Vol.2017-OS-139 No.14
2017/3/2

並行実行木Masstreeの一括構築法

渡辺 敬之[1,a)]　川島 英之[3,b)]　三橋 龍也[2,c)]　建部 修見[3,d)]

概要：Masstree はトライ木のノード部分に B+木を格納する構造である．B+木，トライ木は，それぞれ一括挿入法による高速構築法が提案されてきた．一方，B+木とトライ木を混合した構造である Masstree の一括挿入法は我々の知る限り存在しない．Masstree は 16 並列アクセスまで性能がスケールするが，それを上回る一括構築法が存在すれば，運用開始時の待機時間を削減でき，運用者の負担を削減させられる．我々は Masstree の一括挿入法を本論文で提案する．提案手法では Masstree がトライ木と B+木から構成されている点に着目した．データを整列した後，トライ木のレベルでのデータ分散を行い，各トライ木において B+木の構築を行った．整列を高速に実行するために並列基数ソートを用いた．トライ木構築と B+木構築は逐次的に実行し，並列化を施さなかった．データ分布に一様分布を用いた場合，提案手法は既存手法に比べて 2.48 倍の性能を示した．データ分布に偏りを与えた場合，提案手法は既存手法に比べて 18 倍の性能を示した．

1. 序論

1.1 背景

複数のスレッドが同一のデータ構造に並行的にアクセスして READ 処理と WRITE 処理を行うとき，排他制御が必要になる．そのような排他制御を実現するデータ構造は並行データ構造と呼ばれる．並行データ構造における単純なアクセス方式は全体を停止させ，高々 1 つのスレッドにデータ構造へのアクセスを許す手法である．この手法はマルチコアが常識である現代において不適である．

これまで効率的な並行データ構造が多々研究されてきた．カウンタ操作については Compare-and-Swap 命令 [4] や Fetch-and-Add 命令 [4] が研究されてきた．カウンタよりも複雑なデータ構造であるキューやハッシュに関しても研究が行われ，ロックフリーキュー [11] やロックフリーハッシュ [10] が提案されてきた．

データベースシステムにおいては TPC-C ベンチマークで見られるように，範囲検索が頻繁に使われる．それゆえ範囲検索を効率的に実行するための木構造が重要である．古くは Lehman と Yao による並行 B+木構造 [6] が提案され，近年は Bronson らによる手法も提案されている [2]．

その中で高性能な並行木構造として，Masstree [8] がある．Massstree は B+木とトライ木を組み合わせた構造であり，文字列検索に対して効率的である．Masstree が注目を集めた理由のひとつにはその高い並行性がある．Masstree は全体停止を行わず，複数のスレッドが並行的にデータ構造へアクセスしても，性能が 16 スレッドまでスケールアップすることが論文で報告されている [8]．Masstree は高性能 OLTP システム Silo [13] で導入されており，また不揮発メモリを前提として 1000 万 TPS という高スループットを示す OLTP システム FOEDUS [5] の核となるなど，現代の高性能トランザクション処理システムの基礎を成している．

1.2 研究課題

Masstree は READ 処理と WRITE 処理が並行的に実行される時に高い性能を発揮することがこれまでの研究において分かっている．これはデータベースシステムの運用時に発生する状況である．一方，運用開始時点においては入力データが大量に存在するものの，そのための索引構造が存在しない状況がしばしば存在する．このような状況で運用を早期に開始するには，入力データから索引構造を高速に構築することが好ましい．このような場合において適切な手段は一括構築である．一括構築とは入力データの分布を調整することで，データ挿入時の分割処理コストを削減し，索引構築を高性能化する手法を指す．Masstree はトラ

[1] 筑波大学情報学群情報科学類
College of Information Science, University of Tsukuba
[2] 筑波大学大学院システム情報工学研究科
Graduate School of Systems and Information Engineering, University of Tsukuba
[3] 筑波大学計算科学研究センター
Center for Computational Sciences, University of Tsukuba
[a)] watanabe@hpcs.cs.tsukuba.ac.jp
[b)] kawasima@cs.tsukuba.ac.jp
[c)] mitsuhashi@hpcs.cs.tsukuba.ac.jp
[d)] tatebe@cs.tsukuba.ac.jp

ⓒ 2017 Information Processing Society of Japan

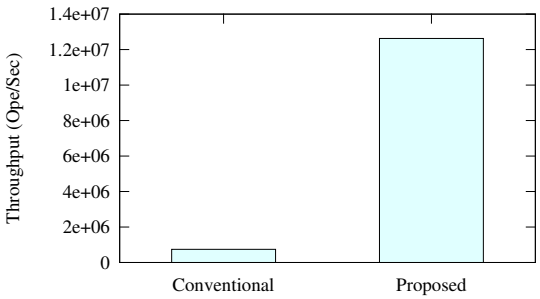

図 1　提案手法の効果

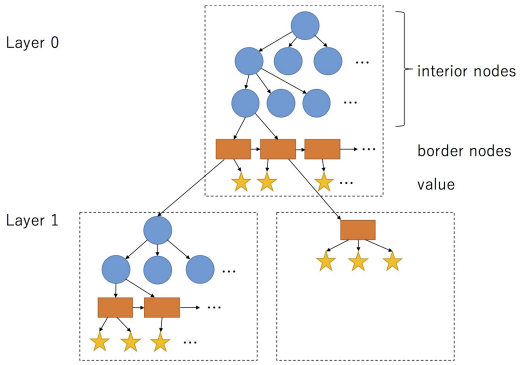

図 2　Masstree の構造 ([8] より引用)

イ木のノード部分に B+木を格納する構造である．B+木，トライ木は，それぞれ一括挿入法による高性能化手法がそれぞれ提案されてきた [7] [1]．一方，B+木とトライ木を混合した構造である Masstree の一括挿入法は我々の知る限り存在しない．Masstree は 16 並列アクセスまで性能がスケールするが，それを上回る一括構築法が存在すれば，運用開始時の待機時間を削減でき，運用者の負担を削減させられる．

1.3　貢献

我々は Masstree の一括挿入法を本論文で提案する．提案手法は Masstree がトライ木と B+木から構成されている点に着目する．データを整列した後，トライ木のレベルでのデータ分散を行い，各トライ木において B+木の構築を行う．整列を高速に実行するため，計算量が $O(N)$ である並列基数ソート [12] を実装して用いる．トライ木構築と B+木構築は逐次的に実行し，並列化を施さない．

Masstree は WRITE 処理が並行的に実行されても高性能を示すと考えられているが，データ分布によっては性能が劣化する．データ分布に偏りのある場合，提案する一括構築法は従来の逐次的な構築法に比べて 18 倍の性能向上を示す．これを図 1[*1]に示す．

1.4　論文構成

本論文の構成は次の通りである．2 章では Masstree とその再実装について述べる．3 章では提案手法である Masstree の一括挿入法を述べる．4 章では提案手法の評価結果を述べる．5 章では関連研究を述べる．6 章では本論文の結論を述べる．

2.　Masstree

2.1　概要

Masstree は B+木とトライ木の性能を合わせた高性能なデータ構造である．2^{64} のファンアウトを持つトライ木であり，トライ木のそれぞれのノードが B+木で構成されている．トライ木は prefix を共有させることによって長いキーを効率良く探索することができる．一方，B+木は短いキーに対してはとても効率が良い．よってどちらの側面も持っている Masstree はより効率良く探索を行える．

言い換えると，Masstree は一つ以上のレイヤーから成る B+木で構成されている．それぞれのレイヤーは異なった 8 バイトに区切られたキーによってインデックスされている．図 2 にこの例を示す．トライ木のルートノードであるレイヤー 0 には各キーの 0-7 バイト目の部分がインデックスされている．さらに 1 つ下のレイヤー 1 には各キーの 8-15 バイト目の部分がインデックスされている．次に深いレイヤー 2 では各キーの 16-23 バイト目の部分がインデックスされるというように以下同様に続いていく．つまり同じ h レイヤーにインデックスされているキー同士は同じ $8h$ バイトの prefix を持っているということである．

それぞれのレイヤーには 1 個以上の border ノードと 0 個以上の interior ノードが含まれている．border ノードとは一般的な B+木の leaf ノードとほぼ同じである．しかし，leaf ノードがキーとその value しか持たないのに対して，Masstree の border ノードは次のレイヤーへのポインターを持つ場合がある点が異なる．

2.2　レイアウト

Masstree のノード構造を図 3 に示す．Masstree の中間ノードと境界ノードはファンアウトが 15 の B+木である．境界ノードは削除と範囲検索のために連結している．*version, permutation* フィールドは並行更新時に使われるが，これは 2.3 節で述べる．

keyslice は 8 バイトの key slice を 64 ビット整数として保持する．境界ノードは key slice, length, suffix を保持する．Length は同一 slice で異なるキーを区別する．

同一 slice を有するすべてのキーは同じ境界ノードに保持される．これにより中間ノードは key length を保持不要になってスリム化される．このスリム化により，並行操作時に保持する不変量が削減され，分割コストが軽減される．

木構造ではキーの他に value を保持する必要がある．

[*1] これは図 12 からの抜粋であり，実験条件を含めた詳細は 4.5 節で述べられる．

ⓒ 2017 Information Processing Society of Japan

struct **interior_node:**
uint32_t *version*;
uint8_t *nkeys*;
uint64_t *keyslice*[15];
node* *child*[16];
interior_node* parent;

union **link_or_value:**
node* *next_layer*;
[opaque] *value*;

struct **border_node:**
uint32_t *version*;
uint8_t *nremoved*;
uint8_t *keylen*[15];
uint64_t *permutation*;
uint64_t *keyslice*[15];
link_or_value *lv*[15]
border_node* *next*;
border_node* *prev*;
interior_node* *parent*;
keysuffix_t *keysuffixes*;

図 3 Masstree のレイアウト ([8] より引用)

Masstree の場合，value は link_or_value 共用体が保持している．この共用体は，value か次のレイヤーに続いている場合にはそのレイヤーへのポインターが格納されている．どちらが格納されているかは *keylen* フィールドを見ることで判別する．

2.3 並行操作の概要

Masstree は細粒度のロック処理と楽観的並行実行制御によりマルチコアマシンで高い性能を発揮する．細粒度ロックの意味は，木の異なる部分への WRITE 処理が並列的に実行されることである．更新には局所的ロック（すなわちラッチ）のみが必要である．楽観的並行実行制御の意味は，READ 処理（例 : GET）でロックを獲得しないことである．そして READ 処理では大域的アクセス可能な共有メモリ領域にデータを書き込まない．共有メモリアクセスは衝突のために DRAM バンド幅を無駄にして性能劣化を引き起こすため，Masstree はそれを回避する．READ 処理では書き込み中のデータにアクセスする可能性があるため，書き手と読み手は協調してこの問題を回避する．この協調で使われる通信チャネルは *version* カウンタである．書き手は中間状態を書く前にこの状態を *dirty* とし，書き終えた後に値をインクリメントする．読み手はノードアクセス前に *version* を眺める．もしもノードの *version* が異なっている，あるいは *dirty* である場合，読み手は問題を認識し，リトライする．

2.3.1 *version* フィールド

version フィールドの構成を図 4 に示す．*locked* ビットは更新や挿入をする前にビットを立てる．*inserting, splitting* は挿入や split 処理を行っている間はそれぞれビットを立てる．*vinsert, vsplit* は挿入や split 処理が終わった後にインクリメントするカウンタである．このカウンタにより読み手がノードを探索している間に，書き手がそのノードに対して挿入や split 処理を終えた場合に，読み手がその処理を見落とすことを防ぐことができる．*isroot* はそのノードが B+木のルートノードかどうかを判定するビットである．*isborder* はそのノードが内部ノードか境界ノードかを判定するビットである．

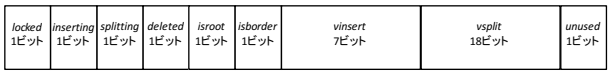

図 4 *version* フィールドの構成

2.3.2 *permutation* フィールド

Masstree の並行操作では，*version* カウンタに加えて *permutation* フィールドも重要である．一般的な B+木のリーフノードへの挿入操作を行う過程で，キーを並び替える必要がある場合には中間状態が作られる．この様子を図 5(a) に示す．図はリーフノードに“break”というキーを挿入したときの例である．ノード内のキーの格納場所を変更する必要があるため，一時的に中間状態が作られている．この対処法の一つは，読み手にリトライを強制することである．

一方で Masstree は，*permutation* フィールドを使うことにより中間状態を作ることなくアトミックな操作でノードへのキーの挿入を行うことができる．*permutation* フィールドとは，ノード内のキーの順序と，現在のノードに格納されているキーの数を簡潔に表している．書き手は *permutation* をコピーし，変更を行った後一度の書き込みで更新できる．そのため，読み手は新しいキーが挿入されていない状態の配置を見るか，新しいキーが挿入された新しい配置を見るかの 2 通りしか起こらないことになる．*permutation* を変更するだけで，実際のノード内のキーの配置は動かすことなくキーの挿入ができるので，読み手はリトライする必要がなくなる．

permutation フィールドは 64 ビットであり，さらにそれを 4 ビットごとに 16 等分したサブフィールドで構成されている．最も下位の 4 ビットはノードに格納されているキーの数を表す *nkeys* である．それに続く残りのビットは 15 の要素を持つ配列 *keyindex*[15] となっている．この配列には，0 から 15 までの数字が入っており，*keyslice*[15] の順番を表している．また，*keyindex*[0] から *keyindex*[*nkeys* - 1] までが存在しているキーとなっている．

permutation を使った挿入方法を図 5(b) に示す．図は B+木のときと同様に境界ノードに“break”というキーを挿入した場合の例である．図 5(a) との違いは *permutation* が追加されたことにより中間状態がなくなったことである．

挿入方法はまず，ノードをロックした後に現在の *permutation* をコピーする．ノード内のキーが挿入されていない空いている場所を探し，その空いている場所の番号を *permutation* 内で，挿入される適切な位置に移動させる．この図の場合は，“double”の右の 3 番目が空いているので，*permutation* の 3 の数字をキー“break”が入る適切な位置、つまり、0 と 1 の間に移動させる．*nkeys* をインクリメントする．先ほどの空いている 3 番目の場所にキー

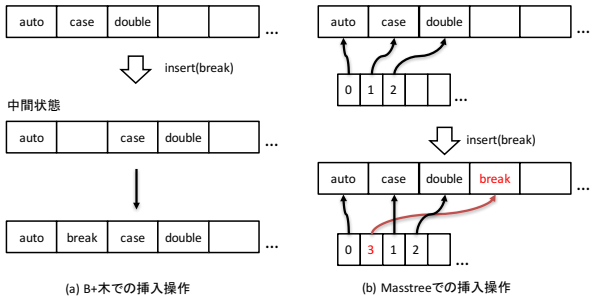

図 5　ノードへの挿入操作

“break”を挿入する．最後に，コピーして修正をした新しい *permutation* を書き戻し，ノードのロックを解除する．この最後のステップで初めて，読み手はキー“break”を見ることができるようになる．

2.4　書き手と書き手の調停

Masstree の書き手と書き手の調停にはノードごとのスピンロックを用いている．このノードのロックは *version* counter のビットで管理される．

ノードのキーや value に挿入や split などをして何か変更を加える場合，そのノードをロックする．また，ノードを split する場合には，同時に複数のロックを取る必要がある．例えばノード n が split するとき，n，新しく作った n の兄弟ノード，n の親ノードを同時にロックしなければならない．

2.5　書き手と読み手の調停

書き手と読み手の調停は楽観的並行実行制御を使用している．全ての挿入ワークロードにこの書き手と読み手の調停が必要となる．これは，挿入を行う際には必ず挿入したいキーがどこのノードに入るのか探索するからである．*version* フィールドを使った書き手と読み手の調停法は次のようになる．

書き手:　ノード n に変更を加える前に，*n.version* を dirty にする．変更を加えた後，このマークを消して *n.version* counter をインクリメントする．

読み手:　探索を行おうとしているノードで初めに *version* のスナップショットをとる．探索を行った後，スナップショットとその時の *version* を比較して変更がないかを確認する．もし変更があった場合には新しいスナップショットをとり，リトライする．

3.　提案：Masstree の一括挿入法

3.1　概要

Masstree は B+木同様に挿入時に split が生じる．従って初期化時に多数のデータから構造を構築する際には，一括挿入法により処理時間を短縮できる可能性がある．

我々が調査した限りでは Masstree の一括挿入法はこれまでに存在しない．本論文で我々はその手法を提案する．提案手法は Masstree がトライ木と B+木から構成されている点に着目する．データを整列した後，トライ木のレベルでのデータ分散を行い，各トライ木において B+木の構築を行う．

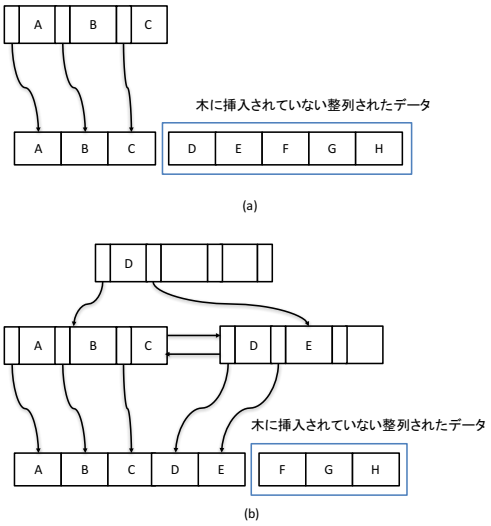

図 6　同一レイヤーでの一括挿入法による Masstree の構築

図 6 に同一レイヤーでの一括構築法による Masstree の構築の例を示す．まず，図 6(a) のように整列されたデータに対してノードにキーを入れていく．この際，一般的な Masstree の挿入方法とは異なり，ルートノードから探索を行う必要はなく，境界ノードからキーを挿入し，木を構築する．その際，境界ノードが最大数のキーを持っていない場合は，整列された順に境界ノードにキーを挿入していけばよい．次に境界ノードが最大数のキーを持ってしまった場合について説明をする．この場合，図 6(b) のように新しい境界ノードを作成する．一般的な Masstree の構築ではここで split 処理が行われるが，一括挿入法では split 処理を行わない．このとき，新しく作った境界ノードに入る最小のキー，つまり次に挿入されるキーを対象としていたノードの親ノードに挿入する．図 6(b) ではキー “D” が親ノードへ挿入されるキーにあたる．この親ノードへの挿入操作はリーフノードのときと同様であり，ノード内のキーの配置を変更する必要はなく，順番に挿入すればよい．これらの操作を繰り返すことで Masstree を構築していく．

3.2　設計と実装

挿入されるデータは文字列のキーと value を持つ．Masstree に一括挿入法でデータを挿入するように命令されたときに使用する機能について述べる．なお，提案手法の実装には C++言語を用いた．

3.2.1　データの整列

一括挿入を行う前に，挿入したいデータを整列する必要がある．これによりデータを挿入する時に，ノード内の

```
class Key:
    int ikey_len_;
    uint64_t ikey_;
    int len;
    char* s_;
    char* first_;
```

図 7　キーの構成

キーの入れ替えが生じなくなる．データの整列方法は様々な種類があるが並列基数ソート [12] を用いた．この並列基数ソートを使うことで，C++標準ライブラリの std::sort よりも高速にデータを整列することが可能となる．このソート方法は非 OpenMP のスレッドプールにより並列に実行するため，ソートの実行時に，非 OpenMP であるスレッドプールのフォーク/ジョインを行っている．

3.2.2　Masstree の構築

整列されたデータに対して Masstree の構築を行う．以降にその際に使用される機能について説明をする．

境界ノードの作成

まだ木が作成されていないときや，挿入したい境界ノードが最大数のキーを持っていてこれ以上キーが入らない場合には，新しい境界ノードを作成する．境界ノードの構造は図 3 の border_node の通りである．*version* フィールドは，0 に初期化をする．*permutation* フィールドは，実際のキーが挿入されている *keyslice* フィールドの順番通りにする．

内部ノードの作成

親ノードへキーを挿入する際に，親ノードが存在しない場合や，挿入したい内部ノードが最大数のキーを持っていてこれ以上キーが入らない場合には，新しい内部ノードを作成する．内部ノードの構造は図 3 の interior_node の通りである．境界ノードと同じく，*version* フィールドと *nkeys* は，0 に初期化をする．

キーの変換とシフト

Masstree のノードへキーを挿入する際には，キーを 8 バイトに区切り，uint_64 型に変換する必要がある．また，次のレイヤーに移動して挿入を行う際には，キーを 8 バイト分シフトして，それを当該レイヤーでのキーとする必要がある．図 7 に Masstree への挿入時のキーの構成を示す．*ikey_*フィールドは現在のレイヤーでのキー，*ikey_len_* フィールドはキーの長さである．*s_*フィールドはキーの suffix の先頭へのポインター，*len* フィールドはその長さである．*first_*フィールドはキー全体の先頭へのポインターである．

境界ノードへの挿入

境界ノードへキーを挿入するために，*permutation* フィールドの *nkeys* を参照してノードに格納されているキーの数を調べる．ノードが最大数のキーを持っていなければ，キーを *keyslice*[*nkeys*] に挿入し，*nkeys* をインクリメントする．この時，*permutation* フィールドの順序の部分は変更する必要はない．最大数のキーを持っている場合は，境界ノードの作成を行い，新しくできたそのノードにキーを挿入する．また，親ノードにもそのキーを挿入する．

内部ノードへの挿入

内部ノードへの挿入でも境界ノードへの挿入と同様に，*nkeys* を参照してノードに格納されているキーの数を調べる．ノードが最大数のキーを持っていなければ，キーを *keyslice*[*nkeys*] に挿入し，*nkeys* をインクリメントする．境界ノードの場合と異なり，最大数のキーを持っている場合には，split 処理を行い，半分のキーを新しく作ったノードへと移動させる．また，親ノードへその中間のノードを挿入する．

レイヤーの作成と移動

キーを挿入する際に，一つ前に挿入を行ったキーとそのレイヤーでのキーが被った場合，border_node が持っている *keylen* フィールドを参照する．このフィールドにはキーの長さの他に，レイヤーを持っているかどうかのフラグも保持している．レイヤーを持っている場合には，*lv* フィールドの *next_layer* から次のレイヤーに移動，キーのシフトをした後，そのレイヤーでキーの挿入を行う．

レイヤーを持っていない場合には，レイヤーの作成を行う．レイヤーの作成は，新しいルートノードを作り，そのノードへ *keysuffixes* にあるキーと現在挿入しようとしているキーを入れる．さらに，*lv* フィールドの *next_layer* にそのノードへのポインタを格納する．その後の処理はレイヤーを持っている場合と同様である．

3.3　アルゴリズム

本章で説明してきた一括構築法の流れを，疑似コードの形式で Algorithm 1 に示す．まず，data に対して並列基数ソートによるデータの整列を行う (line 1)．ルートノードを作成する (line 2)．データのキーと value を全て挿入できるまでループさせる (line 3-32)．

1 つ前に入れたキーと prefix が違う場合について見ていく．その場合は，現在対象としているノードにまだ新たなキーが入るスペースがあるかを調べる (line 23)．スペースがある場合，キーと value をノードに挿入して処理を終了する (line 30)．ノードへの挿入の方法は簡単で，並び替えは必要なく，端から順番に入れていくだけである．

キーを入れるスペースがない場合には，新しいノードを作り，そのノードと挿入しようとしているキーを，現在対象としているノードの親に入れる (line 24-28)．その後の処理はスペースがある場合と同様である．

次に 1 つ前に入れたキーと prefix が同じ場合を見ていく．その場合，そのキーがレイヤーを持っていない時には，新しいレイヤーを作る (line 13-15)．挿入したいキー

ⓒ 2017 Information Processing Society of Japan

を次の 8 バイトにシフトさせる (line 16)．対象としているノードを次のレイヤーに変更する (line 17)．その後に next_layer(line 10) の位置に移動して繰り返す．

Algorithm 1 一括挿入法

```
Data: Data data
 1: data ← multi-threaded radix sort;
 2: node ← create new border node;
 3: for i ← 0 to data.len - 1 do
 4:     while node.next ≠ NIL do
 5:         node ← node.next;
 6:     end while
 7:     n ← node;
 8:     key ← data.key[i];
 9:     value ← data.value[i];
10:     next_layer:
11:     nkeys ← n.nkeys;
12:     if n.keyslice[nkeys-1] = key then
13:         if n.lv[nkeys-1] does not have next layer then
14:             create new layer;
15:         end if
16:         shift key to next 8 bytes;
17:         n ← n.lv[nkeys-1].next_layer;
18:         while n.next ≠ NIL do
19:             n ← n.next;
20:         end while
21:         goto next_layer;
22:     end if
23:     if nkeys is full then
24:         n' ← create new border node;
25:         n.next ← nn;
26:         n'.prev ← n;
27:         insert n, n', and key to interior node;
28:         n ← n';
29:     end if
30:     insert key and value to n;
31:     n.nkeys ← n.nkeys + 1;
32: end for
```

4. 評価

本章では提案手法の評価を行うために実施した 4 種類の性能評価実験 (ソート，データサイズの影響，スレッド数の影響，データ分散の影響) について述べる．

4.1 実験環境

評価実験を行った環境を表 1 に示す．物理コア数は 24 である．

表 1 実験環境

OS	CentOS release 6.6 (Final)
Kernel	Linux 2.6.32-642.1.1.el6.x86_64
CPU	Xeon(R) E5-2695 v2 @ 2.40GHz
# of CPU cores	12 × 2
Memory	64 GB

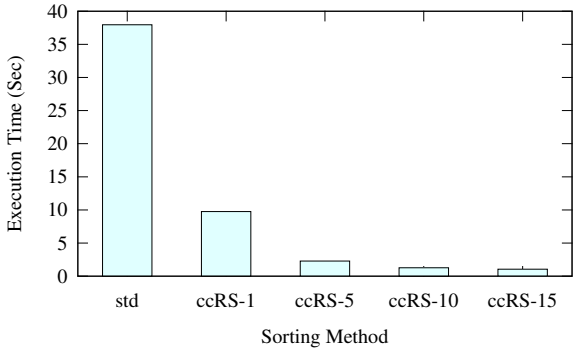

図 8 ソート

4.2 実験 1: ソート

4.2.1 実験内容

提案手法の初段ではソートが必要になる．ソートを高速に行うため，我々は文献 [12] に述べられている並列基数ソートを実装した．コードは GitHub で公開している [9]．この実験では並列基数ソートと C++標準ライブラリのソートに関してキー整列の実行時間を測定した．なお，キーは 10 バイト，データ件数は 1 億とした．

4.2.2 結果

実験結果を図 8 に示す．縦軸は実行時間である．std は C++標準ライブラリのソートを示す．ccRS-N は並列基数ソートを示し，N は並列数である．

(1) ccRS-N は std よりも高速であることがわかる．std は 38 秒程度を要するソート処理に，ccRS-1 は 9.8 秒程度しか要しない．従って並列性を用いずとも ccRS-1 は std よりも 3.9 倍程度高速であることがわかる．

(2) ccRS-15 の処理時間は 1 秒程度であり，それは ccRS-1 よりも 9.7 倍程度高速である．性能向上はコア数に対して完全に線形ではないために理想的ではないが，有意な性能向上が観察される．

以上より，ccRS-N はマルチコア環境で効率的であることがわかった．

4.3 実験 2: データサイズの影響

4.3.1 実験内容

データサイズと提案手法の影響を観察するため，提案手法，Masstree，B+木に関してデータサイズを変えながら挿入/検索スループットを測定した．実験においてキー長は 10 バイトとし，ファンアウトは 16 とした．いずれもスレッド数は 16 とした．スレッド数を 16 に設定した理由は，こ

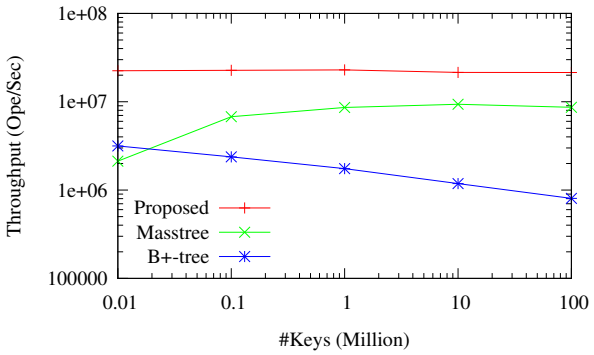

図 9　挿入におけるデータサイズの影響

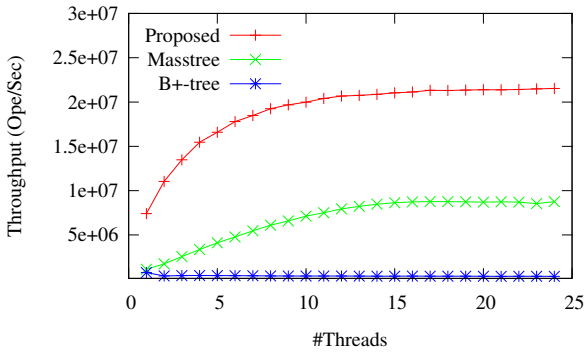

図 10　スレッド数の影響：PUT

の条件下で Masstree が最高性能を示したためである．換言すれば，スレッド数を 17 以上にした場合，Masstree の性能は改善しなかった．

4.3.2　実験結果

挿入に関する実験結果を図 9 に示す．この図より下記が観察される．

(1) Masstree が B+木よりも高い性能を示し，提案手法が Masstree よりも優れることが観察された．
(2) キー数が 1 億 (1e+08) の時，提案手法は Masstree，B+木に対してそれぞれ 2.48 倍，26.6 倍の性能向上を示した．
(3) キー数が増加するに従って B+木は性能が劣化するが，Masstree と提案手法では性能劣化が観察されない．

4.3.3　考察

この実験の結果から下記のことが考えられる．

(1) Masstree が B+木よりも高い性能を示した理由は，Masstree が大域的ロックを用いないからだと考えられる．2.3 節で述べたように，Masstree は細粒度のロック処理と楽観的並行実行制御によりマルチコアマシンで高い性能を発揮するように設計されている．我々の再実装した Masstree においても，この設計指針が適切に守られていたと考えられる．
(2) 提案手法が Masstree よりも高性能な理由は，提案手法が split を一切発生させないこと，ならびに並行処理におけるブロックが存在しないことだと考えられる．
(3) キー数の増加に伴って Masstree のスループットが劣化しない理由は，Mastree は挿入場所の探索が高速だからだと考えられる．

4.4　実験 3: スレッド数の影響

4.4.1　実験内容

提案手法のスケラビリティを観察するため，スレッド数を変動させながらスループットを測定する実験を行った．この実験では提案手法と Masstree に関してスレッド数を変えながら挿入スループットを測定した．キー長は 10 バイト，データ件数は 1 億とした．

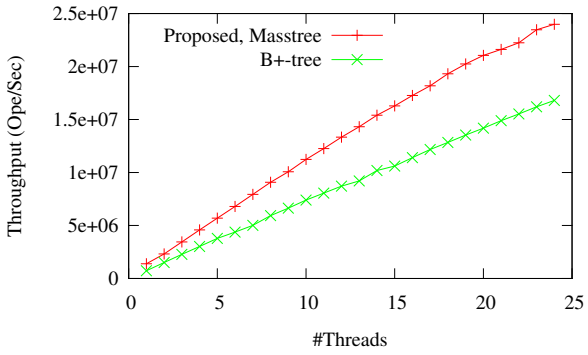

図 11　スレッド数の影響：GET

4.4.2　実験結果

実験結果を図 10 と図 11 に示す．縦軸はスループットである．この図より下記が観察される．

- PUT に関する結果
  (1) 提案手法と Masstree のいずれもがスレッド数の増加に対して性能向上することが観察された．
  (2) 提案手法は全てのスレッド数で Masstree の性能を上回ることが観察された．
- GET に関する結果
  (1) 提案手法と Masstree は同一の性能を示した．
  (2) 提案手法と Masstree は B+木よりも高速であり，最大で 58%の性能向上を示した．これはスレッド数が 7 の場合だった．

4.4.3　考察

この実験の結果から下記のことが考えられる．

(1) スレッド数増加に伴い提案手法の性能が向上した理由は，ccRS-N がそのような性質を有するからだと考えられる．なぜなら提案手法において並列性を活用する部分はソート部分しか存在しないからである．
(2) Masstree がすべてのスレッド数で提案手法に劣る理由は，split 処理が関係していると考えられる．提案手法のアプローチである，ソート&構築という方式は，通常の更新よりも基本的に優れることが示唆される．

ⓒ 2017 Information Processing Society of Japan

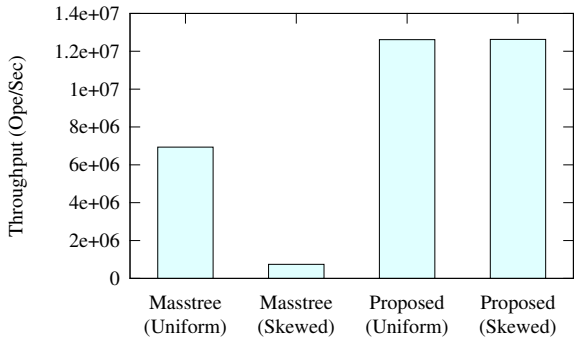

図 12 データ分散の影響

4.5 実験 4: データ分布の影響

4.5.1 実験内容

データ分布が異なると木構造の性能が大幅に変動することは一般的に知られており，一次元索引のみならず，空間索引においても指摘されることは衆目の一致するところであろう．

データ分布が提案手法に与える影響を観察するため，キーの分布を変動させてスループットを測定する実験を行った．この実験では提案手法と Masstree に両方に関して，キーが一様分布である場合と，偏りがある場合についてスループットを測定した．キー長は 20 バイトとし，データ件数は 1 億とした．データの分布はキーの最初の 8 バイト部分を全て同じにした．つまり，Masstree の一番上のレイヤーにはノードが 1 つだけしかない状況を意図的に作成した．

4.5.2 実験結果

実験結果を図 12 に示す．縦軸はスループットである．この図より下記が観察される．

(1) Masstree はデータに偏りがある場合，性能が大幅に劣化することが観察された．
(2) 提案手法はデータに偏りがあっても，性能が殆ど変動しないことが観察された．

4.5.3 考察

提案手法がデータ分布の変動に対して頑健である理由は，ロックを取らないからだと考えられる．マルチスレッドでデータ構造に並行的にアクセスを行う Masstree では，リーフノードを見つけてレイヤーを移るときにロックを取る必要があるため，この実験のように一番上のレイヤーにノードが 1 つしかない場合は性能が極端に劣化すると考えられる．

5. 関連研究

5.1 Atomic Operations

アトミック操作とは，命令に複数の操作を含み，どのシステムから見ても一つの操作に見えるものをいう．全操作が完了するまで途中の操作を見ることができない，一部の操作が失敗したときは，操作を行う前の状態まで戻るという条件を満たさなければならない．この実装はマルチコアを使った並列データ構造を扱う上で必須の手法である．

これまで，Compare-and-Swap 命令 [4] や Fetch-and-Add 命令 [4] のようなカウンタに関するアトミック操作の研究が行われてきた．この Compare-and-Swap 命令は 2 章で述べた Masstree でも利用している．Compare-and-Swap 命令とは，あるメモリ上の値を読み出した値と指定された値を比較し，等しいならば新しい値をその場所へ書き込むアトミック命令である．

Masstree では，ノードをロックするためのアルゴリズムの中でこの命令が用いられている．ロックはスピンロックで実装されており，スレッドがロックを獲得できるまでループを行い，この Compare-and-Swap 命令でロックされているかをチェックし，ロックを獲得する．

5.2 Concurrent Trees

OLFIT [3] は，B^{link} 木 [6] を楽観的並行実行制御で実装したものである．ノードが更新される毎にノードの version number を変更する．また，ノードを探索するときは，その前後にノードの version number をチェックする．もし，version number に変更があった場合にはリトライを行う．Masstree にはこのアイデアが使われている．

Bronson 法 [2] は，AVL 木を楽観的並列実行制御で実装したものである．こちらもノードの更新と共に version number の変更を行う．Bronson 法の場合は，version number を二つの部分に分けた構造になっている．これにより，ノードの更新の種類を判定することができるようになる．よって，探索を行う際のリトライの回数を減らすことができる．Masstree でも同様に，version number を複数に分けた実装を行っている．

6. 結論

Masstree はトライ木のノード部分に B+木を格納する構造である．B+木，トライ木は，それぞれ一括挿入法による高性能化手法がそれぞれ提案されてきた [7] [1]．一方，B+木とトライ木を混合した構造である Masstree の一括挿入法は我々の知る限り存在しない．Masstree は 16 並列アクセスまで性能がスケールするが，それを上回る一括構築法が存在すれば，運用開始時の待機時間を削減でき，運用者の負担を削減させられる．

我々は Masstree の一括挿入法を我々が知る限り初めて考案し，それを本論文で報告した．提案手法では Masstree がトライ木と B+木から構成されている点に着目した．データを整列した後，トライ木のレベルでのデータ分散を行い，各トライ木において B+木の構築を行った．整列を高速に実行するため，計算量が $O(N)$ である並列基数ソート [12] を用いた．トライ木構築と B+木構築は逐次的に実行し，並列化を施さなかった．

ⓒ 2017 Information Processing Society of Japan

データ分布に一様分布を用いた場合，提案手法は既存手法に比べて 2.48 倍の性能向上を示した．データ分布に偏りを与えた場合，提案手法は既存手法に比べて 18 倍の性能向上を示した．本研究では Masstree の一括構築法を創出したと結論する．

謝辞 本研究の一部は，JST CREST「ポストペタスケールデータインテンシブサイエンスのためのシステムソフトウェア」，JST CREST「EBD : 次世代の年ヨッタバイト処理に向けたエクストリームビッグデータの基盤技術」，JST CREST「広域撮像探査観測のビッグデータ分析による統計計算宇宙物理学」，科研費「#16K00150」による．

参考文献

[1] Achakeev, D. and Seeger, B.: Efficient Bulk Updates on Multiversion B-trees, *PVLDB*, Vol. 6, No. 14, pp. 1834–1845 (2013).
[2] Bronson, N. G., Casper, J., Chafi, H. and Olukotun, K.: A Practical Concurrent Binary Search Tree, *SIGPLAN Not.*, Vol. 45, No. 5, pp. 257–268 (2010).
[3] Cha, S. K., Hwang, S., Kim, K. and Kwon, K.: Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems, *27th VLDB Conference* (2001).
[4] Herlihy, M.: Wait-free Synchronization, *ACM Trans. Program. Lang. Syst.*, Vol. 13, No. 1, pp. 124–149 (1991).
[5] Kimura, H.: FOEDUS: OLTP Engine for a Thousand Cores and NVRAM, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 691–706 (2015).
[6] Lehman, P. L. and Yao, s. B.: Efficient Locking for Concurrent Operations on B-trees, *ACM Trans. Database Syst.*, Vol. 6, No. 4, pp. 650–670 (1981).
[7] Ma, D. and Feng, J.: A Generic Approach for Bulk Loading Trie-Based Index Structures on External Storage, *Web-Age Information Management - 15th International Conference*, pp. 55–66 (2014).
[8] Mao, Y., Kohler, E. and Morris, R. T.: Cache Craftiness for Fast Multicore Key-value Storage, *Proceedings of the 7th ACM European Conference on Computer Systems*, pp. 183–196 (2012).
[9] Mitsuhashi, R.: Implementation of Paralell Radix Sort, `https://github.com/ryumt/parallel-radix-sort` (2016), [Accessed 2017-2-7].
[10] Nielsen, J. P. and Karlsson, S.: A Scalable Lock-free Hash Table with Open Addressing, *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 33:1–33:2 (2016).
[11] Prokopec, A.: SnapQueue: Lock-free Queue with Constant Time Snapshots, *Proceedings of the 6th ACM SIGPLAN Symposium on Scala*, pp. 1–12 (2015).
[12] Satish, N., Kim, C., Chhugani, J., Nguyen, A. D., Lee, V. W., Kim, D. and Dubey, P.: Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort, *SIGMOD Conference*, pp. 351–362 (2010).
[13] Tu, S., Zheng, W., Kohler, E., Liskov, B. and Madden, S.: Speedy Transactions in Multicore In-memory Databases, *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 18–32 (2013).

ⓒ 2017 Information Processing Society of Japan