2019 级

《大数据存储与管理》课程

课 程 报 告

基于 BloomFilter 的设计

姓　名 周飞

学　号 U201915183

班　号 计算机 1908 班

日　期 2022.04.12

目　　录

一、基本介绍

Bloom Filter 是 1970 年由布隆提出的。它实际上是一个很长的二进制向量和一系列随机映射函数。布隆过滤器可以用于检索一个元素是否在一个集合中。它的优点是空间效率和查询时间都远远超过一般的算法，缺点是有一定的误识别率和删除困难。

如果想判断一个元素是不是在一个集合里，一般想到的是将集合中所有元素保存起来，然后通过比较确定。链表、树、散列表等等数据结构都是这种思路。但是随着集合中元素的增加，我们需要的存储空间越来越大。同时检索速度也越来越慢。

布隆过滤器的原理是，当一个元素被加入集合时，通过 K 个散列函数将这个元素映射成一个位数组中的 K 个点，把它们置为 1。检索时，我们只要看看这些点是不是都是 1 就（大约）知道集合中有没有它了：如果这些点有任何一个 0，则被检元素一定不在；如果都是 1，则被检元素很可能在。这就是布隆过滤器的基本思想

Bloom Filter 是一种空间效率很高的随机数据结构，它利用位数组很简洁地表示一个集合，并能判断一个元素是否属于这个集合。Bloom Filter 的这种高效是有一定代价的：在判断一个 元素是否属于某个集合时，有可能会把不属于 这个集合的元素误认为属于这个集合（false positive）。因此，Bloom Filter 不适合那些 “零错误”的应用场合。而在能容忍低错误率 的应用场合下，Bloom Filter 通过极少的错误换 取了存储空间的极大节省。

BloomFilter 主要用于数据的去重。在爬虫软件中为了不重复爬取相同的网页常常会用 BloomFilter 作为过滤器。以及垃圾邮件的判断、网盘存储都可能用到该项技术。

本次实验主要实现了两种不同的 BloomFilter，分别是标准的 BF 和可以处理高维数据的 HDBF。

二、原理和理论分析

2.1 原理

提出 BloomFilter 的初衷是因为使用单词 hash 进行数据去重的的 False Positive 率太高，因此想到了使用多个 hash 函数对每个数据进行判断的方法。因此提出了 BloomFilter。

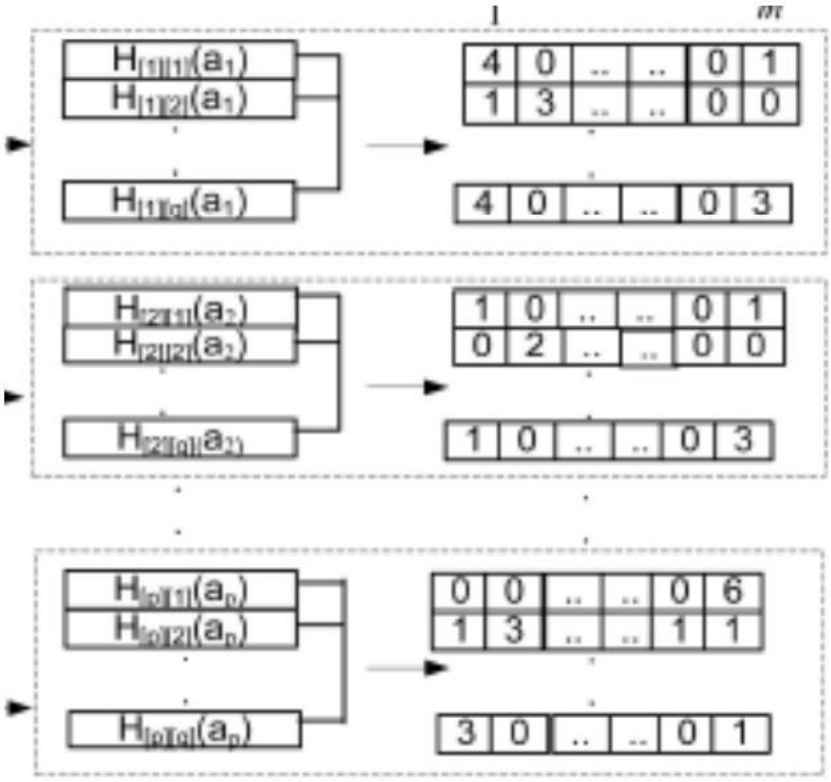

图 2.1 BF 基本示意图

BloomFilter 判断出错的条件比单个 hash 的出错条件更加苛刻。需要所有的 hash 值都冲突，因此更加可靠。

尽管采用了多个 hash 共同判断，但仍保留了出错的可能性，因此大多时候泛用与对准确率的要求并非绝对的高的情况。

主要参数：

m: BloomFilter 中的 bit 位个数或者 CBF 中计数器的个数

n: add 的元素个数

FPP： False positive probability。 误判率

k: 哈希函数的个数。

基本操作：

1 . Add 操作： 向 BloomFilter 保存的数据集合中添加数据。将每个 hash 值对应的 bitset 中的值置 1。 如图 2.2

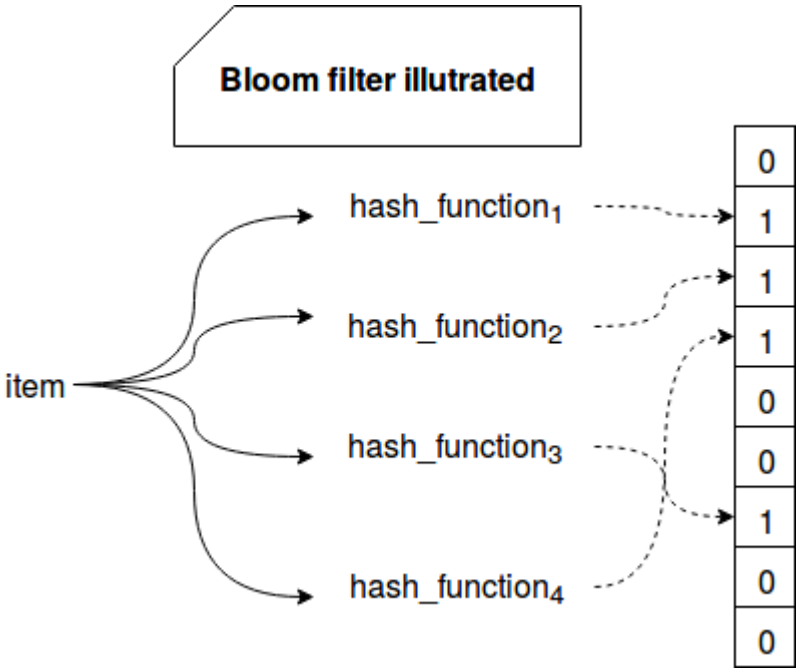

图 2.2 BF ADD 示意图

2 . Check 操作： 查询某项是否在 BlommFilter 中。检查每个 hash 值对应的 bit 位，如果其中有一位为 0 则表示不在 set 中、否则表示在 set 中。

3 . Delete 操作： 删除集合中的某项。

2.2 理论分析

向 Bloom Filter 插入一个元素时，其一个 Hash Function 会将 BitArray 中的某 Bit 置为 1，故对于任一 Bit 而言为 0 的概率：

$$1-\frac{1}{m}.$$

插入一个元素时，其 k 个 Hash Function 都未将该 Bit 置为 1 的概率：

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

Bloom Filter 插入全部 n 个元素后，该 Bit 依然为 0 的概率即为

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

该 Bit 为 1 的概率则为

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

可以得到误判率 FPP=

$$\left(1-\left[1-\frac{1}{m}\right]^{kn}\right)^{k} \approx \left(1-e^{-kn/m}\right)^{k}.$$

将 FPP 对 k 求导，可以算出当 FPP 最小时

$$k=\frac{m}{n}ln2$$

代换后

$$m=-\frac{nlnP(true)}{(ln2)^2}$$

因此，在事先预估了 n 值和 FPP 后可以计算出准确率最高的情况下 k=ln2*m/2。M=-nlnFPP/$(ln2)^2$ 。

三、BloomFilter 的变体

3.1 Counting Bloom Filter。

Counting Bloom Filter 在 Standard Bloom Filter 的基础上增加了计数统计和删除的功能。 CBF 的 bits 不再是每个 hash 值对应一个 bit 大小而是对应 4 个字节大小的一个计数器。 每次添加一个元素都会给对应哈希位置的计数器加 1。 查询时则判断是否有不大于 0 的计数器，如果有表示不在集合里面。 使得原本不支持删除的 BF 在支持删除操作后仍能保持较好的准确率。 与 BF 的区别图见图 3.1

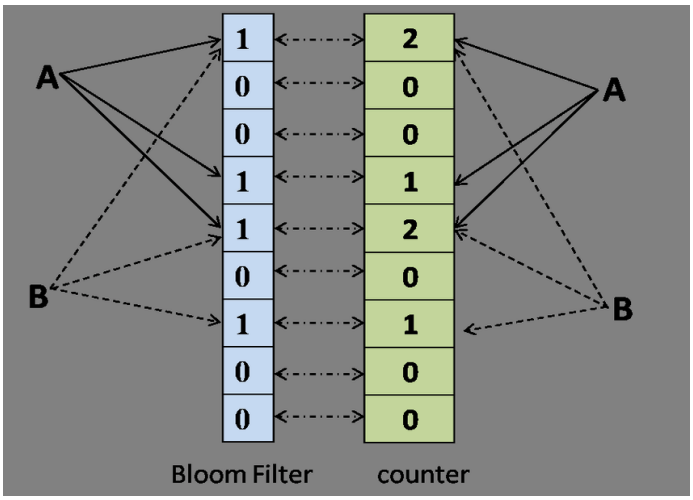

图 3.1 BF AND CBF

3.3 High Dimensional Bloom Filter

HDBF 和 BF 的主要不同是 hash 函数的工作方式。 对于大多 BF 而言，一般采用的哈希函数如 MurMur 都是对字符串进行哈希。其做法需要遍历字符串的每一字节并对其处理。这就导致了在哈希很长的字符串或者是非字符串类型数据如（图片、音乐）等二进制文件时速度过慢。HDBF 采用的是对处理的是整数向量，是每次对数据中一个 4 字节的整数进行处理。 因此 HDBF 可以对很长的数据进行哈希，不论是字符串还是二进制，在处理高维向量（一条数据由很多 4 个字节的整数组成，个数被称为维度）时耗时更少。

示意图如图 3.2

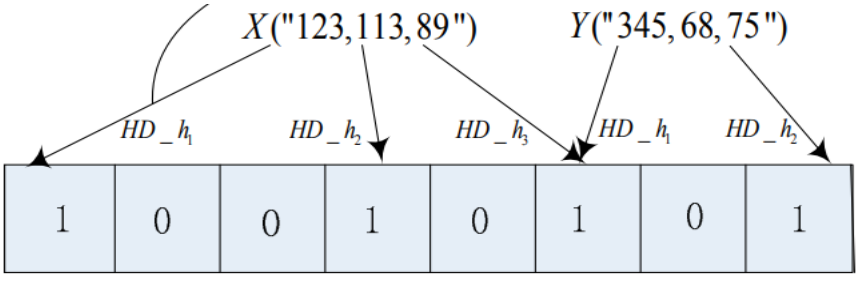

图 3.2 HDBF 示意图

四、实验设计

此次设计主要使用 C++语言完成了标准 BF 和 HDBF。

由于 C++的 bitset 类模板需要在编译时确定位的数量，因此使用 char 类型的 vector 来实现 bits。取某个 bit 的时候需要使用或位运算，赋值时将或运算后的结果赋值给 char。

BF 和 HDBF 类的定义：

```
class BloomFilter{
private:
    int m ; //the length of bits
    int cnt;// the True number in  bits
    int n; // the number expected to add
    int k;// the number of hash functions
    float fp; //expected false positive rate
    std::vector<char>bits;
public:
    BloomFilter(int n, double fp);
    bool add (const char * str,int len);
    bool check(const char * str , int len );
    void clear();

    //inline memfnc
    int size(){return m;}
    int right(){return cnt;}
};

class HighDimBloomFilter{
    int m ; //the length of bits
    int cnt;// the True number in  bits
    int n; // the number expected to add
    int k;// the number of hash functions
    float fp; //expected false positive rate
    int sz;
    std::vector<char>bits;
public:
    HighDimBloomFilter(int n, double fp);
    bool add (int * vec,int dim);
    bool check(int *vec , int dim );
    void clear();

    //inline memfnc
    int size(){return m;}
    int right(){return cnt;}
};
```

图 4 . 1　BF 和 HDBF　类

哈希函数的选取：

BF 使用 HDSax 哈希函数、HDBF 使用 HDSax_hash 函数。

```
static unsigned int HDsax_hash(int *key,int dim ){
    unsigned int h=0;
    while(--dim)
        h ^= (h<<5) + (h>>2) + (unsigned int)*key++;
    return (h& 0x7FFFFFFF);
}

unsigned int sax_hash(const char *key,int len){
    unsigned int h = 0;
    while(--len)
        h^= (h << 5) + (h >> 2) + (unsigned char)*key++;
    return (h& 0x7FFFFFFF);
}
```

图 4.2 hash 函数的选取

BF 主要方法 add 和 check 的实现：

```
bool BloomFilter::add (const char * str,int len){
    bool added=false;
    for(int i=0;i!=k;++i){
        unsigned int haskey;
        haskey=sax_hash(str,len);
        haskey%=m;
        unsigned int index=haskey/8;
        unsigned int offset=haskey%8;
        char mask = 1 << offset;
        if ((bits[index] | mask)!=bits[index]){
            bits[index] |= mask;
            added=true;
            ++cnt;
        }
    }
    return added;
}
bool BloomFilter::check(const char * str , int len ){
    for(int i=0;i!=k;++i){
        unsigned int haskey;
        haskey=sax_hash(str,len);
        haskey%=m;
        int index=haskey/8;
        int offset = haskey%8;
        char mask = 1 << offset;
        if((bits[index] | mask)!=bits[index]){
            return false;
        }
    }
    return true;
}
```

图 4.3 BF 类成员函数 ADD 和 check 的实现

HDBF 主要方法 add 和 check 的实现

```
bool HighDimBloomFilter::add (int *vec,int dim){
    bool added=false;
    for(int i=0;i!=k;++i){
        unsigned int haskey;
        haskey=HDsax_hash(vec,dim);
        haskey%=m;
        unsigned int index=haskey/8;
        unsigned int offset=haskey%8;
        char mask = 1 << offset;
        if ((bits[index] | mask)!=bits[index]){
            bits[index] |= mask;
            added=true;
            ++cnt;
        }
    }
    return added;
}
bool HighDimBloomFilter::check(int* vec , int dim){
    for(int i=0;i!=k;++i){
        unsigned haskey;
        haskey=HDsax_hash(vec,dim);
        haskey%=m;
        int index=haskey/8;
        int offset = haskey%8;
        char mask = 1 << offset;
        if((bits[index] | mask)!=bits[index]){
            return false;
        }
    }
    return true;
}
```

图 4.4 HDBF add 和 check 的实现

五、性能测试和改善

5.1 数据集的选取

数据集使用 sift-small 数据集，可以从网站 Evaluation of Approximate nearest neighbors: large datasets (irisa.fr) 获取。

Sift-small 中每个数据的维度为 128，即 128*4=512 字节。 使用其中的 base10000 条数据作为 n 值 add 到 set 中， 使用 learn 中的 25000 条数据作为测试。

5.2 内存占用

理论分析：
由于 BF 和 HDBF 都使用了位串来保存哈希信息，其空间复杂度位 O(m)。在加入数据条数 n 为 10 亿,误判率 fpp=0.01 的情况下，m=958500000。占有的空间也只有 114MB。 因此内存占用应该是很小的。

实际测试：
n=10000
fpp=0.01
BF 和 HDBF 内存占用如图 5.1、5.2

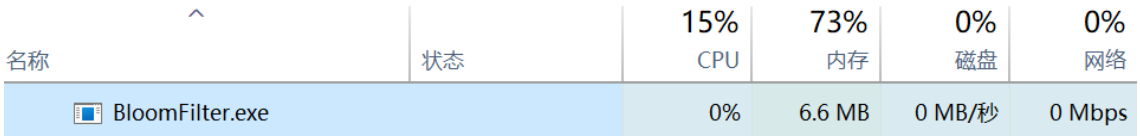

图 5.1 BF 内存占用情况

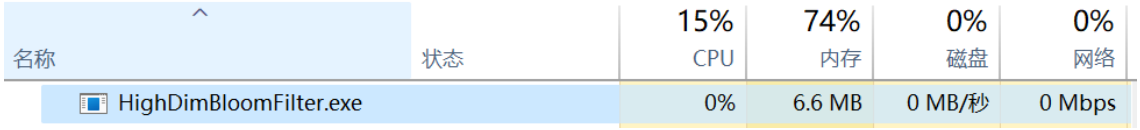

图 5.2 HDBF 内存占用情况

5.3 误判率 FPP

测前猜测：
BF 使用的哈希函数每次取一个字节进行处理，相当于对字符串的处理。HDBF 每次取一个 4 字节整数进行处理， 相当于对整数向量的处理。 BF 处理次数更多，运算更复杂，虽然耗时更久，但或许误判率更低。

实际测试：

预期 FPP0 取 0.01 不变。 更改 n 的大小比较二者的实际的 FPP 如图 5.3

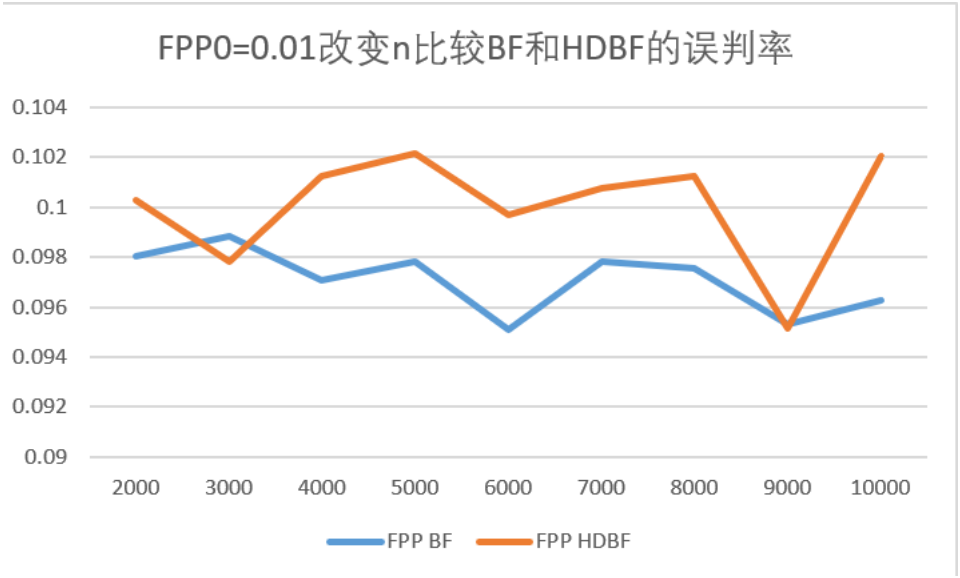

图 5.3

n 的大小不变，更改预期的 FPP0 值比较二者实际的 FPP 值如图 5.4

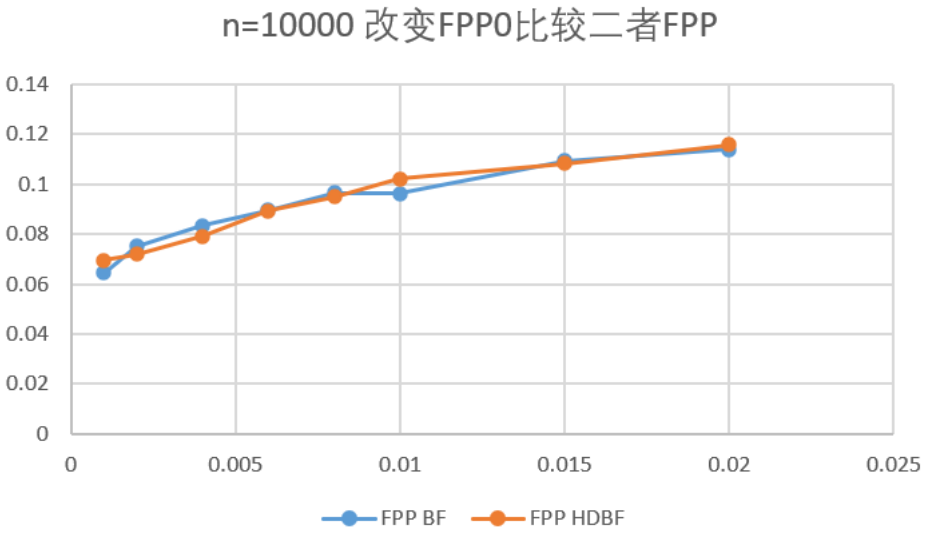

图 5.4

可以看出，两者 FPP 差距不大，BF 相对于 HDBF 误判率稍低，符合测前预计。

5.4 查询延迟

在前面已经分析过，由于 HDBF 采用对整形向量进行哈希，速度相较于 BF 应当更快，特别是在处理长数据时。

N=10000, FPP0=0.01

改变查询次数 Cnt，比较二者的查询时延如图 5.5

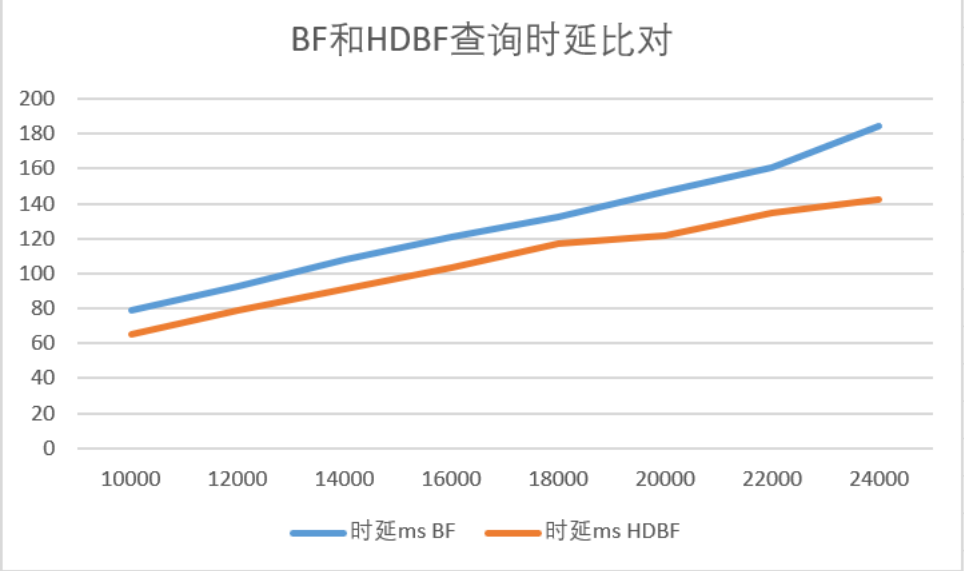

图 5.5

可见 HDBF 在查询时延的减少上确实有效。 本次测试使用的数据维度为 128，对于维度更高的数据，HDBF 在查询时间上的缩短应该会更加明显。 因此在处理二进制和其他大型数据时，使用 HDBF 在保证和 BF 相当正确率的情况下能带来更好的效率。

六、实验总结

通过对 BF 和 HDBF 的实现和比较，得出 HDBF 确实能够明显减少查询延迟，特别是在处理长数据、文件等方面很有作用。

在测试中测试出的误判率 FPP 与预设预判率 FPP0 有一定差距，在改用 MurMur3 作为 hash 函数后得到的实际误判率和预设预判率几乎一致。 但是 MurMur3 难以修改为处理整形向量的版本，为了更好地对比 BF 和 HDBF，故而选用了 SAH 进行改变作为 hash 函数。

BloomFilter 的不同变体适用于不同的场景，大部分在在网络爬虫，垃圾邮件清理，云存储方面有很大作用。它是一种比普通 hash 效率更高的手段，以后在写网络爬虫的时候就可以使用 BF 而不再是低效地使用集合，在平时 coding 过程中也是一项好用的工具。

参考文献

- A Bloom Filter for High Dimensional Vectors Chunyan Shuai 1 , Hengcheng Yang 2 , Xin Ouyang 3,* and Zeweiyi Gong 2
- https://en.wikipedia.org/wiki/Bloom_filter
- F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, “Beyond B loom Filters: From Approximate Membership Checks to Approximate State Machine s,” Proc. ACM SIGCOMM, 2006.
- Y. Zhu and H. Jiang, “False Rate Analysis of Bloom Filter Replicas in Distributed Systems,” Proc. Int’l Conf. Parallel Processing (ICPP ’06), pp. 255-262, 2006.