードの可変領域で受け付ける．これにより統合操作の間も，読み取り操作は新規レコードの可変領域，統合前のノードの可変領域および不変領域の順でレコードを確認することで読み取り操作を実行できる．

5.2 分割

分割操作はノードの差分レコードの数またはノード容量がそれぞれしきい値を越えており，統合後も十分な可変領域を確保できない場合に，そのノードのレコードを新規の 2 つのノードに分散させる操作である．分割操作により，容量が一杯になったノードを 2 つに分け，新たな差分レコードの挿入を受け付ける．

図 9 に B^c-forest における分割操作を示す．構造変更操作が必要になった場合，新たなノード領域を 2 つ用意する．ステータスワードの更新により対象ノード全体を不変状態にした後，レコードコピーを始める前に用意したノードをマッピングテーブルに挿入する．これを実現するために，各ノードは自身の分割キーの位置を統合および分割操作のたびに記録する．分割したノードの挿入後，統合操作と同様の手続きでレコードをコピーする．分割時には，古いノードの分割キー未満の全てのレコードを左ノードにコピーする．残りのレコード，すなわち分割キー以上の全てのレコードは右ノードにコピーする．

レコードのコピー後は，親ノードに右分割ノードへのリンクを挿入することで分割操作が完了する．このとき，親ノードへはリンクの情報のみを挿入し，反映は親ノードの統合操作時に行う．例えば新規ノード E を挿入した際，差分レコード領域に子ノードへのリンク情報のみを追記し統合操作が行われるまでそのリンクの反映を遅延する．つまり，索引の探索中において中間ノードでは差分レコードを確認せず，不変領域のレコードのみを検索する．

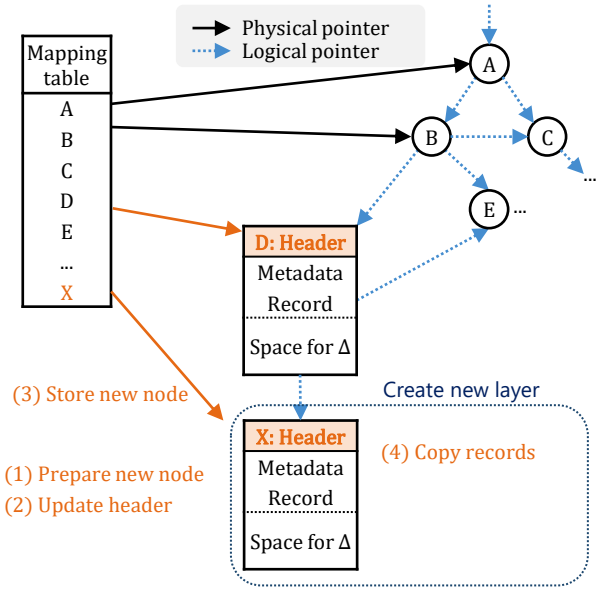

図 10 B^c-forest における下層生成操作

5.3 下層生成

下層生成操作はノードの差分レコードの数またはノード容量がそれぞれしきい値を越えており，統合後も十分な可変領域を確保できない場合に，posting list 内のレコードを下層に移行する操作である．下層生成操作により，ノード内のレコードの一部を下層に移行させるとともに，新たな差分レコードの挿入を受け付ける．

図 10 に B^c-forest における下層生成操作を示す．下層生成操作が必要になった場合，新たなノード領域を用意する．レコードコピーを始める前に用意したノードをマッピングテーブルに挿入する．下層生成操作を引き起こすきっかけとなった，posting list のレコードを新たなノードにコピーする．この時，接尾辞の先頭 8 byte をもとに必要であれば posting list の作成を行う．元ノードの posting list があったペイロード領域に新ノードへのポインタを格納する．

下層生成操作を行っている際に書き込み操作が発生した場合，下層生成の有無を確認する．書き込み操作の一意性チェック時に，posting list のサイズから下層が生成されるか確認する．生成されない場合は，元ノードに差分レコードを挿入する．生成される場合は，下層生成操作が終わるまで書き込みはブロックされる．

6. おわりに

本稿では B^c 木に Mass 木と同様のトライ木構造を適応させた B^c-forest について提案し，その構造および操作を紹介した．今後は提案した索引構造を実装するとともに，Mass 木や近年提案されている Bw 木や Bz 木といったロックフリー索引との性能を比較検証する．

謝辞 本研究は JSPS 科研費 JP20K19804，JP21H03555，JP22H03594，JP22H03903 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] : The Linux Kernel 29.3. Memory Management, https://www.kernel.org/doc/html/latest/arch/x86/x86_64/mm.html#memory-management. Accessed: August 5, 2024.
[2] Arulraj, J., Levandoski, J., Minhas, U. F. and Larson, P.: BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory, *PVLDB*, Vol. 11, No. 5, pp. 553–565 (2018).
[3] 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).
[4] Levandoski, J., Lomet, D. and Sengupta, S.: The Bw-Tree: A B-tree for New Hardware Platforms, *Proc. ICDE*, pp. 302–313 (2013).
[5] Mao, Y., Kohler, E. and Morris, R. T.: Cache Craftiness for Fast Multicore Key-Value Storage, *Proc. EuroSys*, pp. 183–196 (2012).
[6] 匠真. 平野，健人. 杉浦，佳治. 石川，可鏡. 陸：同時実行 B+木におけるロックフリー手続きの改善と実装，第 16 回データ工学と情報マネジメントに関するフォーラム (DEIM Forum 2024)，T2-A-4-02 (2024).
[7] 北川 博之：データベースシステム，昭晃堂 (1996).