DBSJ Journal, Vol.9, No.2
November 2010

素集合判定に適した Bloom フィルタの拡張

An Extension of Bloom Filters for Disjoint Sets Test

的野 晃整 ♥　スティーブン リンデン ♦
谷村 勇輔 ♠　小島 功 ♣

Akiyoshi MATONO　Steven LYNDEN
Yusuke TANIMURA　Isao KOJIMA

Bloom フィルタは集合内における要素の存在確認のための，空間効率に優れたビット配列のデータ構造である． Bloom フィルタによって二つの集合間の共通集合を表現するには，それらに対し論理積を行なうことで容易に実現できる．この仕組みは結合演算の効率化のために広く利用されている手法である．しかし集合が互いに素である場合に，それを示すには，従来では論理積後のビット列がすべて 0 であればよいが，そうなることはほぼ無い．本論文ではこの問題を解決するために Bloom フィルタを拡張する三つの提案を示した．ビットが 1 である確率の低下と，ハッシュ関数の出力範囲の隔離，他関数の結果を入力とする関数の導入である．評価実験を通して，従来では不可能であった素集合判定が提案手法では可能であることを確認し，かつ最も実用的な提案手法を特定した．

Bloom filter is a space-efficient data structure that had been proposed to test whether a set contains an element. In order to represent an intersection of two sets, the bitwise AND operation between their Bloom filters is performed. The mechanism is widely used to improve the performance of some join operations. Assuming the two sets are mutually disjoint, if the bit array after the AND operation are set all the bits to 0, then we can determine it. However such a situation is next to nothing if using the conventional Bloom filters. In this paper, to address this issue, we proposed three extensions for Bloom filters; keep the probability which a certain bit is set to 1 is low, isolate the output ranges of hash functions, and introduce functions that input the results of other hash functions. Through a series of experiments, we evaluated our approach can support disjoint sets test efficiently, and we found the most practical patterns of the extended Bloom filters.

♥ 正会員　産業技術総合研究所　a.matono@aist.go.jp
♦ 非会員　産業技術総合研究所　steven.lynden@aist.go.jp
♠ 非会員　産業技術総合研究所　yusuke.tanimura@aist.go.jp
♣ 正会員　産業技術総合研究所　kojima@ni.aist.go.jp

1. はじめに

Bloom フィルタ [1] は，ある要素が集合に含まれるかどうかを評価するために提案された，空間効率に優れたビット配列のデータ構造である． Bloom フィルタは様々な分野で利用されている．データベース分野でも問合せ処理，特に結合演算の効率化のために利用されている．Bloomjoin[3] や [4] は分散環境における通信コストを減らすために実際の結合を行う前に， 不要なタプルを削除する Semi-join フェーズで Bloom フィルタを利用している． また Oracle DB 11g[5] でも partition pruning の性能向上のために， 結果を持たない不要な partition を削除する，Bloom フィルタを用いる join-filter pruning を導入している．加えて，我々の以前の研究 [6] でも Bloom フィルタを利用して分散環境での結合演算時の転送量を削減する手法を提案した．

Bloom フィルタは集合の各要素をビット配列で表現したダイジェストであるため，複数の集合間に存在する共通集合を表現するには，それら配列のビット論理積を行うことで容易に実現できる．[3, 4, 5, 6] はこの特徴を利用した手法で，実際に各タプルでの結合演算を行う前に， Bloom フィルタの論理積を行って，解集合を表現するビット配列を得て，各要素が解に存在するかどうかを判定することで解候補を絞り込んでいる．

一方で，二つの集合が互いに素である場合，すなわち共通する元が一切存在しない場合を考える．このような場合，二つの集合が素集合であるかどうかの判定が容易に可能になれば，本来必要であった，その後の各要素の存在判定が不要になり，大幅な性能向上が期待できる．素集合判定の応用としては，結合演算，積集合，差集合など広範囲での応用が考えられる．

素集合判定の最も素朴な方法は，まず論理積を行ない，その結果のビット配列の全ビットが 0 であるかどうかを判定する方法である．しかしながら，二つの Bloom フィルタの論理積の結果のビット配列が 1 を含まないことは，従来の Bloom フィルタではほぼ無い．その理由は，従来の Bloom フィルタでは，任意のビットが 1 である確率は 1/2 になるように最適化されているためで，論理積後の配列での確率は 1/4 と非常に高い．そのため，実際には素集合であるにもかかわらず，共通集合が存在するという判定結果が下され，すなわち擬陽性が頻出し，各要素の存在チェックを逐一行うことになってしまう．

本論文では，この問題を解決するために， Bloom フィルタを拡張する次の三つの提案をする．

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

提案手法の性能を評価するため実験を実施した．実験では，三つの提案を施した 13 種類の Bloom フィルタと従来の Bloom フィルタとを比較した．比較内容は Bloom フィルタ本来の機能を保持できているかどうかの評価，素集合判定での誤判定の発生率の評価，素集合判定処理に要する時間の三つである．これらの評価の結果，従来法では不可能であった素集合判定が，提案手法では可能であることを確認した．また，評価した 13 種類の拡張

Bloom フィルタのうち，2 個のハッシュ関数と，一つあるいは二つの「他の関数の結果を入力とする関数」を持つ Bloom フィルタが最も実用的であることを特定した．

2. Bloom フィルタの定義

Bloom フィルタは，長さ m のビット配列と，k 個のハッシュ関数を用いる．ハッシュ関数はビット配列の位置，すなわち $[0, m)$ の範囲の整数を出力する[1]．実装としては SHA-1 や MD5 などの通常のハッシュ関数の結果を m で割った剰余を用いることが一般的である． 空の Bloom フィルタはすべてのビットが 0 である長さ m のビット配列である．ある要素 x を Bloom フィルタに追加するには，x を k 個のハッシュ関数に入力し，k 個の結果を得る．その結果が示す配列の位置のビット $array[f_1(x)], array[f_2(x)], \ldots, array[f_k(x)]$ をすべて 1 にする．なお，$array[a]$ はビット配列 a 番目のビットを示す．ある要素 y の検索，すなわち，y が集合に含まれるかどうかを調べるには， y を k 個のハッシュ関数に入力して得られた k 個の位置の配列のビット $array[f_1(y)], array[f_2(y)], \ldots, array[f_k(y)]$ のうち一つでも 0 であれば，その要素は集合に含まれない．逆にすべてが 1 であれば，その要素は集合に存在するか，あるいは実際には存在しないが，他の要素を追加したときに偶然全部 1 になったために誤判定したかのいずれかである．後者のように擬陽性による誤検出はあり得るが，偽陰性はない．また，要素の削除はできない．ただし， Counting Bloom フィルタ [2] は，ビットの代わりに整数を用いたものであるため，削除も可能になる．

ハッシュ関数が配列の各位置を同じ確率で出力する前提を置くと，要素をひとつ追加したとき，あるビットが選択される確率は $1/m$ で，逆に選択されない確率は $1-1/m$ である． したがって，k 個のハッシュ関数を用いて，n 個の要素を追加した後のあるビットが未だに 0 である確率は次のようになる．

$$\left(1 - \frac{1}{m}\right)^{kn}$$

逆に，任意のビットが 1 である確率 p は次のようになる．

$$p = 1 - \left(1 - \frac{1}{m}\right)^{kn}$$

$m \to \infty$ と仮定すると，テイラー展開より，以下に近似できる．

$$p \approx 1 - e^{-\frac{kn}{m}} \tag{1}$$

ある要素の検索時に，擬陽性が発生する確率 q は以下である．

$$q = p^k. \tag{2}$$

式 (1) と 式 (2) より，以下の式を導くことができる．

$$n \approx -\frac{m}{\ln q} \ln p \ln (1-p) \tag{3}$$

したがって， q や m に関わらず，$p = 1/2$ の時 n が極大となる．

ビット配列の長さ m と要素数 n が与えられたとき，擬陽性の確率が最小となるハッシュ関数の数 k は，$p = 1/2$ を 式 (1) に代入して，以下のように得ることができる．

$$k \approx \frac{m}{n} \ln 2. \tag{4}$$

偽陽性確率 q と要素数 n が与えられたとき，最短のビット配列の長さ m は，式 (2) と 式 (4) より次のように決定できる．

$$m = -\frac{n \ln q}{(\ln 2)^2} \tag{5}$$

従来の Bloom フィルタを使った素集合判定の手法として，最も素朴な手法は論理積後のビット列すべてが 0 であるかどうかである．また，ほぼ精度に違いはないが，ハッシュ関数を k 個使うことを利用した判定法が考えられる．その手法は，論理積後のビット列において，1 であるビットの数が k 以上であるかどうかである．k より少ない場合は，確実に共通の元が一つも存在しないことを保証できる．

3. Bloom フィルタの拡張

互いに素な集合であるかどうか判定できるように，次の三つの Bloom フィルタに対する拡張を提案する．

A) 1 であるビットの比率を下げる．
B) 各関数の出力範囲が重複しないようにする．
C) 他の関数の結果を入力する関数を導入する．

A) については自明ではあるが，従来はビットが 1 である確率が 1/2 では高すぎるため，関数の数を減らすことで，その確率を下げる．それによって論理積後のビット配列の 1 である確率を低下できる．従来は $p = 1/2$ で最適になるよう，擬陽性確率 q と要素数 n を入力とし，関数数 k やビット長 m を決定していたが，提案手法では，擬陽性確率 q，要素数 n に加え，関数数 k も与えられるよう拡張する．q および n，k が与えられビット長 m を求める式は式 (1) と式 (2) より次のように得られる．

$$m' = -\frac{kn}{\ln(1 - q^{\frac{1}{k}})} \tag{6}$$

また，論理積を行うビット配列の長さが異なる場合，二つの手法が考えられる．一つ目は，演算時に両者のビット配列を最小公倍数になるよう両ビット列を延長する手法である．二つ目は，あらかじめ両ビット配列の長さを

$$m'' = 2^{\lfloor \log_2 m' + 0.5 \rfloor} \tag{7}$$

を満足するように縮小あるいは延長するよう初期化しておいて，要素を追加した後，演算時に短いビット配列のみを長い方に一致するように延長する手法である．いずれの延長方法も，関数の実装方法に確認するが，一般的なハッシュ値の剰余を用いた関数であれば，ビット列を繰り返して延長する．

B) については，式 (6) によって決定した長さ m のビット配列を k 個の長さ m/k の部分ビット配列に分割し，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 $(0 \le i \le k-1)$.

[1] $[a, b)$ は $\{x | a \le x < b\}$ を意味する左閉右開区間である．

C) について，従来の Bloom フィルタではハッシュ関数のみを用いていたが，提案手法では複数の他の関数の結果を入力とする関数を導入する．具体的には，関数 f_a と f_b が従来のハッシュ関数であると仮定すると，例えば $mod(f_a(x)+f_b(x), m/k)$ や $mod(f_a(x)\times f_b(x), m/k)$ などがある[2]．本論文では，このような関数を**共起検査関数**と呼ぶ．

共起とは，ある Bloom フィルタにおいて，任意の二つのビットが一つの要素によって同時に 1 に設定されることを言う．例えば，要素 x を追加したとき，ハッシュ関数 f_a と f_b に入力した結果，$i = f_a(x)$ 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言う．

共起検査関数は，従来のハッシュ関数と同様にある要素のメンバ判定の役割以外に，ある二つのビット位置が共起しているかどうかを検査するための関数としての役割も持っている．二つのビット位置 i と j を入力したとき，その結果の位置のビットが 1 であれば i と j は共起している可能性があるが，ビットが 0 であれば共起していないことを保証できる．

関数は，他の関数との間に相関がない関数でなければ，Bloom フィルタとしての機能が失なわれる．相関するとは，二つの関数に同じ要素を入力したとき，それらの結果に何らかの関係があると予想できることを言う．例えば，$mod(f_a(x)+f_b(x), m/k)$ と $mod(f_a(x)+f_b(x)+1, m/k)$ は，常に隣合う二つビットを出力する強く相関した関数である．一方，$hash(x)$ と $hash(x+1)$ はハッシュ関数を介するため相関はない．

これら三つの拡張を施した Bloom フィルタを用いることで，従来の Bloom フィルタではほぼ実現できなかった素集合判定を行なうことができるようになる．例えば，関数の数が 4 で，ビット長が 1024 であるとき，関数 f_a は $[0, 256)$，f_b は $[256, 512)$ の範囲を出力するハッシュ関数で，他の関数は $f_c = mod(f_a(x)+f_b(x), m/k)$ と $f_d = mod(f_a(x)\times f_b(x), m/k)$ であると仮定する．部分ビット列 $[0, 256)$ のうち，1 であるビットの位置の集合 $\{a|array[a] = 1, 0 \le a < 256\}$ と， $[256, 512)$ のうち，1 であるビットの位置の集合 $\{b|array[b] = 1, 256 \le b < 512\}$ を求める．これらの集合に対し，各組合せで和と積が示す位置のビット $array[mod(a_i + b_j, m/k)]$ ， $array[mod(a_i \times b_j, m/k)]$ を評価する．もし，任意の組合せで和と積の両方が 1 であれば，共通集合が存在する可能性があることを示し，すべての組合せで和と積の少なくとも一つが 0 であれば共通集合が一切存在しないことを意味する．なお素集合判定でも偽陽性による誤検出はあるが，偽陰性はないすなわち，共起検査関数は従来のハッシュ関数としての役割だけでなく，任意の 2 ビットが同じ要素によって立てられたかどうかを検査するために利用する．

4. 実験による性能評価

本論文では，実験を通して提案手法の性能を評価する．実験は次の 3 種類を行った．

1. Bloom フィルタとしての性能
2. 素集合判定処理時の擬陽性による誤検出発生率
3. 素集合判定処理の時間

実験で用いた Bloom フィルタは次の 14 種類である．ハッシュ関数は MD5 を用い，複数のハッシュ関数の場合，i 番目のハッシュ関数は入力要素に $i-1$ を加えたものを入力する．

HK*1　従来の Bloom フィルタで，関数数 k とビット長 m は式 (4) と 式 (5) によってそれぞれ決定する．素集合判定は論理積後ビット列に 1 であるビットが k より少ないかどうかで行なう．

H1*1　ハッシュ関数が一つで，m は 式 (6) によって決定する．以降すべて，m は 式 (6) で決定する．

H2*1　2 個のハッシュ関数を用い，出力範囲は分割しない．

H4*1　4 個のハッシュ関数を用い，出力範囲は分割しない．

H1*2　2 個のハッシュ関数を用い，出力範囲は $m/2$ の長さに 2 分割する．

H1*4　4 個のハッシュ関数を用い，出力範囲は $m/4$ の長さに 4 分割する．

H2*1_F1*1　2 個のハッシュ関数と，それらの和を出力する共起検査関数を用い，出力範囲は $2m/3$ と $m/3$ の長さに 2 分割し，前者をハッシュ関数，後者を共起検査関数の出力範囲とする．

H1*2_F1*1　2 個のハッシュ関数と，それらの和を出力する共起検査関数を用い，出力範囲は $m/3$ の長さに 3 等分し，各関数が各部分ビット列を担当する．

H1*2_F1*2　$m/4$ の長さに 4 等分する．2 個のハッシュ関数とそれを入力とする和と積の 2 個の共起検査関数を用いる．すなわち，前節の最後に示した例と同じである．

H1*4_F1*2　4 個のハッシュ関数と 2 個の共起検査関数を用い，$m/6$ の長さに 6 等分する．共起検査関数は総和と総積である．

H1*2_F1*4　2 個のハッシュ関数と 4 個の共起検査関数を用い，$m/6$ の長さに 6 等分する．共起検査関数は和と積，差，商である．

H1*2_F1*6　$m/8$ の長さに 8 等分する．2 個のハッシュ関数と，和，積，差，商，とビット論理和，ビット論理積の 6 個の共起検査関数を用いる．

H1*2_F1*8　$m/10$ の長さに 10 等分する．2 個のハッシュ関数と，和，積，差，商，とビット論理和，ビット論理積，ビット排他的論理和，ビットシフトの 8 個の共起検査関数を用いる．

H1*2_F1*10　$m/12$ の長さに 12 等分する．2 個のハッシュ関数と，和，積，差，商，とビット論理和，ビット論理積，ビット排他的論理和，ビットシフト，及びハッシュ値を更にハッシュした値の和と積の 10 個の共起検査関数を用いる．

4.1 Bloom フィルタとしての性能

提案手法は，互いに素であることを特定できるよう Bloom フィルタを拡張したものである．しかしながら，そもそもの

[2] mod 関数は剰余演算を意味する．

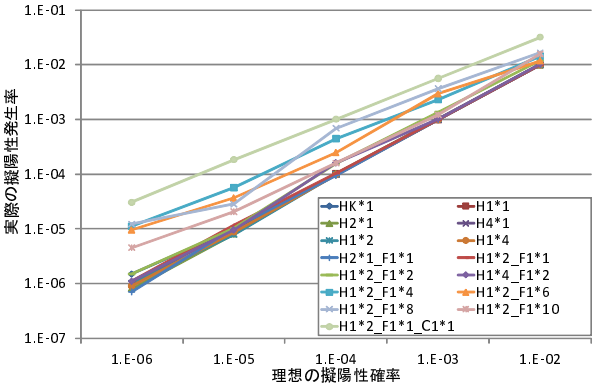

図 1　拡張 Bloom フィルタにおける理想の擬陽性確率に対する実際の偽陽性発生率

Fig. 1　False positive probability of extended Bloom filters.

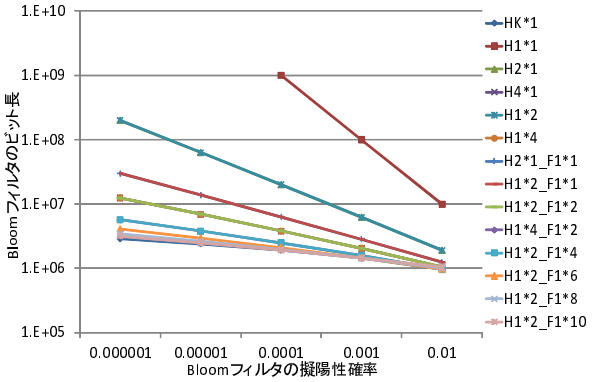

図 2　偽陽性確率に対する拡張 Bloom フィルタのビット長

Fig. 2　Bit length of extended Bloom filters.

Bloom フィルタとしての機能，すなわちある要素が集合のメンバであるかどうか評価できる機能が低下しては本末転倒である．したがって，本節では Bloom フィルタとしての機能が低下するかどうかを評価するために，まず擬陽性発生率を実験によって求めた．実験は，空の Bloom フィルタに集合 S を追加した後，S の元でない要素 $x \notin S$ がその Bloom フィルタに存在しているかどうかを評価する．要素 x は集合 S の元でないため，もし存在すると判定した場合は擬陽性による誤検出が発生したことを意味する．このテストを何度か繰り返して，擬陽性発生率を求める．集合 S は一様乱数によって生成する．

図 1 は Bloom フィルタの擬陽性発生率を示したグラフである．横軸は Bloom フィルタを初期化する際に与えた理想の擬陽性確率で，縦軸は実際に計測した擬陽性発生率である．したがって，横軸の値から外れると性能が低下していることになる．なお，実験は Bloom フィルタへの追加要素数は $100,000$ で，評価回数は各 $10,000,000$ 回である．すなわち，10^{-6} の場合に許容される擬陽性発生回数は 10 回である．

また，共起検査関数が他の関数と相関がある場合に偽陽性発生率がどうなるかを評価するため，本実験に限り，H1*2_F1*1_C1*1 を追加した．H1*2_F1*1_C1*1 は，関数の出力範囲分割は 4 で，二つのハッシュ関数と二つの共起検査関数で構成され，共起検査関数として，ハッシュ関数の和 $mod(f_a(x)+f_b(x), m/k)$ と，それに明らかに相関する，単に 1 を加算した $mod(f_a(x)+f_b(x)+1, m/k)$ を用いた．

図 1 から判断できることは，H1*2_F1*1_C1*1 が終始，理想の偽陽性発生率から外れていることが確認できる．すなわち，共起検査関数にはできる限り，互いに相関がないものを利用しなければならないことが判る．その他は，H1*2_F1*4，H1*2_F1*6，H1*2_F1*8，H1*2_F1*10 が外れていることが確認できる．これらは完全に相関関係のない共起検査関数が用意できなかったためと考えられる[3]．また図からは分かりづらいが，H1*1 はビット長が integer の最大値からあふれたため，10^{-5} 以下は評価できなかった．その他のものは，理想値にほぼ沿っているため，提案手法の A) B) C) とも基本的に従来の利用に耐えうることが確認できる．ただし，関数の数が非常に小さい時にはビット列の長さが問題になる．また共起検査関数はより相関の無いものを採用するか 4 より小さくする必要がある．

次に Bloom フィルタのビットの長さを調べる．長すぎるとオーバーフローが発生したり，処理速度が低下したりするなど問題が発生するため，短い方がよいことは自明ではあるが，われわれは応用によってはある程度の長さは許容できると考えている．図 2 は擬陽性確率に対する Bloom フィルタのビット長を示した図である．追加要素数は $100,000$ とした．

この図から，いずれも Bloom フィルタの擬陽性確率に比例しているのがわかる．また，図 1 で述べたように，H1*1 は 10^{-5} 以下では桁あふれし，計測できていない．追加要素数が大きく，擬陽性確率が小さい時は，H1*1 は利用は難しいことがわかった．ビット長の増加の割合は，関数の数の累乗に比例しているように見える．すなわち，ビット列が長い場合，関数を一つ追加すると，長さは半分になる．ビット数の長さが問題になるような場合は，関数の数を一つでも増やすと，長さは大幅に短縮できる．

4.2　素集合判定の擬陽性発生率

素集合判定の擬陽性発生率は最も重要な指標で，この値を改善することが本研究の目的である．素集合判定の擬陽性発生率の評価実験は，二つの Bloom フィルタを用意し，それらに互いに素な集合 R と S を追加し，この Bloom フィルタ同士で素集合判定処理を行う．実際には共通集合が存在しないため，共通集合があると判定された場合は擬陽性による誤検出が発生したことになる．一定回繰り返して，擬陽性が発生した回数を求め，その比率を擬陽性発生率とする．論理積を行う二つの Bloom フィルタは，同じタイプの Bloom フィルタで，追加要素数，擬陽性確率も等しい値を用いたため，ビット配列の長さも等しい．

素集合判定の擬陽性発生率の評価は，2 種類の実験を行った．Bloom フィルタの擬陽性確率をパラメタとした場合と，Bloom フィルタに追加する要素数をパラメタとした場合の 2 種類であ

[3] 本論文では，われわれは完全に相関のない共起検査関数を用意できなかったが，探せば存在する可能性は十分ある．

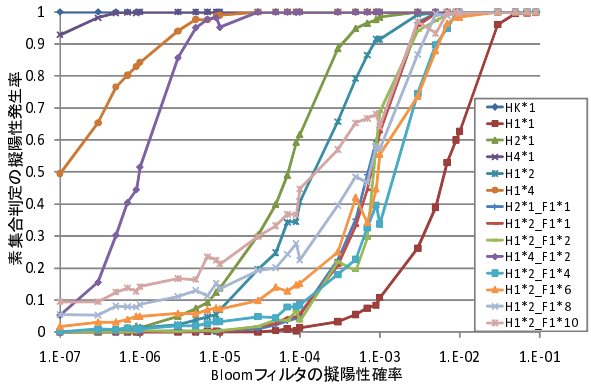

図 3　Bloom フィルタの擬陽性確率に対する素集合判定の擬陽性発生率

Fig. 3　False positive probability of intersections for that of Bloom filters.

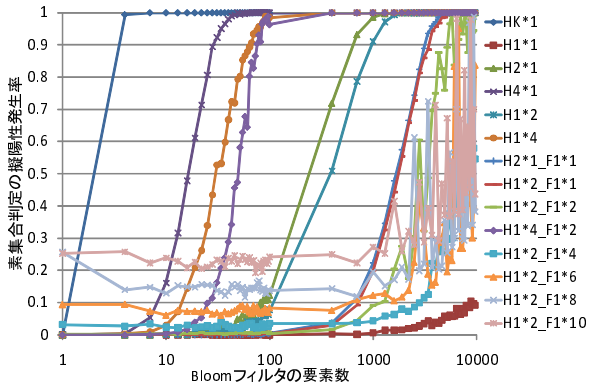

図 4　追加要素数に対する素集合判定の擬陽性発生率

Fig. 4　False positive probability of intersections for the number of inserted elements.

る．それぞれの実験結果を図 3 と図 4 に示す．いずれも縦軸は素集合判定の擬陽性発生率である．横軸は，図 3 が Bloom フィルタの擬陽性確率で，図 4 が Bloom フィルタへの追加要素数で，いずれも対数スケールである．すなわち，いずれも素集合判定の擬陽性発生率が 0 から離れるのが遅ければ遅いほど良いことを意味している．図 3 では要素数を 100 とし，図 4 では Bloom フィルタの擬陽性確率を 0.00001 として，実験を実施した．

図 3 から，H1*1 が最も優れており，HK*1 が最も劣っていることから，提案の A)「1 であるビットの 比率を下げる」方法が有用であることが確認できる．また，H4*1 より H1*4 が優れており，H2*1 より H1*2 が優れていることがわかる．このことから，提案の B)「各関数の出力範囲が重複しないようにする」方法が有効な手段であることが確認できる．また，H4*1 や H1*4 より H1*1_F1*2 が非常に優れていることから，提案の C)「他の関数の結果を入力する関数を導入する」方法も大変効果的であることが確認できた．

各 Bloom フィルタは，最も優れた手法が H1*1 ではあったが，H1*1 は 4.1 節での実験で Bloom フィルタ自体の性能が十分とは言えないことを確認している．次に共起検査関数を多く利用する H1*2_F1*4，H1*2_F1*6，H1*2_F1*8，H1*2_F1*10 は Bloom フィルタの擬陽性確率が低い場合でも，素集合判定の擬陽性発生率が 0 にならない．これは関数の数が多くなると 提案の A)「1 であるビットが比率を下げる」ことに反するためである．これらから現時点でも最も優れた拡張 Bloom フィルタは H2*1_F1*1，H1*2_F1*1，H1*2_F1*2 の三つが通常の Bloom フィルタとしても，素集合判定の擬陽性の面でも安定して優れた性能である．

図 4 でも同様に，提案の B)「各関数の出力範囲が重複しないようにする」方法や C)「他の関数の結果を入力する関数を導入する」方法が効果的であることが確認できる．また図 3 とほぼ同様の結果になっている．すなわち，最も優れた手法が H1*1 で，最も悪い手法が HK*1 であることや，H1*2_F1*4，H1*2_F1*6，H1*2_F1*8，H1*2_F1*10 は要素数が少ない場合でも素集合判定の擬陽性発生率が 0 にならないことは一致した結果である．異なった結果としては，図 4 では H1*2_F1*2 が H2*1_F1*1，H1*2_F1*1 より，やや優れていることを示した点が異なる．

なお，ラインがジグザグになっているのは，共起検査関数を多く用いた場合で，共起検査関数同士が相関しているために起こると考えられる．

4.3　素集合判定の処理時間

最後に，提案した拡張 Bloom フィルタで素集合判定処理を行った場合の処理時間を評価する．いくら擬陽性が発生しないとしても処理コストが非常に高いようであれば，その Bloom フィルタは実用的でないため，処理時間を評価する必要がある．

実験に用いた計算機は CPU が Intel Core2 Quad Q9450 (2.66 GHz)，メインメモリが 4 Gbytes, OS が Ubuntu Linux 10.04 (kernel 2.6.32) であった．実験は，Bloom フィルタはディスク上に構築し，素集合判定はオンメモリで行なった．

実験は 図 4 の実験の際に同時に素集合判定処理時間を計測し，平均処理時間 (ミリ秒) を求めた．その結果を図 5 に示す．横軸は追加要素数で，縦軸は素集合判定処理の 1 回に要する処理時間の平均である．

図 5 から H1*4 が最も高速で，それに次いで H4*1 が速い．このことから，精度の面だけでなく，速度の面でも，提案の B)「各関数の出力範囲が重複しないようにする」方法が有効な手段であることが確認できる．また，共起検査関数を多く用いた場合は処理性能が低下することも確認できる．H1*1 は処理内容は単純であるが，ビット長が長いため，その読み込みに時間を要すると考えられる．これまでの結果から最も実用的で優れた性能の H1*2_F1*2 や H2*1_F1*1，H1*2_F1*1 は，H1*2_F1*2 がやや遅いが，十分実用的な速度で処理できることがわかる．

これらの実験結果より，提案手法のうち，H1*2_F1*2，H2*1_F1*1，H1*2_F1*1 の 3 手法が最も優れた手法であることが確認できた．H2*1_F1*1 と H1*2_F1*1 の特性は，本実験ではほぼ同じであったため，さらなる詳細な実験を行なう必要がある．また H1*2_F1*2 は図 4 の要素数に対する疑陽性発生率に関しては，H2*1_F1*1 と H1*2_F1*1 より優れた結果をであった．一方処理時間では H2*1_F1*1 と H1*2_F1*1 の方が高速で

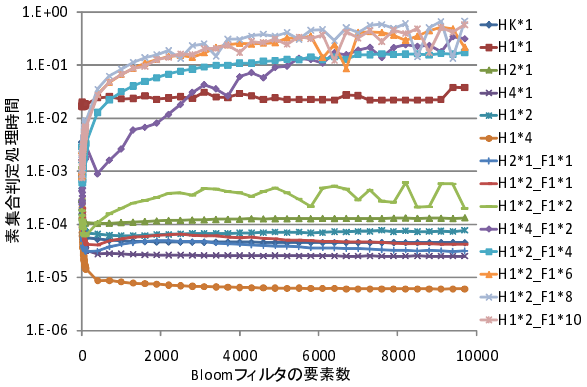

図 5　追加要素数に対する素集合判定の処理時間

Fig. 5　The processing times of a disjoint sets test for the number of inserted elements.

あるという結果となった．

また実装の面から考えると，H2*1_F1*1 と H1*2_F1*1 はビット列を 3 で割ることになる．そのため，式 (7) で示した，異なる長さのビット列で論理積を行うための方法を採用する場合に，3 で割る必要のある H2*1_F1*1 と H1*2_F1*1 より 4 で割ることができる H1*2_F1*2 の方が 2 進数である計算機と相性がよい実装である．

5.　まとめ

本論文では，Bloom フィルタを拡張し，異なる集合が互いに素であるかどうかを容易に判定できる三つの拡張手法を提案した．

A) 1 であるビットの比率を下げる
B) 各関数の出力範囲が重複しないようにする
C) 他の関数の結果を入力する共起検査関数を導入する

提案手法の性能を評価するため 3 種類の実験を行った．Bloom フィルタ自体の性能を評価と，素集合判定の精度評価，素集合判定の速度評価である．実験結果から，従来の Bloom フィルタでは不可能であったが，提案手法は素集合判定に非常に適した拡張であることを確認した．また， 2 個のハッシュ関数と 2 個の共起検査関数を用いた H1*2_F1*2 と 2 個のハッシュ関数と 1 個の共起検査関数を用いた H2*1_F1*1 と H1*2_F1*1 の三つが優れた性能で，速度面では H2*1_F1*1 と H1*2_F1*1 の方が優れ，精度面と実装面では H1*2_F1*2 が優れていることがわかった．

[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．

[文献]

[1] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[2] L. Fan, P. Cao, J. M. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, 2000.

[3] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In W. W. Chu, G. Gardarin, S. Ohsuga, and Y. Kambayashi eds., *VLDB*, pp. 149–159. Morgan Kaufmann, 1986.

[4] L. Michael, W. Nejdl, O. Papapetrou, and W. Siberski. Improving distributed join efficiency with extended bloom filter operations. In *AINA*, pp. 187–194. IEEE Computer Society, 2007.

[5] Oracle Corporation. Data warehousing on oracle rac best practices, October 2008. An Oracle White Paper, http://www.oracle.com/technology/products/database/clustering/pdf/bp_rac_dw.pdf.

[6] 的野, 小島. 分散 rdf 問合せ処理時の転送量減少のためのブルームフィルタの拡張. 情報処理学会論文誌　データベース（TOD）［電子情報通信学会データ工学研究専門委員会共同編集］, Vol.2(No.1):33–45, 3 月 2009 年.

的野 晃整　Akiyoshi MATONO
産業技術総合研究所情報技術研究部門研究員．2005 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，工学博士．RDF データベースの研究に従事．情報処理学会，ACM，日本データベース学会各会員．

スティーブン リンデン　Steven LYNDEN
産業技術総合研究所情報技術研究部門特別研究員．2004 カーディフ大学 (UK) にて Ph.D. 取得．マルチエージェントシステムおよび，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　Yusuke TANIMURA
産業技術総合研究所情報技術研究部門研究員． 2004 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，工学博士．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． USENIX，情報処理学会，各会員．

小島 功　Isao KOJIMA
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．情報処理学会，ACM，IEEE 各正会員． OASIS メンバ．OGF DAIS Working Group Co-Chair.