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

2. 関連研究

近年では，様々な索引構造が研究されている．2 章では，ロックを取得しなロックフリー索引である B^c 木と，バイナリ比較可能なキーに対しキャッシュ効率を改善した 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と CAS 命令を用いたロックフリー索引である．B^c 木の概形を図 1 に示す．

2.1.1 木構造

マッピングテーブルはノード間のリンクを仮想化し，メインメモリ上の差分更新を可能とする．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 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．

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．

2.1.2 ノード構成

各ノードの領域はイミュータブル領域とミュータブル領域（ノード内バッファ）に分けられる．

イミュータブル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．ヘッダはイミュータブル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．

ミュータブル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ミュータブル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．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．各ノードへ構造変更操作を行う際は，構造変更後のノードから構造変更前のノードへ物理リンクを張り，古いノードへの参照を可能にする．なお，古いノードは構造変更が済み次第，ガベージコレクションへ追加され，他のスレッドから参照されないことが保証された時点で削除される．

2.2 Mass 木

Mass 木は B^+ 木を基本単位とした階層構造を作り，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 2 に示す．

Mass 木は複数の B^+ 木と layer 構造から構成される．Layer0 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，Layer0 で完結する．先頭 8 byte で一意性が確保出来ない場合，Layer1 を作成し，Layer0 から Layer1 への物理リンクを張る．Layer1 はキーの先頭 8〜15 byte で構成される B^+ 木の集合ある．例えば，Layer0 に「aaaaaaaab」をキーとするレコードが格納されており，そこに「aaaaaaaac」をキーとするレコードが書き込まれるとする．この場合，Layer0 には「aaaaaaaa」

[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

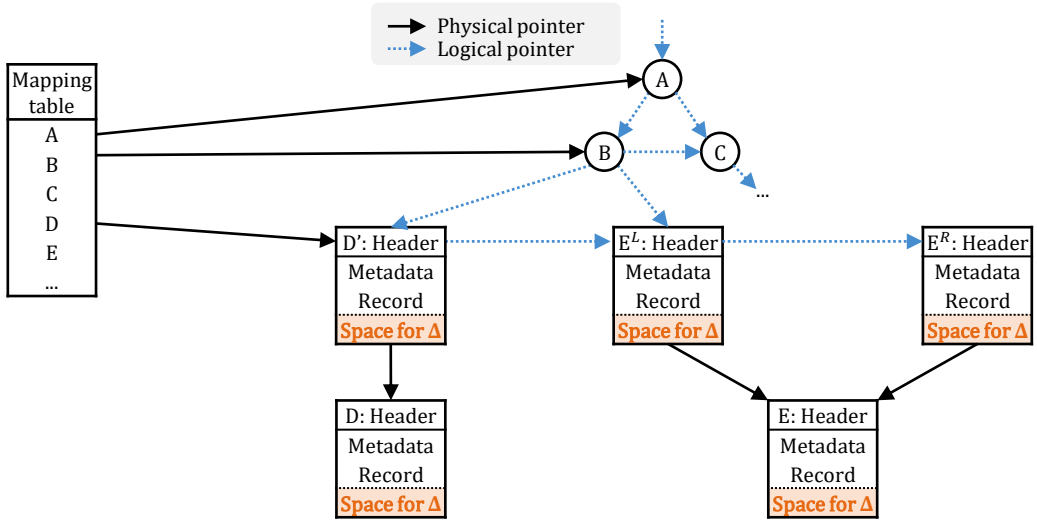

図 1 B^c 木の概形

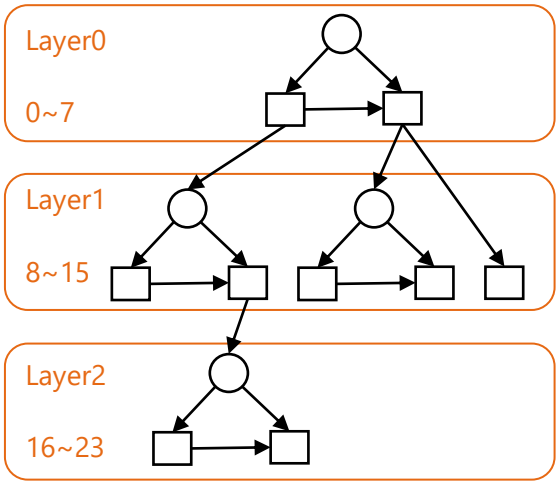

図 2 Mass 木の概形

をキーとし，Layer1 へのポインタをペイロードとして持つレコードが格納される．Layer1 には新たに B^+ 木作成され，「b」をキーとするレコードと「c」をキーとするレコードが格納される．つまり，同じ接頭辞キー「aaaaaaaa」持つ B^+ 木が生成される．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している．これにより，上記に示す階層化（共通部分の集約）が可能になり，キャッシュ効率を改善している．

3. B^c-forest の構造

B^c-forest は B^c 木を基本単位とする階層を持つ索引構造である．3 章では B^c 木との差異である，トライ木構造の適応と posting list の導入について説明する．

3.1 トライ木構造の適応

Mass 木は B^+ 木にトライ木構造を適応させることで，キャッシュ効率を改善させた．B^c-forest では B^c 木に対し，同様の改善を図る．

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 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．

3.2 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 3 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，layer1 を作成する．同様に 8〜16 byte 目が共通するため，layer2 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 layer1 では，1 つのレコードしか格納されていない状態で layer2 を作成している．layer1 にのみ注目すると，空間利用率は約 6.25 %しかない．

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 4 は posting list を導入した際の索引構造を示したものである．layer1 において posting list 作成することで，layer2 の無駄な階層化と空間利用率の悪化を防ぐ．

4. B^c-forest のノード操作

4.1 書き込み

4.2 読み取り

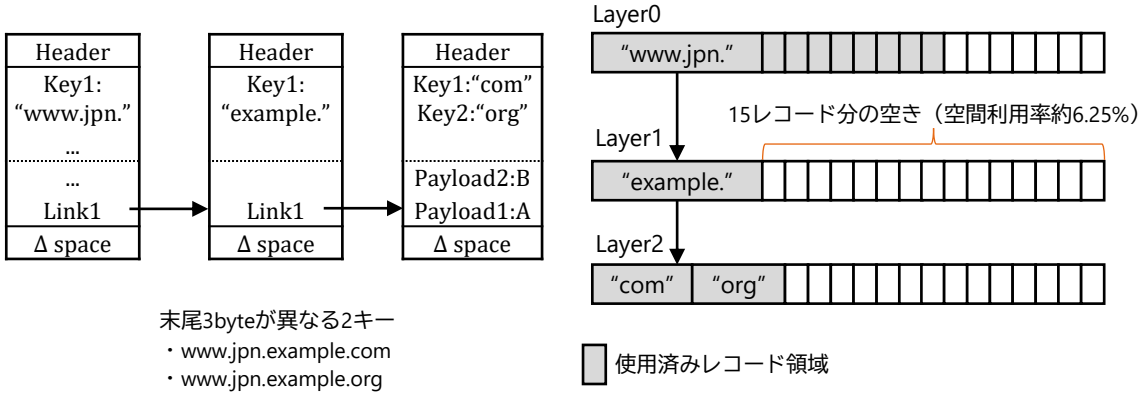

図 3 Mass 木の空間利用率

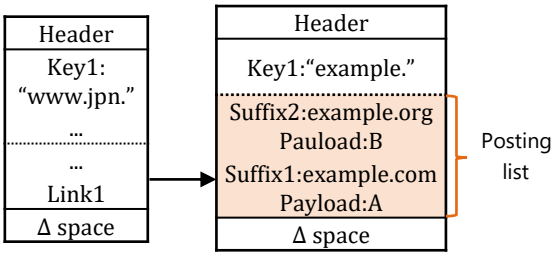

図 4 posting list の導入

5. B^c-forest の構造変更操作

5.1 統合

5.2 分割

5.3 新層作成

6. おわりに

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