2N-01

ロックフリー索引BzTreeにおける並列一括挿入法の実装

中山 宗 † 杉浦 健人 †† 石川 佳治 †† 陸 可鏡 ††

† 名古屋大学情報学部コンピュータ科学科 †† 名古屋大学大学院情報学研究科

1 はじめに

近年，ロックフリーアルゴリズムに基づく索引が注目されている．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ためである．BzTree（Bz 木）[1] は近年提案された B^+ 木のロックフリー拡張の 1 つであり，ロックフリーな動作により高い同時実行性を達成する．

しかし，元論文にて Bz 木の一括挿入法は提案されていない．一括挿入法とは，運用開始時に大量の入力データから高速に索引構造を構築する手法である．運用開始時の読み書きが行われない状況を想定し，その状況に合わせて工夫することで，個々にデータを挿入するよりも高速に索引を構築する．

そこで本稿では，Bz 木の一括挿入法として，シングルスレッドでの手法とマルチスレッドでの手法を提案する．まず伝統的なソートベースの手法 [2] に基づきシングルスレッドにおける一括挿入法について提案し，それをマルチスレッドに拡張する．

2 Bz 木の概要

Bz 木は multi-word compare-and-swap（MwCAS）命令を用いて B^+ 木を拡張したロックフリーの索引構造である．MwCAS 命令は compare-and-swap（CAS）命令の拡張であり，メモリ上の複数ワードを対象に値の比較と交換をアトミックに行う．Bz 木は MwCAS 命令を用いてレコードの挿入および木の構造変更におけるマルチスレッド間の順序付けを行い，ロックフリーな動作を実現している．

2.1 構造

Bz 木は，Bz 木の根を指すルートポインタと，複数の内部ノードと葉ノードからなる．全体像を図 1 に示す．ルートポインタが根となる内部ノードへのポインタ，内部ノードが子となる内部ノードまたは葉ノードへのポインタを持つ．これにより，全体として木構造を構築する．

図 1 に示されているように，各ノードはヘッダ，メタデータ，レコードを持つ．ヘッダには，ノードの大きさやレコード数，ノード更新時に必要な情報など，ノード自身の情報が格納される．内部ノードと葉ノードの区別も，ヘッダ内の情報を用いて行う．メタデータは 1 つのレコードに対して 1 つ存在し，対応するレコードの情報を格納する．レコードはキーとペイロードの組からなる．ペイロードは，内部ノードの場合は葉ノードへのポインタ，葉ノードの場合はデータそのもの，またはデータへのポインタである．

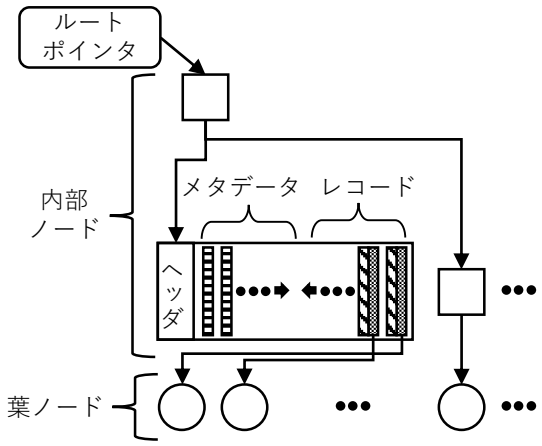

図 1 Bz 木の構造

3 一括挿入法の実装

索引構造における一括挿入の方法には伝統的なソートベースの手法が存在し，2 つの提案手法もこの手法に基づく．この手法は，挿入対象のレコードをキーでソートした後，ボトムアップに木構造を構築する．個々にデータを挿入する方法と比較して，レコードを挿入する葉ノードの探索や構造変更の時間を削減できる．以下では 3.1 節にてシングルスレッドでの，3.2 節にてマルチスレッドでの一括挿入法について述べる．

3.1 シングルスレッドでの一括挿入法

シングルスレッドでの一括挿入は伝統的なソートベースの手法に基づいており，レコードのソート，木構築の準備，葉ノードの作成，および葉ノードの挿入の 4 工程に大別できる．レコードのソート，木構築の準備を経て，葉ノードの作成・挿入を繰り返すことにより Bz 木を構築していく．それぞれの工程について詳細に述べていく．

■レコードのソート まず，挿入対象のレコードをキーの順に並び替える．この際，キーの重複するレコードが存在する場合は何らかの基準に従い削除し，特定のキーを持つレコードが唯一となるようにする．

■木構築の準備 ソートの次は，木構造の構築に向けて，Bz 木の根となるノードと右端ノードスタックを準備する．右端ノードスタックは図 2 に示すように，構築する木構造の右端にある全ノードへのポインタを保持するスタックである．このスタックの用途については，葉ノードの挿入の工程の詳細と共に述

Implementation of Parallel Bulk-Loading in a Lock-free Index BzTree
Shu Nakayama†, Kento Sugiura††, Yoshiharu Ishikawa††, and Kejing Lu††
†Department of Computer Science, School of Informatics, Nagoya University
††Graduate School of Informatics, Nagoya University

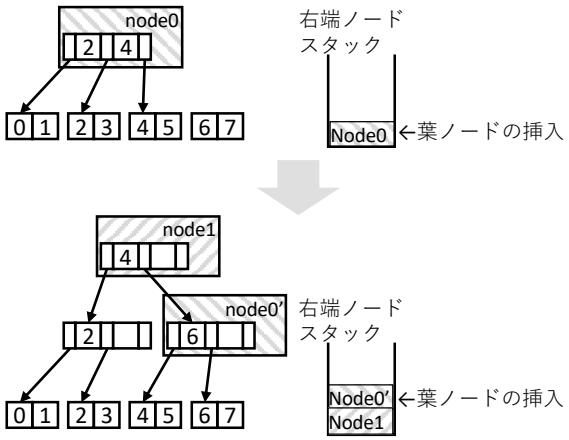

図 2　スプリット時の動作例

べる．

■葉ノードの作成　準備の完了後，ソート済みのレコードから葉ノードを作成する．空の葉ノードを作成し，未挿入の対象レコードを先頭から順に葉ノードへ挿入していく．レコードはソート済みのため，この葉ノード内のレコードもソート済みの状態となる．なお，レコードを挿入する際，そのレコードに対応するメタデータも作成し挿入する．レコードとメタデータの挿入を繰り返しそのデータサイズの合計が上限に達した場合，葉ノードの作成を終了し次の工程に移る．

■葉ノードの挿入　作成した葉ノードは，右端ノードスタックの先頭の内部ノードに挿入する．挿入後，その内部ノードの容量が上限に達していないか確認し，達していた場合はそのノードをスプリットする．スプリットしたノードの親ノードに対してもスプリットが必要か確認し，以降同様にスプリット操作を繰り返す．確認とスプリットが終了した後は次の葉ノードを作成するために前工程へと移る．

ここで，図 2 に示す例を用いて，この流れについて確認すると共に右端ノードスタックの有用性を示す．図は，3 つの葉ノードが挿入され，node 0 の容量が上限に達した状況を表している．この場合，node 0 がスプリットされた後，スプリットに伴い右端ノードスタックも更新される．その後，容量が上限に達するまで，右下の内部ノードである node 0’ に葉ノードの挿入が繰り返される．容量が上限に達した場合は，node 0’ をスプリットした後にその親ノードである node 1 に対してスプリットの必要の有無を確認する．このように，ソートベースの手法では右下の内部ノードに葉ノードの挿入を繰り返すことで葉ノード同士の順番を保つ．この影響でスプリットが生じる内部ノードは木の右端のノードのみとなるため，そのスタックを用意することで挿入先の葉ノードおよびスプリットが生じうる内部ノードの探索時間を削減できる．

全てのレコードを葉ノードへ挿入し中間ノードのスプリットが終了したとき，木構造の構築は完了するため構築結果として根ノードを返す．

3.2　マルチスレッドでの一括挿入法

マルチスレッドを用いた一括挿入法はマルチスレッドで複数の部分木を構築した後にそれら統合する手法であり，対象レコード群のソート，レコード群の分割，部分木の構築，および部分木の統合の 4 工程に大別できる．レコード群のソートの工程については，ソートをマルチスレッドで行う点以外はシングルスレッドのものと同様である．そのため，残りの 3 工程について詳細に述べていく．

■レコード群の分割　対象レコードのソート後，マルチスレッドで部分木を構築するために各スレッドに挿入対象のレコードを割り振る．この際，データ数とスレッド数を考慮して各スレッドの扱うデータ数がほぼ等しくなるように，レコードの先頭から順に割り振る．

■部分木の構築　各スレッドは割り振られたデータを対象に，シングルスレッドの場合と同様に部分木を構築する．そのため，本工程の終了時にはスレッド数だけ部分木がつくられる．

■部分木の統合　最後に，構築された複数の部分木をシングルスレッドで統合し全体の構築を完了する．基本的には Bz 木の根となる空の内部ノードを用意し，そこに各部分木の根を挿入していくことで Bz 木を構築する．しかし，文字列などの可変長データを直接索引に埋め込む場合など，部分木同士の高さは異なる可能性がある．そこで，このような場合は部分木同士の順に注意しながら，高い木の適切な内部ノードに低い木の根を挿入する処理を行うことで対処する．

4　評価分析

本稿で提案した 2 つの手法と，個々にデータを挿入する方法の 3 つの性能を比較する．具体的には，データ数，分布，キーの重複の有無，ソート済みか否かといった要素を変更した複数のデータセットに対する性能を比較する．

5　おわりに

本稿では，ロックフリー索引である Bz 木の概要を述べ，Bz 木の一括挿入法を提案した．今後は，4 章で述べたように提案手法の性能評価のための実験していく予定である．

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．

参考文献

[1] J. Arulraj, J. Levandoski, U. Minhas, and P. Larson, *BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory*, pp. 553–565. 2018.

[2] R. Ramakrishnan and J. Gehrke, *Database Management Systems*. 2002.