Vol.2017-OS-141 No.1
2017/7/26

並行実行木Masstreeにおける一括構築法の並列化

渡辺 敬之[1,a)]　川島 英之[2,b)]　建部 修見[2,c)]

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．トライ木構築は逐次的に実行し、B+木構築は並列的に実行した．提案手法は筆者らの既存手法と比べて 2 倍の性能を示した．B+木の構築部分だけに注目した場合，提案手法は既存手法に比べて 5.2 倍の性能を示した．

1. 序論

1.1 背景

人々の耳目を惹く並行木構造として，Masstree [10] 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 [10]．Masstree は高性能 OLTP システム Silo [13] で導入されたり，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 [6] 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．

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段がそれぞれ提案されてきた [9] [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．

1.2 研究課題

我々は Masstree の一括構築法を既に提案した [15]．この手法では 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] を実装して用いた．トライ木構築と B+木構築は逐次的に実行し，並列化を施さない．この一括構築法の効率化は限定的である．なぜならトライ木と B+木の構築における並列構築化を行っていないからである．

1.3 貢献

本研究では Masstree の一括挿入法における，トライ木と B+木の並列構築法を提案する．提案手法はまず，整列処理のデータに対して走査を施し，B+木ごとにタグを付ける．この走査は単一スレッドで実行される．次にタグ事

[1] 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
[2] 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)] tatebe@cs.tsukuba.ac.jp

に B+木を構築する．この構築処理は複数のスレッドで並列実行される．提案手法が筆者らの既存手法よりも効率的であることを実験的に示す．

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 とその一括構築法について述べる．3 章では提案手法である B+木構築の並列化を述べる．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. 準備

2.1 Masstree

2.1.1 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 1 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．

2.1.2 レイアウト

Masstree のノード構造を図 2 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．

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

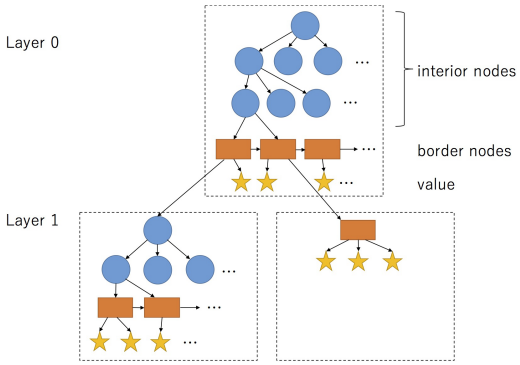

図 1 Masstree の構造 ([10] より引用)

```
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 2 Masstree のレイアウト ([10] より引用)

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．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．

2.2 従来の Masstree の一括挿入法

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．提案技法は 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 3 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 3(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，整列された順に境界ノードにキーを挿入

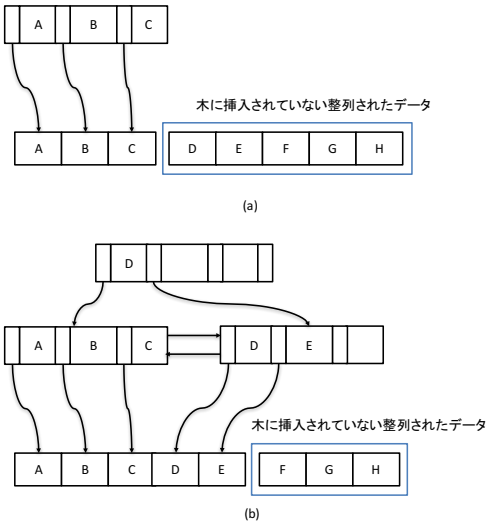

図 3　同一レイヤーでの一括挿入法による Masstree の構築

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 3(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 3(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. 提案：B+tree 構築の並列化

3.1 概要

Masstree ではトライ木のノードが B+木となるような構成になっている．つまり，キーが同じ 8 バイトの prefix を持っている場合，新しい B+木を子ノードとして作る．そしてそこにキーの同じ prefix 部分より後を挿入する．そのため，各データが属する B+木，その B+木の深さ，B+木同士の接続がわかれば，B+木を並列に構築することが可能である．

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 に一括構築法で並列にデータを挿入する際に使用する機能について述べる．提案手法は大きく分けて，データの整列，キーの分割，B+木の構築の 3 つに分けられる．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

一括構築法でデータを挿入するためには，挿入したいデータが整列されている必要がある．データの整列により，ルートからの木の探索，ノード内のキーの入れ替え，各 B+木のリーフノードでの split 操作を省略することができる．データの整列方法は並列基数ソートを用いて並列に整列できるようにした．

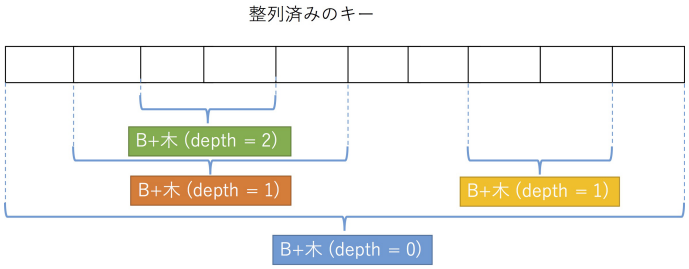

図 4　キー分割の例

class **TreeInfo:**
Tree* *tree*;
int *depth*;
int *start*;
int *end*;
TreeInfo** *next_layers*;

図 5　B+木の情報

3.2.2 キーの分割

整列されたデータに対してキーの分割を行う．各 B+木を構築するためには挿入されるキー，木の深さ，木同士の接続の情報が必要である．キーの分割とはこれらの情報を得ることである．

図 4 にキーの分割の例を示す．まず，整列済みのキーの最初の 8 バイト部分 (depth = 0) を先頭から順番に探索する．同じ prefix を持つキーが 2 つ以上あった場合，その範囲のキーは別の B+木に挿入されるキーである．そして，その範囲に対して次の 8 バイト部分 (depth = 1) を同様に探索していく．この処理を深さ優先探索で最後のキーまで行う．この図の例では，合計で 4 つの B+木を作る必要があることがわかる．

具体的には，新しい B+木を作る必要がある度に図 5 に示す TreeInfo クラスを追加する．TreeInfo クラスの変数について以下に述べる．*tree* 変数は B+木へのポインタであり，この段階では空の B+木を指している．木の構築の際，ここにデータを挿入していく．*depth* 変数はノードの深さを表している．言い換えると，この木にはキーのどの部分を挿入するかを表している．*start*, *end* 変数は木に挿入されるキーの始めと終わりを示している．*next_layers* 変数は木にリンクしている子ノードの TreeInfo クラスのポインタである．

3.2.3 B+木の構築

キーの分割処理で得た情報 (TreeInfo クラス) を元に B+木を構築していく．各 B+木の構築は 2.2 で述べたように行う．また，この各 B+木の構築はお互いに競合していな

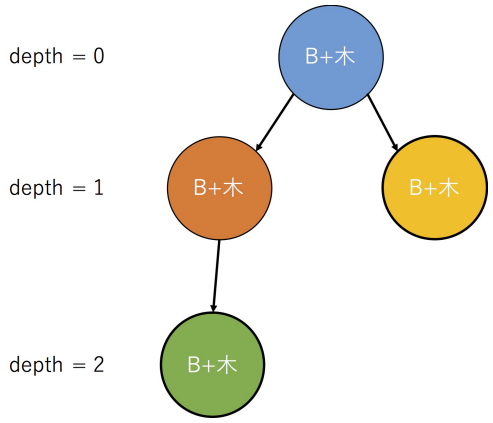

図 6　B+木構築の例

いため，並列に実行することができる．図 6 に図 4 での例で分割したキーを元に構築した例を示す．各 B+木には，分割処理で得た範囲のキーの 8*depth～8*(depth+1) バイト目が挿入される．

4. 評価

本章では提案手法の評価を行うために実施した 3 種類の性能評価実験 (スレッド数の影響，B+木の数の影響，実行時間の内訳) 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

4.2 実験 1: スレッド数の影響

4.2.1 実験内容

提案手法のスケー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実行時間を測定する実験を行った．キー長は 20，データ件数は 1 億とした．

4.2.2 実験結果

実験結果を図 7 と図 8 に示す．縦軸は実行時間，横軸はスレッド数である．original は一括構築法を使わずに挿入した Masstree，bulk_single は一括構築法を使い B+木の構築を逐次的に行なった Masstree，bulk_multi は一括構築法を使い B+木の構築を並列に行った Masstree (提案手法) である．また，共通部分とは整列にかかる時間のことである．

この図より下記が観察される．

- 全体の実行時間に関する結果
  (1) 全ての手法で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) 一括構築法で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．
- 共通部分を抜いた実行時間に関する結果
  (1) スレッド数が 1 のときを除いて提案手法が高速であることが観察された．
  (2) 提案手法ではスレッド数が 15 になるまでは性能がスケールすることが観察された．

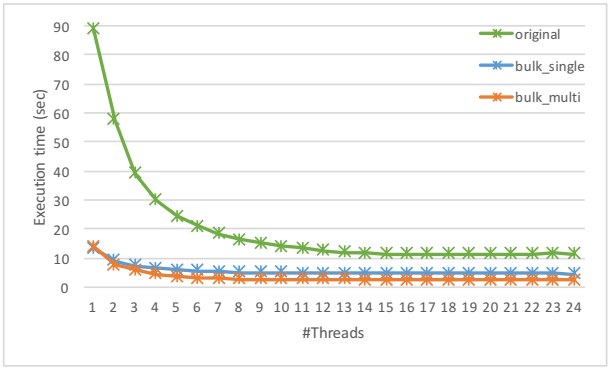

図 7　スレッド数の影響：全体の実行時間

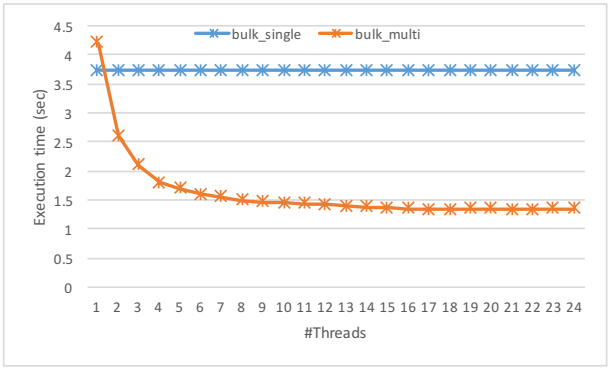

図 8　スレッド数の影響：共通部分 (整列の時間) を抜いた実行時間

4.2.3 考察

この実験結果から以下のことが考えられる．

(1) スレッド数増加に伴い一括構築法の性能が向上した理由は，整列部分を並列化しているからである．また，提案手法では B+木の構築にも並列化が利用されているため，bulk_single よりも高速である．

(2) スレッド数が 1 のときに提案手法より bulk_single が高速な理由は，提案手法にはキーの分割処理があるためである．また，スレッド数が 1 では B+木の構築部分で並列に処理できないことも原因である．

4.3 実験 2: B+木の数の影響

4.3.1 実験内容

作成される B+木の数により，性能にどのような影響を与えるのか実験を行なった．キー長は 20，データ件数は 1 億とした．提案手法である bulk_multi は 24 スレッドで実

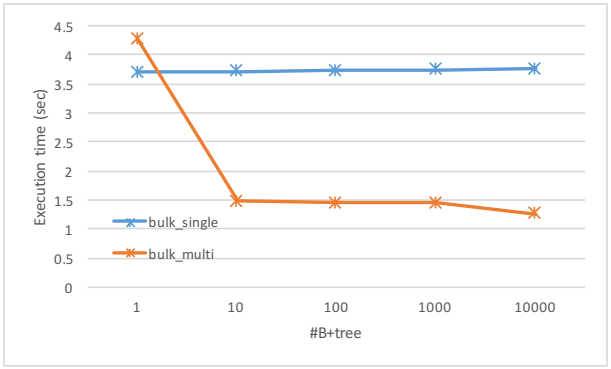

図 9　B+木の数の影響

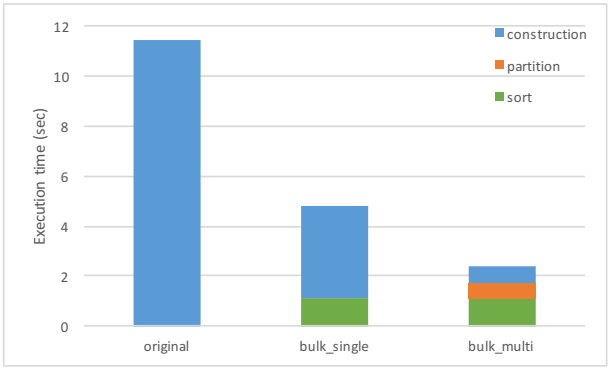

図 10　実行時間の内訳

験した．

4.3.2　実験結果

実験結果を図 9 に示す．縦軸は実行時間，横軸は作成した B+木の数である．bulk_single は一括構築法を使い B+木の構築を逐次的に行なった Masstree，bulk_multi は一括構築法を使い B+木の構築を並列に行った Masstree (提案手法) である．

この図より下記が観察される．

(1) 提案手法では B+木の数が 1 のときは性能が劣化していることがわかる
(2) 提案手法では B+木の数が 1 万のとき，bulk_single に対して 2.9 倍，B+木の数が 1 のときに対して 3.3 倍程度高速である．

4.3.3　考察

この実験結果から以下のことが考えられる．

(1) B+木の数が 1 のときに提案手法の性能が劣化する原因は，B+木を構築する際に並列化ができなくなるためである．つまり 1 スレッドで実行している場合と変わらない状態になっていると言える．
(2) B+木の数が多くなると性能が良くなる理由は，一つ一つの B+木に挿入されるキーの数が少なくなり，非リーフノードの数が減るので，余計な処理がいらなくなるからだと考えられる．また，提案手法では B+木単位でスレッドを割り振っているため，B+木が多い方が並列度が上がると思われる．

4.4　実験 3: 実行時間の内訳

4.4.1　実験内容

実行時間の内訳を観察するため，スレッド数を 24 としたときの実行時間を測定する実験を行った．キー長は 20，データ件数は 1 億とした．

4.4.2　実験結果

実験結果を図 10 に示す．縦軸は実行時間である．original は一括構築法を使わずに挿入した Masstree，bulk_single は一括構築法を使い B+木の構築を逐次的に行なった Masstree，bulk_multi は一括構築法を使い B+木の構築を並列に行った Masstree (提案手法) である．また，construction は B+木の構築時間，partition はキーの分割時間，sort はキーの整列時間を表している．

この図より下記が観察される．

(1) 一括構築法では構築以外にキーの整列もしなければならないが，合計しても Masstree より高速である．
(2) 一括構築法の並列化ではキーの分割処理も加わるが，それ以上に B+木の構築が高速になるため，性能は提案手法の方が良いことが観察される．
(3) 提案手法での B+木の構築時間は，bulk_single，original に対してそれぞれ 5.2 倍，16 倍の性能向上を示した．

4.4.3　考察

この実験結果から以下のことが考えられる．

(1) Masstree よりも一括構築法が高速な理由は，split 処理や木の探索時間を省けるからである．
(2) bulk_single よりも提案手法が高速な理由は，B+木構築の並列化が関係していると考えられる．キーの分割にかかる時間よりも B+木の構築を並列に処理する時間の方が短いことが観察された．

5.　関連研究

5.1　Cuncurrent Trees

OLFIT [5] は，B^{link} 木 [7] 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 [4] 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．

5.2 木の一括構築

木の一括構築手法には様々な手法がある．文献 [3] では sort-based, buffer-based, sample-based などの手法が挙げられている．我々の手法は sort-based である．木の一括構築では STR-R 木 [8,11,14] など空間索引に関する議論が多い．B 木に関する一括構築の近年の研究には MVBT に関するものがある [2]．B+木を並列構築してトライとして繋ぎ合わせる我々のアイディアに類似する技術は見当たらない．

6. 結論

本研究では Masstree の一括挿入法における，トライ木と B+木の並列構築法を提案した．提案手法はまず，整列処理のデータに対して走査を施し，B+木ごとにタグを付ける．この走査は単一スレッドで実行される．次にタグ事に B+木を構築する．この構築処理は複数のスレッドで並列実行される．提案手法が筆者らの既存手法よりも 2 倍の性能向上があることを実験的に示した．B+木の構築部分だけに注目した場合，提案手法は既存手法に比べて 5.2 倍の性能を示した．

謝辞 本研究の一部は，JST CREST JPMJCR1413，JST CREST JPMJCR1303，科研費 #16K00150，科研費 #JP17H01748 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] Achakeev, D. and Seeger, B.: Efficient Bulk Updates on Multiversion B-trees, *Proc. VLDB Endow.*, Vol. 6, No. 14, pp. 1834–1845 (2013).
[3] Bercken, J. V. d. and Seeger, B.: An Evaluation of Generic Bulk Loading Techniques, *Proceedings of the 27th International Conference on Very Large Data Bases*, pp. 461–470 (2001).
[4] 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).
[5] 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).
[6] 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).
[7] 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).
[8] Leutenegger, S. T., Lopez, M. A. and Edgington, J.: STR: A Simple and Efficient Algorithm for R-Tree Packing, *ICDE*, pp. 497–506 (1997).
[9] 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).
[10] 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).
[11] Mitsuhashi, R., Kawashima, H., Nishimichi, T. and Tatebe, O.: Three-dimensional spatial join count exploiting CPU optimized STR R-tree, *2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016*, pp. 2938–2947 (2016).
[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).
[14] You, S., Zhang, J. and Gruenwald, L.: Parallel Spatial Query Processing on GPUs Using R-Trees, *Workshop on Analytics for Big Geospatial Data*, pp. 23–31 (2013).
[15] 渡辺敬之，川島英之，建部修見：並行実行木 Masstree の一括構築法，Vol. 2017-OS-139, No. 14, pp. 1–9.