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

ムーアの法則の終焉により，CPU のコア単体性能は限界に達しつつある．一方で，IT 技術の発展に伴い管理すべきデータは爆発的に増えつつある．この状況に対処するため，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る．データベース分野においても例外ではなく，近年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・大容量メモリに適合させる必要がある．

代表的な索引構造である B^+ 木 [4] では，ロックを用いた同時実行制御が行われている．しかし，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するため，スケーラビリティが悪化する．そこで，B^+ 木をロックフリー化させた索引として Bw 木 [2] や Bz 木 [1]，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）が提案されている．

また，インターネットの普及に伴い URL の管理や EC サイトにおける文字列検索など，特定のキーに対し効率的に処理する索引構造が求められている．Mass 木 [3] は，文字列型や整数型などのバイナリ比較可能なキーに特化した索引構造の 1 つである．B^+ 木を階層的に作成することにより，キャッシュ効率を改善している．

本研究の B^c-forest 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，Mass 木のと同様の拡張および性能改善を行う．Mass 木の観点では B^+ 木から B^c 木になることにより，ロックフリーによる書き込み性能の改善を図る．また，Posting list の利用により，Mass 木における空間利用率の改善を図る．B^c 木の観点では，整数型や文字列型などのバイナリ比較可能なキーに対し，キャッシュ効率の改善を図る．

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 1 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（logical

[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

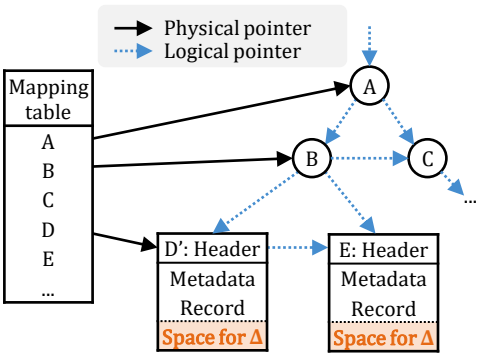

図 1 B^c 木の概形

page ID, LPID）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 2 に B^c 木における差分レコードの挿入を示す．

ノードフッタにはミュータブル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ミュータブル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 3 に示す．

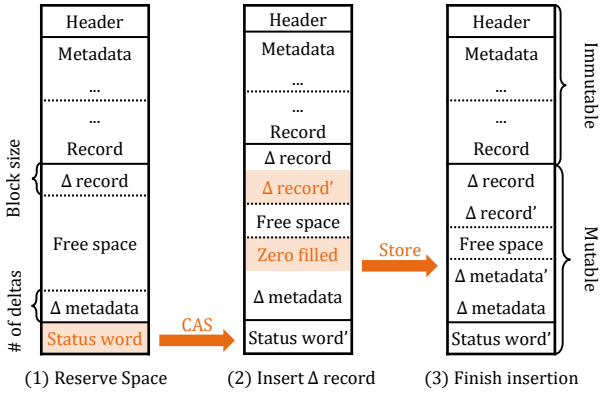

図 2 B^c 木における差分レコードの挿入

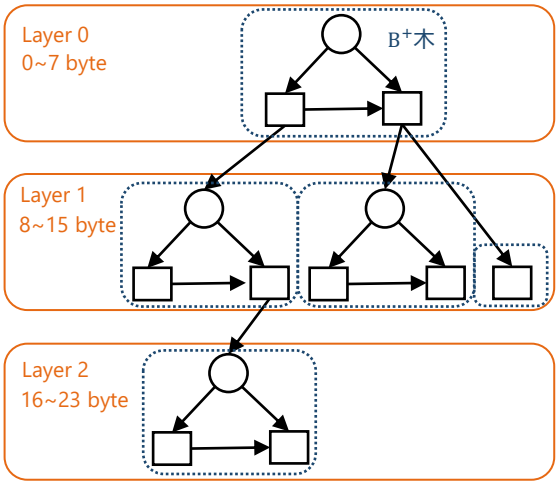

図 3 Mass 木の概形

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，分割が可能なキーに限定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

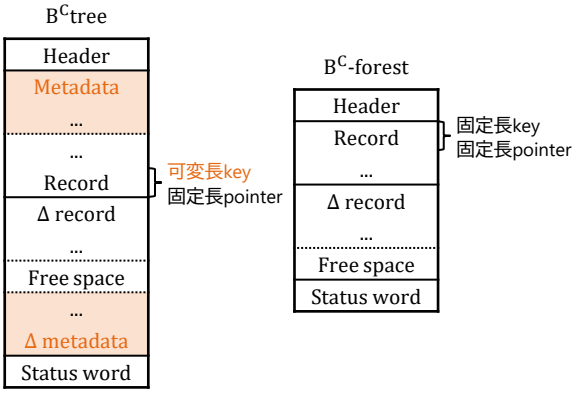

図 4 B^c-forest 中間ノードにおけるレコードメタデータの除外

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することが出来る．図 4 に，B^c 木および B^c-forest の中間ノードを示す．

B^c-forest では中間ノードはキーを 8 byte で分割しているため，固定長キーとして扱うことが出来る．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．この特性を利用し，B^c-forest 内の B^c 木における中間ノード内のレコードメタデータの除外を行う．これにより，索引層における探索性能の改善及びキャッシュ効率を改善を図る．

3.2 葉ノードにおける posting list の導入

Mass 木においては，空間利用効率が問題となる．図 5 は末尾 3 byte が異なる 19 byte の 2 キーを格納した 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 8〜16 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することが出来る．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 を作成している．layer 1 にのみ注目すると，空間利用率は約 6.25 %しかない．

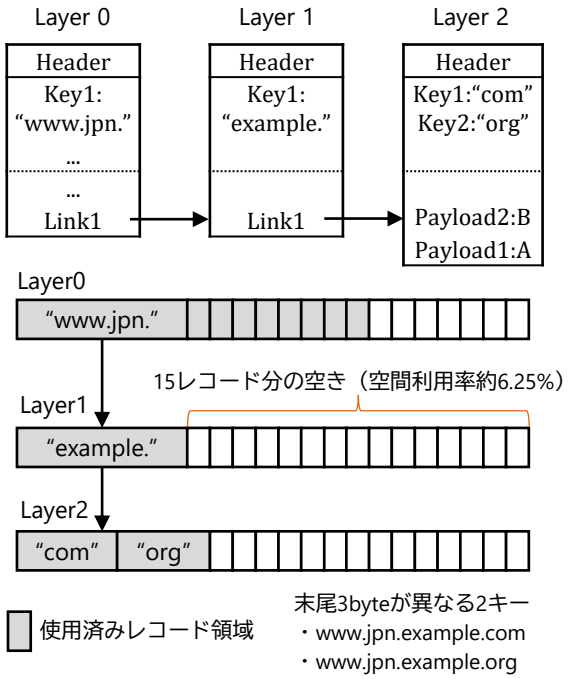

図 5 Mass 木の空間利用率

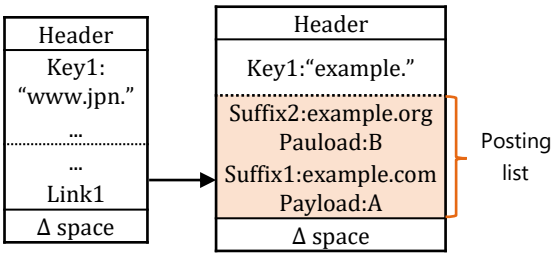

図 6 posting list の導入

B^c-forest では空間利用率の改善として，posting list を導入する．posting list では，1 つのキーに対して複数のペイロードを対応付けることが出来る．図 6 は posting list を導入した際の索引構造を示したものである．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

本節では，B^c-forest のノード操作について述べる．B^c-forest は読み取り操作および書き込み操作をサポートする．

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，葉ノード内のイミュータブル領域を二分探索し，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．

4.1 読み取り

読み取り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，葉ノード内の差分レコードの線形探索とイミュータブルレコードの二分探索の 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，まずミュータブル領域を線形探索し，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，ノードヘッダからイミュータブルレコードの数を読み取り，二分探索によって対象キーの有無を確認する．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．先頭ノードが統合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

書き込み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作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まずミュータブル領域を線形探索し，一致するキーがあるか確認する．存在する場合，そのメタデータが持つ posting list のサイズに書き込み対象のレコードのサイズを加算し，最新のサイズを計算する．存在しない場合，イミュータブル領域を二分探索し，posting list のサイズを取得する．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 2 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，差分レコードの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統合操作，構造変更操作いずれかの操作を行うか判定する．書き込み処理はロックフリーに動作するが，後述する構造変更操作との競合によって待ち時間が発生しうる．

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とノードの分割，下層作成操作を持つ．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，差分レコードとイミュータブルレコードをソートしてイミュータブル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，イミュータブル領域に posting list（3.2 節）を作成する．統合操作を行うことで，ミュータブル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 7 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イミュータブル領域を計算する．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イミュータブルとなり，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イミュータブル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イミュータブル領域を可視化する．この際に，後述する分割操作で用いられる分割キーを，統合後のノードのイミュータブル領域から計算する．

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ミュータブル領域で受け付ける．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ミュータブル領域，統合前のノードのミュータブル領域およびイミュータブル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

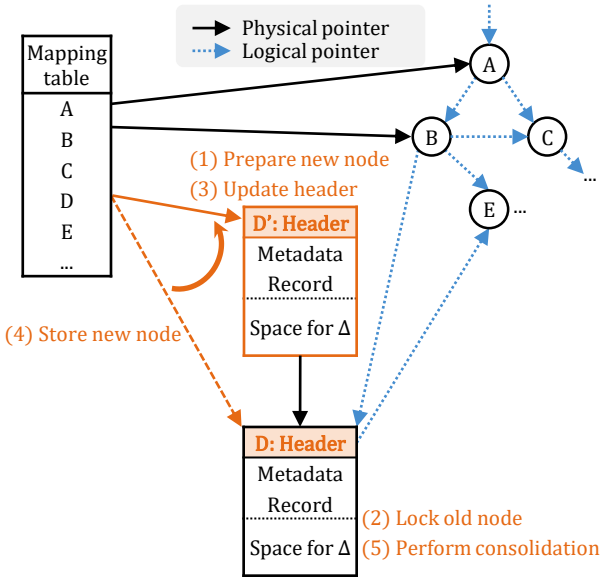

図 7 B^c-forest における統合操作

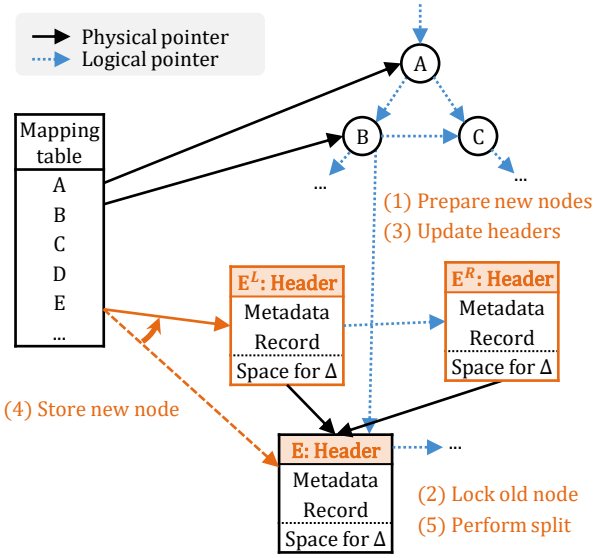

図 8 B^c-forest における分割操作

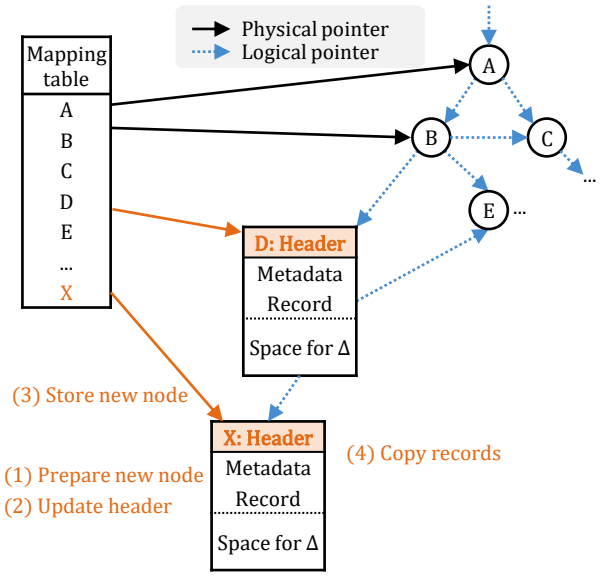

図 9 B^c-forest における下層作成操作

がそれぞれしきい値を越えており，統合後も十分なミュータブル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 8 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イミュータブル状態にした後，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，イミュータブル領域のレコードのみを検索する．

5.3 下層作成

下層作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，統合後も十分なミュータブル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．下層作成操作により，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 9 に B^c-forest における下層作成操作を示す．下層作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．下層作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．

下層作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，下層作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] 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).
[2] Levandoski, J., Lomet, D. and Sengupta, S.: The Bw-Tree: A B-tree for New Hardware Platforms, *Proc. ICDE*, pp. 302–313 (2013).
[3] Mao, Y., Kohler, E. and Morris, R. T.: Cache Craftiness for Fast Multicore Key-Value Storage, *Proc. EuroSys*, pp. 183–196 (online), DOI: 10.1145/2168836.2168855 (2012).
[4] 北川 博之：データベースシステム，昭晃堂 (1996).