情報処理学会論文誌　データベース　Vol. 4　No. 2　114–125 (July 2011)

BF マージ結合：Bloom フィルタを保持する B+木を用いた低結合選択率に適した結合手法

的野 晃整[†1]　スティーブン リンデン[†1]
谷村 勇輔[†1]　小島 功[†1]

本論文では，結合不能タプルを読み飛ばしできるマージ結合アルゴリズム（BF マージ結合）と，それに用いる，各ノードに Bloom フィルタを付与するよう拡張した B+木（BB+木）を提案する．BF マージ結合は，2 つの BB+木の各ノードにおいて，子孫ノードの読み飛ばしのために，子孫集合が互いに素であるかどうかの判定を行う．この判定は，的野ら（2010）が提案した素集合判定可能な Bloom フィルタを用いる．BB+木の機能として，Bloom フィルタを遅延更新可能なレコードの追加と削除に加え，木全体を高速にメンテナンスする操作も提案する．実験を通して，BB+木および BF マージ結合の性能を評価した結果，BF マージ結合は選択率が低い等結合処理時に非常に効率的であることを確認した．

BF Merge Join: A Merge Join for Low Selectivity Using Extended B+ Trees with Bloom Filters

Akiyoshi Matono,[†1] Steven Lynden,[†1]
Yusuke Tanimura[†1] and Isao Kojima[†1]

In this paper, we propose a novel merge join algorithm, called BF merge join, that can skip dangling tuples using extended B+ trees, which we call BB+ trees, each node of which has Bloom filters to represent its descendant keys. BF merge join traverses BB+ trees, comparing nodes to check whether the nodes' descendants are disjoint ot not. This disjoint test uses Bloom filters that we have extended to be able to check whether they are disjoint. The proposed BB+ tree supports lazy scalable insertion and deletion. Furthermore, we propose a memtenance operation to updete entire tree. We evaluated the performance of our BB+ tree and BF merge join. The results show that a BF merge join can give performance improvements, especially in the case of low-selectivity equi-join queries.

1. はじめに

選択率の低い結合演算は，データベース問合せ処理の中でも不可欠な演算の 1 つであり，それは多様な分野で広く利用されている．バイオインフォマティックスや，天文学，ジオインフォマティックスなど，ほとんどの e-Science プロジェクトで非常に大規模なデータが日々生成されている．しかしながら，人が扱うことができるデータ規模はそれよりはるかに小さく，さらにその中に存在するであろう有用な知識となると，そのサイズは非常に小さい．実際，Atre ら[2] は，多くの e-Science プロジェクトでメタデータ記述のために利用されている，Resource Description Framework（RDF）[3] のデータに対する結合演算を 3 つに分類しており，そのうちの 1 つとして低選択率の結合をあげている．また，ソーシャルネットワークなど Web 上のデータ規模も情報爆発により拡大している．したがって，低選択率の結合演算の効率化は重要な課題である．

低選択率の問合せを改善するうえでの最大のボトルネックは，I/O コストである．I/O コストを減少させるための手法として，たとえば，不要データの読み飛ばし，圧縮，キャッシュ，プリフェッチなど，多くの手法がある．近年，データベースコミュニティで話題になっているカラム指向 DB[4),5)] も I/O コストの減少を目的とした手法である．カラム指向 DB は，データを行単位で格納するのではなく，列単位で格納する手法である．同一列では特にカーディナリティが低い場合に圧縮効率が良くなる傾向があり，また，射影によって削除されるカラムを最初から読み込む必要もない．

FlashJoin[6)] はカラム指向 DB を前提とした結合アルゴリズムで，これも I/O コストの減少を目的としている．まず，結合キーとなるカラムのみを読み込んで結合索引を構築し，その後それを参照しながら，非結合キーだが必要なカラムは後から読み込んで解を生成する．すなわち，遅延マテリアライゼーションである．FlashJoin はカラム指向 DB の従来の結合に比べ，非結合キーのカラムでは，すべてを読み込む必要はなく，結合可能タプルのみを読み込めばよい点が異なる．しかしながら，これらの手法はいずれも，結合キーのカラム

†1 産業技術総合研究所
National Institute of Advanced Industrial Science and Technology (AIST)

は，解に含まれないものであっても，読み飛ばすことはできない．

本論文で提案する手法は，このような結合結果に含まれない結合不能タプルを読み飛ばすためのマージ結合アルゴリズムと，それに用いるための索引として Bloom フィルタ[7] を付与した B+木を提案する．本論文では，提案マージ結合を *Bloom* **フィルタマージ結合**（*BF* **マージ結合**）と呼び，拡張 B+木を *Bloom* **フィルタ** *B*+**木**（*BB*+**木**）と呼ぶ．ソートマージ結合や，Zig-zag 結合[8]，Bucket skip マージ結合[9] など，従来のマージ結合で B+木を利用することは一般的である．BF マージ結合では，与えられた 2 つの BB+木で各ノードどうしを比較して，それぞれの子孫ノードの読み飛ばし判定のために，各ノードが保持する Bloom フィルタ間でビット論理積を行い，互いに素であるかどうかの判定を行う．BB+木に付与する Bloom フィルタは先行研究で提案した素集合判定を可能にするよう拡張した Bloom フィルタ[1] を用いる．

本論文の構成は次のとおりである．まず 2 章で，先行研究で提案した素集合判定可能な Bloom フィルタの概要を述べ，その後 3 章で，その Bloom フィルタを保持するよう拡張した BB+木とその操作として，レコードの検索，追加，削除，およびメンテナンスについて述べる．4 章では，与えられた BB+木を用いる BF マージ結合を提案する．5 章では提案した BB+木と BF マージ結合の性能を評価するため，実験を通して評価を行う．最後に 6 章で関連研究について述べ，7 章でまとめる．

2. 素集合判定に適した Bloom フィルタの拡張

本章では，先行研究[1] で提案した，素集合判定を可能にするよう拡張した Bloom フィルタを簡潔に述べる．まず，従来の Bloom フィルタの問題点について述べ，その後でその拡張について述べる．

従来の Bloom フィルタは，集合内に特定の要素が存在するかどうかを判定することを目的としたデータ構造で，特徴として空間効率に優れている点と，偽陽性による誤検出はあるが，偽陰性はない点がある．本研究では，2 つの集合間に共通する元が存在するかどうかを判定する目的で Bloom フィルタを利用する．従来の Bloom フィルタをこの用途で利用する場合，ハッシュ関数の数を k とすると，2 つのビット列間でビット論理積演算を行い，その結果のビット列のうち 1 であるビット数が k より少なければ，それらの集合は互いに素であることが保証できる．素集合判定の誤検出においても，偽陽性があるが，偽陰性はない．

しかしながら，従来の Bloom フィルタで素集合判定した場合，誤検出がほぼ発生してしまう．その理由は，1 であるビットが多すぎるためで，従来の Bloom フィルタではビット長 m とハッシュ関数の数 k を決定する際，最も空間効率を高めるために，1 であるビットの確率が 1/2 になるよう，要素数 n，メンバ判定の誤検出発生確率 q を入力とする以下の式を用いている．

$$m = -\frac{n \ln q}{(\ln 2)^2} \tag{1}$$

$$k \approx \frac{m}{n} \ln 2 \tag{2}$$

従来の Bloom フィルタを用いて，上述した手順で素集合判定を行ったとしても，ビット論理積演算の結果のビット列のうち，1/4 のビットが 1 であるため，ほぼ共通集合が存在するという判定になってしまう．

この問題を解決するために，先行研究で，Bloom フィルタを拡張した．拡張は次の 3 点である．

A) ビットが 1 である比率の抑制
B) 各関数の出力範囲を隔離
C) 他の関数の結果を入力する関数の導入

A) については自明ではあるが，ビットが 1 である確率を下げる拡張である．次に示す式を用いてビット長 m を求める．

$$m' = -\frac{kn}{\ln(1 - q^{\frac{1}{k}})} \tag{3}$$

B) は，長さ m のビット配列を長さ m/k の部分ビット配列に k 等分し，各関数の出力範囲が重複しないようにする手法である．具体的には，i 番目の関数が出力する範囲は，$[0, m)$ ではなく $[im/k, (i+1)m/k)$ にする（$0 \leq i \leq k-1$）．

C) では，従来のハッシュ関数に加え，**共起検査関数**と呼ぶ関数を導入した．共起検査関数は，他の関数の結果を入力とする関数で，2 つの役割を担っている．メンバ判定の際にはハッシュ関数と同様の役割で，素集合判定の際には，ある複数のビットが共起しているかどうかの判定に用いる．

共起とは，ある Bloom フィルタにおいて，任意の複数ビットが 1 つの要素の追加によって同時に 1 に設定されることをいう．たとえば，要素 x を追加したとき，ハッシュ関数 f_a と f_b を用いた結果，$i = f_a(x)$ 番目のビットと，$j = f_b(x)$ 番目のビットが 1 に設定されるため，i 番目のビットと j 番目のビットは要素 x において共起しているという．

共起検査関数は，複数の 1 であるビットの位置を入力とし，それらのビットが共起しているかどうかを判定するためのもので，たとえば，2 つのビット位置 i と j を共起検査関数

f に入力したとき，その結果である $f(i,j)$ の位置のビットが 1 であれば i と j は何かの要素によって共起している可能性があるが，ビットが 0 であればいかなる要素によっても共起していないことを保証できる．

先行研究での実験の結果，現時点で最適と考えられる素集合判定可能な Bloom フィルタは，ビット長 m のとき，利用する関数は，通常のハッシュ関数を 2 つと共起検査関数を 2 つの計 4 つを用いた場合であった．出力範囲はハッシュ関数 f_a が $[0, m/4)$ で，f_b が $[m/4, m/2)$ である．共起検査関数は，$f_c(x) = m/2 + mod(f_a(x) + f_b(x), m/4)$ と $f_d(x) = 3m/4 + mod(f_a(x) \times f_b(x), m/4)$ である．本研究ではこの素集合判定可能な Bloom フィルタの利用を前提とする．

3. Bloom フィルタ B+木

本章では，BB+木について述べるが，その前に B+木を簡潔に定義する．B+木では，次数と呼ばれる木構造内のノードの容量の尺度が定義される．次数 b の B+木では，あるノードで実際に保持しているエントリ数 e は $\lceil b/2 \rceil \leq e \leq b$ を満足する．ただし，ルートノードに限り $2 \leq e \leq b$ まで認められる．B+木は，葉ノード（leaf node）と内部ノード（inner node）の 2 種類で構成されており，葉ノードはキーと値のペアからなるエントリを保持し，内部ノードでは，キーと子ノードへのポインタのペアからなるエントリを保持する．内部ノードのキーとポインタは次の関係をつねに満足する．

$$keys(p_i) \leq k_i < keys(p_{i+1}) \leq k_{i+1} \quad (0 \leq i < e)$$

なお，$keys(x)$ はノード x（あるいはポインタ x が差す子ノード）の全子孫のキーの集合を意味し，p はポインタを意味し，k はキーを意味する．

3.1 BB+木の構造

提案する BB+木は，B+木の各内部ノードに次のデータを追加したものである．1) 最小キー k_{min}，2) Bloom フィルタ配列，3) ビット配列および，4) タイムスタンプの 4 つである．**図 1** は BB+木の構造を示した図である．図 1 の a) は内部ノードの構造，b) は木全体の構造である．葉ノードはそれが保持するキーを列挙した箱で示されており，内部ノードはキーの箱のリストとその直下の大きい箱で表す．内部ノードの最小キーはキーの箱のリストの左端で，ビット列は大きい箱の左端の縦の数字で `bits` と表記し，Bloom フィルタ配列はその右側で `current` と `old` と表記している．タイムスタンプは一番下で `time` と表記されている．

1) 各ノードのすべての子孫のキーのうち，最小のキーを保持する．すなわち，内部ノードのキーとポインタは以下の関係を満足する．

$$k_{min} \leq keys(p_i) \leq k_i < keys(p_{i+1}) \leq k_{i+1} \quad (0 \leq i < e).$$

なお，一般的に k_i は $keys(p_i)$ の最大値であるため，各ノードは最小値と最大値を保持することになる．

2) Bloom フィルタ配列は，e 個の Bloom フィルタシリーズで構成され，1 つの Bloom フィルタシリーズは，v 個の Bloom フィルタで構成される．すなわち，ビットの 3 次元配列である．本論文では，i 番目の Bloom フィルタシリーズの j 番目の Bloom フィルタを $\mathcal{F}[i,j]$ と表記する（$0 \leq i < e, 0 \leq j < v$）．$i$ 番目の Bloom フィルタシリーズは $\mathcal{F}[i,*]$ と表記し，i 番目のポインタのすべての子孫のキーを表現する．Bloom フィルタシリーズは最初のフィルタとそれ以外とに分類し，それぞれを**現フィルタ**と**旧フィルタ**と呼び，$\mathcal{F}[i,0]$ と $\mathcal{F}[i,1-]$ と表記する．また，図では `current` と `old` として表記し，図 1 a) にあるように，本来はビット列あるいはビット列の集合であるが，理解のためにキーの列として表記する．複数の Bloom フィルタを利用するアイデアは Almeida ら[10] も採用している．また，削除に対応するには，Counting Bloom フィルタ[11] を使うことで可能にはなるが，桁のオーバフローなどの他の問題が発生する．

1 つの Bloom フィルタシリーズにおける，現フィルタと旧フィルタは以下のルールを満足する．

- 現フィルタは，子孫キーのうち存在を保証できるもののみを表現する．
- 旧フィルタは，存在を保証できないものや，実在するが現フィルタにまだ含まれていないものを表現する．
- 現フィルタはつねに 1 つ，旧フィルタは長さの異なる複数のビット列からなる．
- 新しい現フィルタが作成されると，それまでの現フィルタは旧フィルタになる．このとき，同じ長さの旧フィルタに，ビット論理和によってマージする．
- 更新するのは現フィルタのみ，旧フィルタはいったん作成されると，削除されるまで更新されることはいっさいなく，現フィルタに戻ることもない．
- 現フィルタが存在するすべての子孫キーを表現し，かつ不要なものを含まない状態になったら，旧フィルタをすべて削除する．

提案する更新の基本コンセプトは，追加や削除の過程でアクセスする内部ノードのみ，すなわち，すべてのノードではなく読み込んだノードのみを更新することである．更新は，葉ノードにアクセスし，レコードの追加削除を行ったのち，そのノードに現存する他のレコードを保持したまま，根に向って戻る過程で経由する内部ノードで，保持したレコードの情報

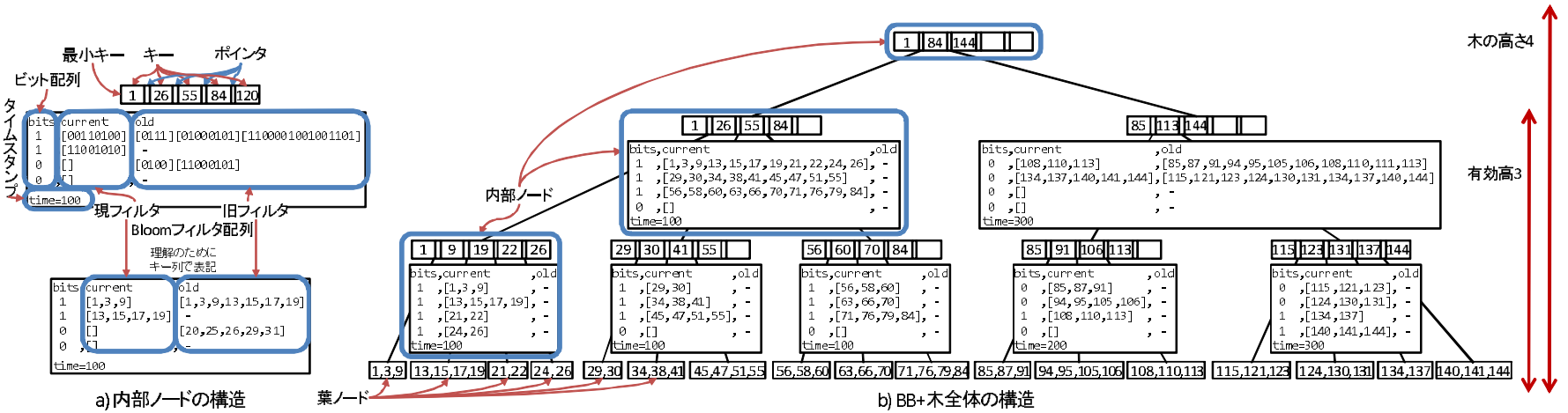

図 1　BB+木の構造
Fig. 1　The architecture of BB+ tree.

を更新する．これによって不要な I/O コストを減少できる．もし更新の度に必要なすべてのノードを更新するのであれば，旧フィルタやビット配列，タイムスタンプは不要であるが，それは非常に高コストである．すなわち，提案する更新は遅延更新の一種である．遅延更新のため，旧フィルタには誤った情報を含んでいるかもしれない．それは，たとえば，あるノードが 2 つに分割されるとき，それらの親ノードの現フィルタもまた分割する必要があるが，Bloom フィルタを分割することはできないため，分割する代わりに，2 つにコピーし，それを旧フィルタに移動させる．このように不要データも保持しているため，木全体を完全に一括更新するメンテナンス操作も必要である．メンテナンス操作については，3.6 節で述べる．

3) 内部ノードは長さ b のビット配列も保持する．各ビットは対応するフィルタシリーズの現フィルタが完全かどうかを表現するためのもので，i 番目のビットが 1 であれば，それは，i 番目のポインタの子孫キーのすべてが，i 番目の現フィルタ $\mathcal{F}[i,0]$ に格納されており，不要なものも含まれていないことを意味し，加えて，その子孫キーはすべてその先祖の現フィルタにも格納されていることを意味する．もし i 番目のビットが 1 になったら，i 番目の旧フィルタ $\mathcal{F}[i,1-]$ はすべて削除される．もしすべてのビットが 1 になったら，それはそのノードの子孫すべての情報を先祖に送ったことを意味するため，親ノードのビット列の 1 つを 1 に設定する．

4) 加えて，タイムスタンプは最新の現フィルタが作成された時刻を示す．基本的に，BB+木はルートに近ければ近いほどタイムスタンプは古い状態を維持する．もし，ある内部ノードのタイムスタンプの方がその親ノードより古ければ，その内部ノードの知らないうちに，親ノードが更新されていることを意味し，内部ノードがこれまで親ノードに送信して，親ノードの現フィルタに蓄積したはずのすべてのキーは，いつの間にか旧フィルタに移動している可能性がある．そのため，その内部ノードでは，ビット配列を 0 に初期化し，タイムスタンプを親ノードと同じ時刻に設定する．

3.2　ハイブリッド BB+木

Bloom フィルタのメンバ評価における偽陽性発生率 q は式 (3) によってビットの長さを決定することである程度保証できる．しかしこの式は，q を入力としてビット長を決定する式であるため，メンバ評価における偽陽性発生率は保証できても，素集合判定における偽陽性発生確率は保証できない．追加キーの数が多くなればなるほど，素集合判定における偽陽性発生確率は高くなる．すなわち，BB+木のルートに近いノードほど高くなり，Bloom フィルタによる素集合判定をしても，共通集合が存在するという判定を下すことがほとんどで，その必要性が薄れる．

Alg. 1: `Search` of a BB+ tree

```
1  Input: a key key
   Output: a record record
2  node ← root node;
3  while node is an inner node do
4  |  i ← binarySearch(node, key);
5  |  if key contains in a series of filters F[i,*] then
6  |  |  node ← node.p_i; /* ith pointer of node */
7  |  else /* there is no child which contains key */
8  |  |  return null;
9  i ← binarySearch(node, key);
10 if key = node.k_i then
11 |  return node.record_i;
12 else /* there is no record whose key matches key */
13 |  return null;
```

参照ノード *node* が内部ノードであれば 3–8 行目を，葉ノードであれば 9 行目以降を処理する．4 行目は二分探索によって目的のキーを検索し，Bloom フィルタを用いて，目的キーがメンバであるか評価し，メンバであれば *node* を子ノードに置き換え，メンバでなければ null を返す．葉ノードでも，二分探索を行い，目的キーが存在すればレコードを返し，存在しなければ null を返す．

この問題に対応するため，BB+木と B+木のハイブリッドを考えた．すなわち，BB+木のルートに近いノードは Bloom フィルタ配列および，ビット列，タイムスタンプを保持しない，ほぼ通常の B+木のノードである．このアイデアは S-tree [12] でも提案されている手法である．Bloom フィルタを保持する木の高さを指定する指標として，**有効高** h_a を導入する．$h_a = 3$ のとき，葉ノードの親と祖父が Bloom フィルタを保持するノードでそれ以上のノードは持たない．木の高さが h_t であるとき，有効高は $1 \leq h_a \leq h_t$ である．図 1 b) はルートノードは，内部ノードであるが Bloom フィルタ配列および，ビット列，タイムスタンプを保持していないため，有効高が 3 のハイブリッド BB+木である．以降，BB+木と表記する場合は，ハイブリッド BB+木のことを示す．

3.3　BB+木におけるキーの検索

BB+木に対する操作には，キーの検索，レコードの追加および削除，さらに従来の B+木にはない操作であるメンテナンスの 4 つがある．Alg. 1 にキーの検索手順を示す．従来の B+木における検索とほぼ同じであるが，異なる点は各内部ノードで子孫に目的のキーが存在するかどうかの判定を行い，もし存在すれば，該当の子ノードに遷移し，もし存在しなければ，null を返す．すなわち，従来の B+木ではかならず葉まで探索する必要があったが，BB+木では目的のキーが存在しない場合，葉まで到達しない．なお，Alg. 1 ではハイブリット BB+木に対する機構を省略してある．

Alg. 2: `Insertion` of a BB+ tree

```
1  Input: node node, record r, timestamp time_p
   Output: a set of keys, a set of keys, timestamp
2  if node is a leaf node then
3  |  node.records ← node.records ∪ r;
4  |  if node is not overflowed then
5  |  |  return (keys(node), ∅, current time);
6  |  else // node is overflowed
7  |  |  (l_l, l_r) ← Split(node);
8  |  |  return (keys(l_l), keys(l_r), current time);
9  else /* node is an inner node */
10 |  if node.time < time_p then
11 |  |  Set all bits of node.bits to 0;
12 |  |  node.time ← time_p;
13 |  i ← BinarySearch(node, r.key);
14 |  child ← node.p_i;
15 |  (k_l, k_r, t) ← Insertion(child, r, node.time);
16 |  node.time ← t;
17 |  node.k_i ← Max(child.k_*);
18 |  node.k_min ← Min(child.k_*, node.k_min);
19 |  if child is not overflowed then
20 |  |  Add k_l ∩ k_r into node.F[i,0];
21 |  |  SetBit(node, i, child);
22 |  |  return (k_l ∩ k_r, ∅, t);
23 |  else // child is overflowed
24 |  |  Divided children (c_l, c_r) ← Split(child);
25 |  |  Take a space at i+1 in node to store c_r;
26 |  |  if node is also overflowed then
27 |  |  |  Set all bits of node.bits[*] to 0;
28 |  |  else  Set node.bits[i] and node.bits[i+1] to 0;
29 |  |  Create current filters node.F[i,0] and node.F[i+1,0];
30 |  |  Copy old filters node.F[i,*] to node.F[i+1,*];
31 |  |  SetChild(node, i, c_l, k_l);
32 |  |  SetChild(node, i+1, c_r, k_r);
33 |  |  if c_l is in right in node then  k_l ← k_l ∩ k_r; k_r ← ∅;
34 |  |  else if c_r is in left in node then  k_r ← k_l ∩ k_r; k_l ← ∅;
35 |  |  return (k_l, k_r, t);
```

15 行目で再帰呼び出しされる再帰関数である．最初に呼ばれるときは，*node* はルートノード，r は追加レコード，$timp_p$ は親のタイムスタンプを意味するため，最も古い時刻である 0 を代入する．返り値として，葉ノードで取得したレコード集合を返す．オーバフローへの対応のために 2 つの集合を返す．*node* が葉であるとき（2–8 行）とそうでないとき（9–35 行）に場合分けされ，さらに前者は葉自体がオーバフローするかどうかで場合分けし，後者のときは子ノード *child* がオーバフローするかどうかで場合分けする．
ポイントしては，親とのタイムスタンプの比較は，10–12 行目で行い，ビット列を 0 に初期化するのは 26–28 行目で行われ，現フィルタの新規作成は 29 行目で行われる．

3.4　BB+木へのレコードの追加

Alg. 2 および，Alg. 3，Alg. 4 に BB+木へのレコード追加アルゴリズムを示す．なお，Alg. 2 ではハイブリット BB+木の機構に加え，ルートがオーバフローする場合の処理も省

Alg. 3: `SetChild` called from `Insertion`

```
1 Input: node node, int index, node child, keys keys
2 node.p_index ← child;
3 node.k_index ← Max(child.k_*);
4 Add keys into node.F[index, 0];
5 SetBit(node, index, child);
```

Alg. 2 の 31，32 行目で呼ばれる関数であるが，意味的なまとまりではなく，紙面の制約上 Alg. 2 への記載が困難であったためサブルーチン化した．

Alg. 4: `SetBit` called from `Insertion`

```
1 Input: node node, int index, node child
2 if child is an inner node then
3     if all bits of child.bits are set to 1 then
4         Set node.bits[index] to 1;
5     else Set node.bits[index] to 0;
6 else Set node.bits[index] to 1;
7 if all bits of node.bits are set to 1 then
8     for i ∈ (1..) do
9         Remove node.F[index, i];
```

Alg. 2 および Alg. 3 で，キーを現フィルタに追加した直後に，ビット列の状態を維持するために，呼ばれる関数である．子ノードが内部ノードで，子のビット列 *child.bits* がすべて 1 なら，関連するビットを 1 にセットする（3–5 行）．もし，ビット列 *node.bits* がすべて 1 になったなら，旧フィルタをすべて削除する（7–9 行）．

略してある．**図 2** は追加の例を示した図である．この例では，キーが 16 であるレコードを次数 4 の BB+木に追加したときの BB+木の状態の一部で，左が追加前，右が追加後を表す．中間は途中経過の特徴的な状態を示した図である．

キー 16 の追加により，葉ノード b_{112} とその親の内部ノード b_{11} は，追加前にすでに一杯であったために，オーバフローを起こす．Alg. 2 の *node* が b_{11} に，*child* が b_{112} に割り当てられているとき，24 行目から 35 行目が実行されて b'_{11} が得られる．次に 24 行目から 35 行目が実行されるときは，*node* に b_1 が，*child* に b'_{11} が割り当てられる．その結果，b'_{11} は a_{11} と a_{12} に分割され，b_1 は b'_1 の状態を介して，a_1 に変化する．

3.5　BB+木からのキーの削除

削除の手順を Alg. 5 に示す．なお，Alg. 5 でもハイブリット BB+木の機構とルートがアンダフローする場合の処理を省略してある．**図 3** は削除の例を示した図である．この例では，キーが 30 であるレコードを次数 4 の BB+木から削除したときの BB+木の状態の一部を示した図である．左が削除前，右が削除後，中間は途中経過の状態を示している．

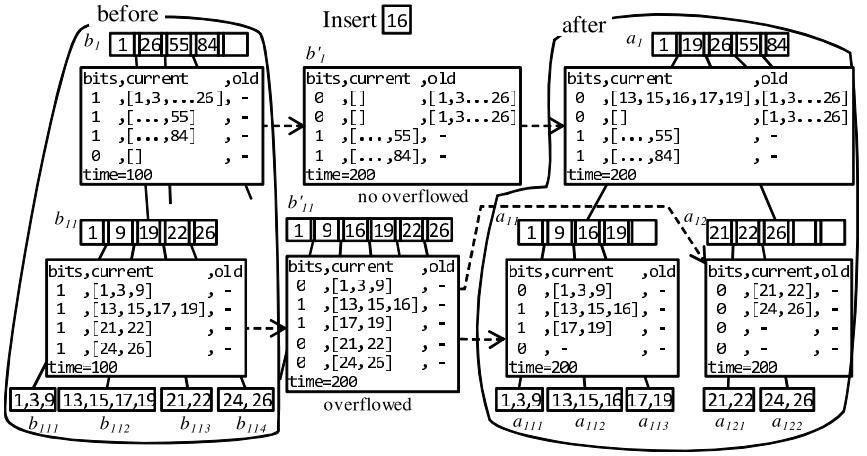

図 2　BB+木に 16 を追加する前と後
Fig. 2　BB+ trees before and after the insertion of 16.

キー 30 の削除により，2 つの葉ノード b_{121} と b_{122}，および 2 つの内部ノード b_{11} と b_{12} が，いずれも次数の半分しかエントリを保持していないため，アンダフローが発生する．Alg. 5 の *node* が b_{12} に割り当てられているとき，その子ノードの b_{122} がアンダフローを起こし，17 行目から 38 行目が実行される．この処理によって，b_{12} が b'_{12} に変更される．次に，b'_{12} がアンダフローしているので，17 行目から 38 行目が実行される．このとき *node* は b_1 で，これによって，b_1 は b'_1 を経て a_1 となり，a_{11} は b_{11} and b'_{12} から構築される．

3.6　BB+木のメンテナンス

BB+木は追加と削除で Bloom フィルタの遅延更新をサポートしており，完全な状態を維持するわけではなく，不要なデータを含んでいる．また更新性能も従来の B+木に比べると高コストである．そのため本論文では，BB+木全体を一括でメンテナンスする操作を提案する．Alg. 6 にメンテナンスアルゴリズムを示す．`Maintenance` を使うことで，BB+木へのバルクロードを可能にできる．つまり B+木を作成した後，それに対して `Maintenance` を実行することで，BB+木を作成できる．これまでのアルゴリズムでは，ハイブリット BB+の機能を省略していたが，Alg. 6 はそれを含んでいる．

Alg. 5: `Deletion` of a BB+ tree

```
1  Input: node node, record r
   Output: a set of keys, timestamp
2  if node is a leaf node then
3      node.records ← node.records − r;
4      return (node.keys, current time);
5  else /* node is an inner node */
6      i ← BinarySearch(node, r.key);
7      child ← node.p_i;
8      (keys, time) ← Deletion(child, r);
9      node.time ← time;
10     if node is root then
11         Create a current filter node.F[i, 0];
12         Set a bit node.bits[i] to 0;
13     else
14         Create all current filters node.F[∗, 0];
15         Set all bits of node.bits to 0;
16     if child is underflowed then
17         sbl ← node.p_j; //j is sibling node ID
18         if node is root then
19             Create a current filter node.F[j, 0];
20             Set a bit node.bits[j] to 0;
21         Copy old filters node.F[j, ∗] to node.F[i, ∗];
22         n ← #entries to move from sbl to child;
23         Move n entries from sbl to child;
24         if child is a leaf then
25             Add sbl.keys into node.F[j, 0];
26             SetBit(node, j, sbl);
27             keys ← keys ∩ sbl.keys;
28         else
29             Set sbl.bits[∗] to all 0's;
30             sbl.time ← time;
31             Move n filter series & n bits from sbl to child;
32         if sbl = ∅ then
33             node.k_min ← Min(child.k_min, sbl.k_min);
34             Remove sbl and jth entry of node;
35         else
36             if i < j then sbl.k_min ← Max(child.k_∗);
37             else child.k_min ← Max(sbl.k_∗);
38             node.k_j ← Max(sbl.k_∗);
39     Add keys into node.F[i, 0];
40     SetBit(node, i, child);
41     node.k_i ← Max(child.k_∗);
42     node.k_min ← Min(keys, node.k_min);
43     return (keys, time);
```

8 行目で再帰呼び出しされる再帰関数である．最初に呼ばれるときは，*node* はルートノード，r は削除レコードを代入する．返り値として，葉ノードで取得したレコード集合と削除時のタイムスタンプを返す．

node が葉であるとき（2–4 行）とそうでないとき（5–43 行）に場合分けされ，さらに後者は細かく場合分けされる．まず *node* がルートであるとき（10–12 行）とそうでないとき（13–15 行）に場合分けする．子ノード *child* がアンダフローした場合の処理が，16–38 行で行われ，その中でも細かく場合分けされる．18–20 行が *node* がルートのときの処理で，*child* が葉ノードのとき 24–27 行を，そうでないとき 28–31 行を実施する．また，兄弟ノードが空になっていれば 32–34 行を処理し，空でなければ 35–38 行目を処理する．深さ優先探索で，目的の葉ノードに到達し，目的レコードを削除した後，そのノードに残ったレコードと削除時刻を保持したまま，根に戻る．復路の過程で，Bloom フィルタやビット列，最大値，最小値を更新する．アンダフローが発生したときは，従来と同様に兄弟ノードからエントリを移動する．

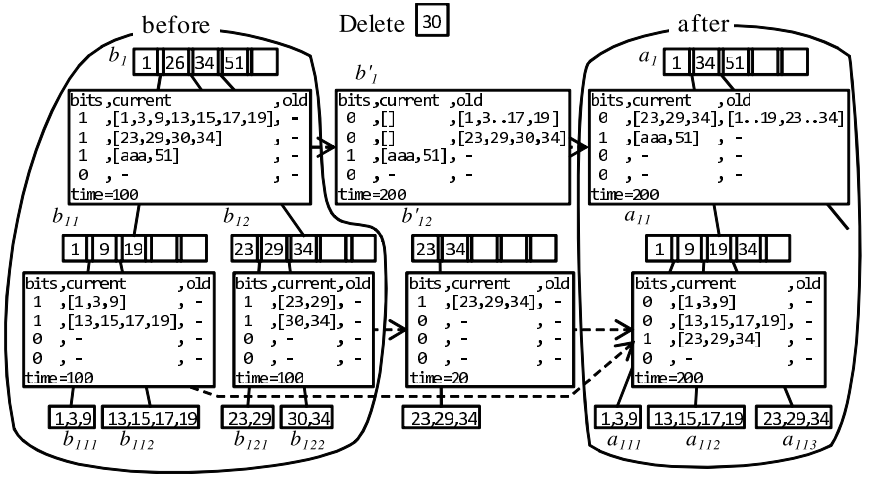

図 3　BB+木から 30 を削除する前と後
Fig. 3　BB+ trees before and after the deletion of 30.

4. Bloom フィルタマージ結合

本章では，2 つの BB+木を用いたマージ結合である BF マージ結合について述べる．本手法のポイントは 2 つのノードの子孫どうしが互いに素であるかどうかを判定する点で，そのノードが保持する Bloom フィルタを用いた素集合判定を行う．もし互いに素であれば，それらの子孫のノードにはアクセスする必要がなく，これは I/O コストが大幅に削減できることを意味している．

Alg. 7 および Alg. 8 に BF マージ結合のアルゴリズムを示す．relation はレコードの集合で構成され，bound は最小キー min，最大キー max および Bloom フィルタシリーズ $\mathcal{F}[i,*]$ で構成される．bound は内部ノードから生成される場合，min と max はそのノードが保持する隣りあうキーで，$\mathcal{F}[i,*]$ はそのキーの間のポインタの子孫キーを格納したものである．一方，葉ノードから生成される場合，min と max は同じキーで，$\mathcal{F}[i,*]$ は null とする．$lIterator$ と $rIterator$ は $left$ と $right$ からそれぞれ生成される反復子である．$next$ 関数は次の bound を返す．Alg. 8 は Alg. 7 から呼ばれる関数で，この関数内で素集合判定を行い，読み飛ばすかどうかを決定している．

Alg. 6: `Maintenance` of a BB+ tree

```
1  Input: node node, timestamp time, int available, double fpp,
          stack stack
2  if node is a leaf node then
3  |  A set of keys set ← stack.peek();
4  |  Add all keys in the leaf node to set;
5  else /* node is an inner node */
6  |  if available ≥ current height of node then
7  |  |  node.time ← time;
8  |  |  Set all bits of node.bits to 0;
9  |  else node.time ← 0; node.bits ← null; node.F ← null;
10 |  for i ∈ (0..) /* repeat all children */ do
11 |  |  child ← node.p_i;
12 |  |  stack.push(a empty set of keys);
13 |  |  child.Maintenance(child,time,available,fpp,stack);
14 |  |  A set of keys set ← stack.pop();
15 |  |  if available ≥ current height of node then
16 |  |  |  Remove all filters node.F[i,∗];
17 |  |  |  Create a current filter node.F[i,0] based on fpp;
18 |  |  |  Set a bit of node.bits[i] to 1;
19 |  |  |  Add set into child.F[i,0];
20 |  |  |  A set of keys parentSet ← stack.peek();
21 |  |  |  parentSet ← parentSet ∩ set;
```

深さ優先で探索する再帰関数である．木全体の Bloom フィルタを一括して更新できる関数で，有効高 *available* やメンバ判定の偽陽性確率 *fpp* も再設定できる．

探索の往路で，タイムスタンプを *time* に設定し，ビット列の初期化を行う（6–8 行）．復路で，旧フィルタを削除し，現フィルタを新規作成し，ビット列の 1 ビットを立てて，新規の現フィルタにすべての子孫キーを追加する（16–19 行）．すべての子孫キーを保持するためにスタック *stack* を用いている．

Alg. 7: `BF-Merge-Join`

```
1  Input: BB+ tree left, BB+ tree right
   Output: relation R
2  relation R ← ∅, bound l ← null , bound r ← null;
3  Initialize lIterator and rIterator using left and right;
4  while true do
5  |  if l is null then l ← lIterator.next(r);
6  |  if l is null then return R;
7  |  if r is null then r ← rIterator.next(l);
8  |  if r is null then return R;
9  |  if l and r are overlapped then
10 |  |  if l.min ≠ l.max ∧ r.min ≠ r.max then
11 |  |  |  if l.max >= r.max then r ← null;
12 |  |  |  else l ← null;
13 |  |  else if l.min ≠ l.max then l ← null;
14 |  |  else if r.min ≠ r.max then r ← null;
15 |  |  else /* Both l and r are leaf */
16 |  |  |  R ← R ∪ l.value + r.value;
17 |  |  |  l ← null; r ← null;
18 |  else /* r overtakes l or r undertakes l */
19 |  |  if l.max < r.min then l ← null;
20 |  |  else if r.max < l.min then r ← null;
```

left か *right* の次の bound がなくなるまで 4–20 行を繰り返す．もし l か r が null なら，`next` 関数によって次の bound を代入し，それでも null なら R を返す（5–8 行）．9 行以降は次に進めるべき bound を null に設定する．9 行目は，l と r の min と max のみを比較し，範囲が重なっているかどうかを評価する．10–12 行は l と r の両方が内部ノードのときに，13–14 行は l と r のいずれかが葉のときに実行される．両方が葉であれば，同じキーであるため，解として R に追加し，両方を null に設定する（15–17 行）．18–20 行は l と r の範囲が重なっていない場合で，小さい bound を null にする．

5. 評価実験

提案手法である BB+木と BF マージ結合の性能を評価するため，それぞれ実験を行った．実験では 1 台の計算機[⋆1]上で二次記憶装置として HGST HDD と Intel SSD[⋆2]を用いた．また，B+木の実装として，オープンソース Java プログラムである，jdbm[⋆3]を用い，これを改造して BB+木および BF マージ結合を実装した．

5.1 BB+木の性能評価

BB+木の性能評価実験は，レコードの追加処理時間と削除処理時間，木全体のメンテナンス処理時間を計測した．キーの検索についての性能評価は割愛する．なお，実験はいずれも Intel SSD 上で行った．その結果を**図 4** と**図 5**，**図 6** に示す．

図 4 と図 5 は，次数 16 の木に対して，20,000 から 100,000 までの異なるレコード数のときに，1 レコードを追加，あるいは削除する処理の平均時間（ミリ秒）である．比較は従来の B+木と，いくつかのパラメータの異なる BB+木である．BB+ tree(h=4, q=1e-6) の h は有効高で，q は Bloom フィルタのメンバ判定における偽陽性確率である．すなわち，BB+ tree(h=1, q=1e-5) は B+木とほぼ等しい．そのため，いずれの結果からもそれを判断できる．

いずれも BB+ tree(h=1, q=1e-5) 以外の条件では，B+木の更新性能より遅いことが分

⋆1 CPU: Intel Core2 Quad Q9450 (2.66 GHz), main memory size: 4 Gbytes, OS: Ubuntu Linux 10.04 (kernel 2.6.32)

⋆2 HDD: HGST Deskstar 7K1000.C HDS721050CLA362 (500BG, SATA, 7200 rpm, cache=16 MB, seek time=14 ms, transfer rate=1546 Mbps, latency=4.17 ms) と SSD: Intel X25-M Mainstream SSDSA2MH160G2C1 (160 GB, SATA, bandwidth=250/100 MB/s, latency=65/85 μs [read/write])

⋆3 JDBM: http://jdbm.sourceforge.net/

Alg. 8: `next` called from a `BF-Merge-Join`

```
1  Input: bound tgt
   Output: bound
   Global vars.: bound cur,
                 stack stack ← the root of the BB+ tree
2  NODE-LOOP: while stack ≠ ∅ do
3      node ← stack.peek();
4      if cur ≠ null then
5          if cur and tgt are overlapped then
6              if cur.max < tgt.max then
7                  if intersection(cur, tgt) then
8                      if cur is leaf then return cur;
9                      else
10                         node ← node.child;
11                         stack.push(node);
12             else if tgt is leaf then
13                 node ← node.child;
14                 stack.push(node);
15     BOUND-LOOP: while true do
16         cur ← the next bound of cur in node;
17         if cur ≠ null then
18             if tgt = null then return cur;
19             if cur and tgt are overlapped then
20                 if cur.max < tgt.max then
21                     if intersection(cur, tgt) then
22                         if cur is leaf then
23                             return cur;
24                         else
25                             node ← node.child;
26                             stack.push(node);
27                             continue BOUND-LOOP;
28                     else continue BOUND-LOOP;
29                 else return cur;
30             else
                   //cur overtakes tgt completely
31                 if tgt.max < cur.min then return cur;
                   //cur undertakes tgt completely
32                 else continue BOUND-LOOP;
33         else
34             cur ← null;
35             stack.pop();
36             continue NODE-LOOP;
37 return null;
```

bound *tgt* を入力とし，深さ優先探索に基づいて不要な枝を読み飛ばしながら，*tgt* に対応する次の bound を返す関数である．*tgt* と返り値の bound はそれぞれ異なる BB+木から生成されたものである．グローバル変数として，現在の bound *cur* と，ルートからの *cur* の元ノードまでのノードを格納するためのスタック *stack* を保持する．

このアルゴリズムが停止する（すなわち何かを return する）のは，`intersection` 関数によって *cur* と *tgt* が互いに素でないと判断された場合か，*cur* が *tgt* を完全に追い越して，範囲が重複しなくなった場合のいずれかである．次に `next` 関数が呼ばれるときは，グローバル変数を用いて，前に停止した木の位置から開始する．

2–36 行と 15–36 行の 2 つの繰返しで構成される．前者は *stack* が null になるまで，後者は次の bound がなくなるまで繰り返す．それぞれの繰返しフェーズは `NODE-LOOP` と `BOUND-LOOP` とラベル付けされている．`intersection` 関数は 7 行目と 21 行目で呼ばれ，与えられる bound *cur* と *tgt* の Bloom フィルタシリーズ間で，素集合判定が行われる．

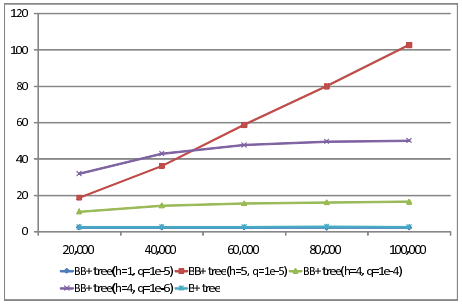

図 4　レコード数の増加に対する平均追加時間（ms）
Fig. 4　Insertion times for the number of records.

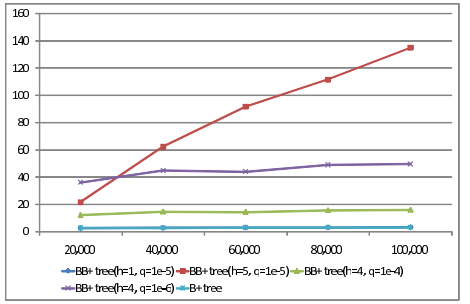

図 5　レコード数の増加に対する平均削除時間（ms）
Fig. 5　Deletion times for the number of records.

かる．これは Bloom フィルタ等内部ノードが保持するデータが余分にあるためで，従来の B+木ではほとんどの場合，葉ノードのみの更新でよいのに対し，BB+木では必ず内部ノードを更新する設計になっているためである．

BB+ tree(h=4, q=1e-4) と BB+ tree(h=4, q=1e-6) の差（つまり，h を固定し q を変化させた比較）と BB+ tree(h=1, q=1e-5) と BB+ tree(h=5, q=1e-5) の差（つまり，q を固定し h を変化させた比較）を見ると，前者の方が差が小さいことが分かる．これは，有効高も偽陽性確率も追加性能に影響を与えているが，より影響するのは有効高であることを示唆している．さらに，BB+ tree(h=5, q=1e-5) と BB+ tree(h=4, q=1e-6) の差（つまり，h と q の両方を変化させた比較）を見ると，速度が途中で逆転していることが分かる．これは，レコード数が少ないときは，有効高よりも偽陽性確率の方が影響が大きいことを意味している．

BB+ tree(h=4, q=1e-4) と BB+ tree(h=4, q=1e-6) に注目すると，最初は徐々に平均処理速度が遅くなっているが，60,000 を過ぎるとそれぞれ処理速度が一定になっている．BB+ tree(h=5, q=1e-5) では追加のグラフには表れていないが，削除のグラフは徐々に安定に向う傾向がわずかに見えている．このことから，木のレコード数が大きい場合は，それにかかわらず，処理速度は一定になることが予測できる．ただし，どこのあたりで一定化するかは，有効高に依存する．

従来の B+木に比べ更新性能は悪いことが分かったが，今回の実験条件では，最悪の場合で，追加では 100 ミリ秒，削除では 140 ミリ秒程度であるため，e-Science データのような更新頻度が高くない応用であれば，提案手法の更新性能でも満足できると考えている．

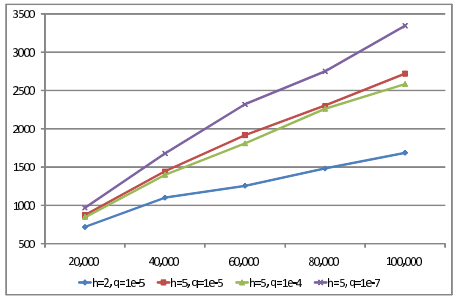

図 6　レコード数の増加に対するメンテナンス時間（ms）
Fig. 6　Maintenance times for the number of records.

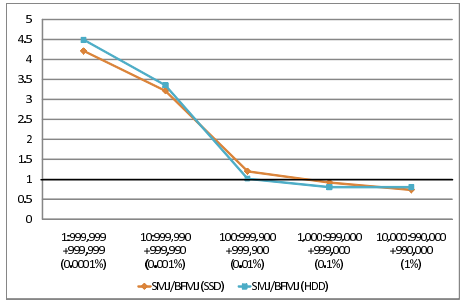

図 7　選択率に対する BFMJ と SMJ の結合処理時間の比率（$q = 10^{-6}$，$b = 16$，$h_a = 5$）
Fig. 7　Join processing time ratio of SMJ to BFMJ for various selectivities ($q = 10^{-6}$, $b = 16$, $h_a = 5$).

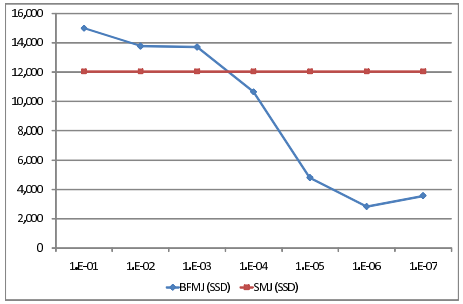

図 8　偽陽性確率に対する結合処理時間（ms）（1:999,999+999,999，$b = 16$，$h_a = 5$）
Fig. 8　Join processing times (ms) for various false positive probabilities (1:999,999+999,999, $b = 16$, $h_a = 5$).

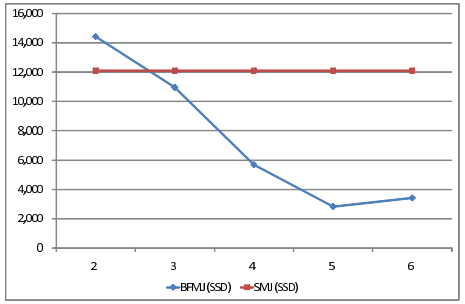

図 9　有効高に対する結合処理時間（ms）（1:999,999+999,999，$b = 16$，$q = 10^{-6}$）
Fig. 9　Join processing times (ms) for various available height factors (1:999,999+999,999, $b = 16$, $q = 10^{-6}$).

図 6 は，次数 16 で，20,000 から 100,000 までの異なるレコード数が格納済みの木に対して，1 回のメンテナンス処理の時間（ミリ秒）である．いずれもレコード数の増加にほぼ比例して処理時間が増加していることが分かる．これは，アルゴリズムが深さ優先探索でノードにアクセスして累積したキーに基づいて，内部ノードを更新する設計であるため，木の大きさに比例して依存していることを意味している．また，偽陽性確率が小ければ小さいほど，有効高が大きければ大きいほど，傾斜が急であるため，それらに性能は大きく依存しているといえる．h=2，q=1e-5 と h=5，q=1e-5 の差と，h=5，q=1e-4 と h=5，q=1e-7 の差を見ると，それほど大きくは違いはないため，偽陽性確率の有効高のどちらかの方が影響が大きいかは判断できない．絶対的な処理時間を見ると，100,000 レコードの木のメンテナンスでもたかだか 3.5 秒程度で，レコードの追加や削除のたびにメンテナンスするには実用的な時間とはいえないが，システムがアイドル状態のときにメンテナンスするような応用では，十分実用的な性能であるといえる．このため，メンテナンス機能を B+木に対して実施することで，バルクロードのように BB+木を効率的に作成することが可能であることも示している．

5.2　BF マージ結合の性能評価

本節では，ソートマージ結合と比較し，BF マージ結合の性能を評価する．実験では，それぞれを SMJ と BFMJ と表記する．実験は，2 個の BB+木 T_1，T_2（SMJ 評価なら B+木）を作成し，さらに 3 つの互いに素な集合 S_a，S_b，S_c を作成し，S_a を T_1 に，S_b を T_2 に格納し，S_c は T_1 と T_2 両方に共通集合として格納した．選択率を示すために $|S_c|:|S_a|+|S_b|$ と表示し，パーセント表示は $|S_c|/(|S_a|+|S_b|) \times 100$ で計算した値である．

図 7 に選択率が増加した場合の，BFMJ に対する SMJ の結合時間の比率を示す．すなわち 1 より高いと BFMJ が優れていることを意味する．BF マージ結合に用いた BB+木のパラメータは BB+木の偽陽性確率が $q = 10^{-6}$，次数が $b = 16$，有効高が $h_a = 5$ である．本実験では，Intel SSD と HGST HDD を用いて，それぞれでの傾向を見た．

図 7 から判断できることは，まず予測したとおり，選択率が低い場合は BFMJ が非常に効率的であることが分かる．また，SSD と HDD でもそれほど傾向に違いがないことが分かる．なおこれはあくまで傾向が類似しているのであって，絶対的な処理速度は大きく異なっていた．また，選択率が 0.01%までは傾斜が急だが，それ以降はほぼ横這いになった．このことから判断できることは，選択率が小さいときは，それが性能に大きく影響を与えるが，選択率が大きくなると，あまり性能に影響を与えないことを意味している．

図 8 と**図 9** は，SMJ と BFMJ の結合処理時間（ミリ秒）で，横軸はそれぞれ偽陽性確率と有効高である．BF マージ結合に用いた BB+木のパラメータはそれぞれ 1:999,999+999,999，$b = 16$，$h_a = 5$ と，1:999,999+999,999，$b = 16$，$q = 10^{-6}$ である．実験はいずれも Intel SSD で実施した．

図 8 と図 9 を見ると，両者とも類似した結果となっている．偽陽性確率は低ければ低いほど，有効高は高ければ高いほど，BFMJ が高速になっていることが分かる．本実験では，

偽陽性確率は 10^{-6} のとき，有効高は 5 のときが最も効率的で，その性能差は提案手法がおよそ 4.3 倍高速であった．また，偽陽性確率は $q = 10^{-4}$ 以下で，有効高は 3 以上で逆転しており，提案手法を利用する場合，そのように設定すればよいことが分かる．また注目すべき点として，偽陽性確率は 10^{-7} のとき，有効高は 6 のときにやや処理速度が劣化していることが分かる．これは，いずれも最適値が存在することを意味している．この要因は内部ノードが保持する Bloom フィルタが長くなり，データ量が増加したことで，I/O コストが増加したためで，性能の向上が限界に達したと考えられる．逆にいえば，Bloom フィルタ自体の I/O コスト増加による性能の劣化があったとしても，読み飛ばしによる I/O コストの減少による性能の向上がそれを相殺し，少なくとも最適値までは提案手法が有用であることを意味している．

6. 関 連 研 究

関連研究として，Zig-zag 結合[8] や Bucket skip マージ結合[9] は，結合不能タプルを読み飛ばすという目的が同じ研究である．Zig-zag 結合は B+木などの索引を利用したマージ結合の一般的な概念を示しているため，本手法も Zig-zag 結合の一種であるといえる．しかしながら，Zig-zag 結合は具体的なアルゴリズムや実験による性能評価への言及はなく，本手法と比較することはできない．Bucket skip マージ結合はソートされた集合を，いくつかのバケットに分割し，それぞれ最小値と最大値どうしを比較しながら，マージ結合する手法である．Bucket skip マージ結合はバケットのサイズが固定であるのに対し，提案手法は木の内部ノードどうしで比較するため，比較対象の要素集合の数が柔軟である．また，素集合の判定方法において，Bucket skip マージ結合はバケットの最大値と最小値のみで重複しているかどうかを判定しているのに対し，提案手法では素集合判定可能な Bloom フィルタを用いている点が異なる．

また，純粋な結合アルゴリズムではないため提案手法と比較はできないが，Bloom フィルタを用いた結合手法として，Bloomjoin[13] や Oracle DB 11g[14] がある．Bloomjoin は分散環境で，実際の結合前に行う半結合ステージで結合不能タプルを削除する手法で，結合不能タプルを他方に送信する必要をなくす手法である．Oracle DB 11g では，partition pruning のために各 partition が不要であるかどうかを判定するために Bloom フィルタを利用している．

S-tree[12] は，シグネチャファイルを B+木に保持するよう拡張している点で，BB+木に最も類似した研究であるが，S-tree のシグネチャファイルのビット長は，木の高さにかかわらず固定である点と，結合演算に関しての言及がない点が最も異なる．

7. ま と め

本論文では，結合選択率が低い等結合に適した BF マージ結合と，それに用いる Bloom フィルタが各ノードに付与された BB+木を提案した．

本提案手法は，e-Science における低選択率の結合演算を動機としており，特に RDF データを前提としているため，あくまで性能の傾向を評価できる規模と条件を想定して行った．その結果提案手法が有用であることを示すことができた．しかしながら，提案手法は，将来的にはより汎用的な用途も視野に入れることを前提としているため，提案手法の特性を正しく評価するためには，今後様々な状況・規模を想定した緻密な実験を実施し，提案手法の汎用的な用途を前提とした特性を決定したい．

また，機能の面での今後の課題として，一貫性の問題や，分散・並列環境への適用，またスター結合への応用などを検討している．

謝辞　本研究の一部は，総務省戦略的情報通信研究開発推進制度（SCOPE）092103006 の助成を受けたものである．

参 考 文 献

1) 的野晃整，スティーブンリンデン，谷村勇輔，小島　功：素集合判定に適した Bloom フィルタの拡張，日本データベース学会論文誌 DBSJ Journal，Vol.9, No.2, pp.7–12 (2010).
2) Atre, M., Chaoji, V., Zaki, M.J. and Hendler, J.A.: Matrix “Bit” loaded: A scalable lightweight join query processor for RDF data, *WWW*, Rappa, M., Jones, P., Freire, J. and Chakrabarti, S. (Eds.), pp.41–50, ACM (2010).
3) Manola, F. and Miller, E.: RDF Primer, http://www.w3.org/TR/rdf-primer/ (2004). W3C Recommendation 10 February 2004.
4) Copeland, G.P. and Khoshafian, S.: A Decomposition Storage Model, *SIGMOD Conference*, Navathe, S.B. (Ed.), pp.268–279, ACM Press (1985).
5) Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O’Neil, E.J., O’Neil, P.E., Rasin, A., Tran, N. and Zdonik, S.B.: C-Store: A Column-oriented DBMS, *VLDB*, Böhm, K., Jensen, C.S., Haas, L.M., Kersten, M.L., Larson, P.-Å. and Ooi, B.C. (Eds.), pp.553–564, ACM (2005).
6) Tsirogiannis, D., Harizopoulos, S., Shah, M.A., Wiener, J.L. and Graefe, G.: Query Processing Techniques for Solid State Drives, *SIGMOD Conference*, Çetintemel, U.,

Zdonik, S.B., Kossmann, D. and Tatbul, N. (Eds.), pp.59–72, ACM (2009).
7) Bloom, B.H.: Space/Time Trade-offs in Hash Coding with Allowable Errors, *Comm. ACM*, Vol.13, No.7, pp.422–426 (1970).
8) Garcia-Molina, H., Ullman, J.D. and Widom, J.D.: *Database System Implementation*, Prentice Hall (1999).
9) Kamath, M. and Ramamritham, K.: Bucket Skip Merge Join: A Scalable Algorithm for Join Processing in Very Large Databases using Indexes, Technical Report UM-CS-1996-020, University of Massachusetts, Amherst, MA, USA (1996).
10) Almeida, P.S., Baquero, C., Preguiça, N.M. and Hutchison, D.: Scalable Bloom Filters, *Inf. Process. Lett.*, Vol.101, No.6, pp.255–261 (2007).
11) Fan, L., Cao, P., Almeida, J.M. and Broder, A.Z.: Summary cache: A scalable wide-area web cache sharing protocol, *IEEE/ACM Trans. Netw.*, Vol.8, No.3, pp.281–293 (2000).
12) Deppisch, U.: S-Tree: A Dynamic Balanced Signature Index for Office Retrieval, *SIGIR*, pp.77–87, ACM (1986).
13) Mackert, L.F. and Lohman, G.M.: R* Optimizer Validation and Performance Evaluation for Distributed Queries, *VLDB*, Chu, W.W., Gardarin, G., Ohsuga, S. and Kambayashi, Y. (Eds.), pp.149–159, Morgan Kaufmann (1986).
14) An Oracle White Paper: Data Warehousing on Oracle RAC Best Practices (2008). http://www.oracle.com/technology/products/database/clustering/pdf/bp_rac_dw.pdf

(平成 22 年 12 月 20 日受付)
(平成 23 年 4 月 1 日採録)

（担当編集委員　義久 智樹）

的野　晃整（正会員）

2005 年奈良先端科学技術大学院大学情報科学研究科博士後期課程修了．博士（工学）．同年産業技術総合研究所にて特別研究員を経て，2007 年同研究所入所．現在，同研究所情報技術研究部門サービスウェア研究グループ研究員．RDF データベース検索の研究に従事．ACM 会員．

スティーブンリンデン

2004 年英国カーディフ大学より Ph.D.（マルチエージェントシステム）．2003 年から 2004 年英国ニューキャッスル大学にて Research Associate．2005 年から 2007 年英国マンチェスター大学にて Research Associate．2007 年より産業技術総合研究所情報技術研究部門サービスウェア研究グループ特別研究員．データインテンシブ分散システムの研究に従事．OGF DAIS Working Group Secretary.

谷村　勇輔（正会員）

2004 年同志社大学大学院工学研究科知識工学専攻博士課程修了．博士（工学）．現在，産業技術総合研究所情報技術研究部門サービスウェア研究グループ研究員．グリッドコンピューティング，クラウドコンピューティングの基盤システムに関する研究に従事．

小島　　功（正会員）

1958 年生．1984 年京都大学大学院工学研究科情報工学専攻修士課程修了．同年電子技術総合研究所入所．現在，産業技術総合研究所情報技術研究部門サービスウェア研究グループグループ長．データグリッドの研究に従事．ACM，IEEE 各正会員．OASIS メンバ．OGF DAIS Working Group Co-Chair.