2018 级

《物联网数据存储与管理》课程

实 验 报 告

姓　名　库尔夏提·亚森

学　号　U201714621

班　号　计算机1804班

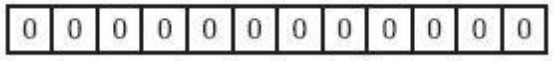
日　期　2021.06.28

Bloom Filter 概念和原理

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。

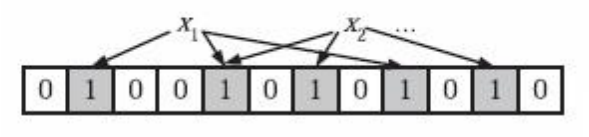
集合表示和元素查询

Bloom Filter是如何用位数组表示集合的。初始状态时，Bloom Filter是一个包含m位的位数组，每一位都置为0。



为了表达S={x1, x2,… ,xn}这样一个n个元素的集合，Bloom Filter使用k个相互独立的哈希函数（Hash Function），它们分别将集合中的每个元素映射到{1,… ,m}的范围中。对任意一个元素x，第i个哈希函数映射的位置hi(x)就会被置为1（1≤ i≤ k）。注意，如果一个位置多次被置为1，那么只有第一次会起作用，后面几次将没有任何效果。

在下图中,k=3,且有两个哈希函数选中同一个位置(从左边数第五位)。



在判断y是否属于这个集合时,对y应用k次哈希函数,如果所有$h_i(y)$的位置都是1（$1 \leq i \leq k$），那么就认为y是集合中的元素，否则就认为y不是集合中的元素。下图中y_1就不是集合中的元素。y_2或者属于这个集合，或者刚好是一个*false positive*。

错误率估计

Bloom Filter在判断一个元素是否属于它表示的集合时会有一定的错误率（false positive rate），下面来估计错误率的大小。在估计之前为了简化模型,假设kn<m且各个哈希函数是完全随机的。当集合S={x1, x2,… ,xn}的所有元素都被k个哈希函数映射到m位的位数组中时，这个位数组中某一位还是0的概率是：

$$p' = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}.$$

其中1/m表示任意一个哈希函数选中这一位的概率(前提是哈希函数是完全随机的)，(1- 1/m)表示哈希一次没有选中这一位的概率。要把S完全映射到位数组中，需要做kn次哈希。某一位还是0意味着kn次

哈希都没有选中它，因此这个概率就是（1- 1/m）的kn次方。令p = e- kn/m是为了简化运算，这里用到了计算e时常用的近似：

$$\lim_{x\to\infty}\left(1-\frac{1}{x}\right)^{-x}=\mathrm{e}$$

令ρ为位数组中*0*的比例，则ρ的数学期望*E(ρ)= p'*。在ρ已知的情况下，要求的错误率（*false positive rate*）为：

$$(1-\rho)^k \approx (1-p')^k \approx (1-p)^k.$$

(1- ρ)为位数组中1的比例，(1- ρ)k就表示k次哈希都刚好选中1的区域，即false positive rate。上式中第二步近似在前面已经提到了，现在来看第一步近似。p' 只是ρ的数学期望，在实际中ρ的值有可能偏离它的数学期望值。M. Mitzenmacher已经证明[2] ，位数组中0的比例非常集中地分布在它的数学期望值的附近。因此，第一步的近似得以成立。分别将p和p' 代入上式中，得：

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

$$f = \left(1-\mathrm{e}^{-kn/m}\right)^k = (1-p)^k.$$

相比*p'*和*f'*，使用*p*和*f*通常在分析中更为方便。

最优的哈希函数个数

Bloom Filter要靠多个哈希函数将集合映射到位数组中，那么应该选择几个哈希函数才能使元素查询时的错误率降到最低呢？这里

有两个互斥的理由：如果哈希函数的个数多，那么在对一个不属于集合的元素进行查询时得到0的概率就大；但另一方面，如果哈希函数的个数少，那么位数组中的0就多。为了得到最优的哈希函数个数，我们需要根据上一小节中的错误率公式进行计算。

先用p和f进行计算。注意到f = exp(k ln(1 − e−kn/m))，我们令g = k ln(1 − e−kn/m)，只要让g取到最小，f自然也取到最小。由于p = e- kn/m，可以将g写成

$$g = -\frac{m}{n}\ln(p)\ln(1-p),$$

根据对称性法则可以很容易看出当p = 1/2，也就是k = ln2· (m/n)时，g取得最小值。在这种情况下，最小错误率f等于(1/2)k ≈ (0.6185)m/n。另外，注意到p是位数组中某一位仍是0的概率，所以p = 1/2对应着位数组中0和1各一半。换句话说，要想保持错误率低，最好让位数组有一半还空着。

需要强调的一点是，p = 1/2时错误率最小这个结果并不依赖于近似值p和f。同样对于f′ = exp(k ln(1 − (1 − 1/m)kn))，g′ = k ln(1 − (1 − 1/m)kn)，p′ = (1 − 1/m)kn，我们可以将g′ 写成

$$g' = \frac{1}{n\ln(1-1/m)}\ln(p')\ln(1-p'),$$

同样根据对称性法则可以得到当*p′ = 1/2*时，*g′*取得最小值。

位数组的大小

在不超过一定错误率的情况下，Bloom Filter至少需要多少位才

能表示全集中任意n个元素的集合。假设全集中共有u个元素，允许的最大错误率为ϵ，下面我们来求位数组的位数m。

假设X为全集中任取n个元素的集合，F(X)是表示X的位数组。那么对于集合X中任意一个元素x，在s = F(X)中查询x都能得到肯定的结果，即s能够接受x。显然，由于Bloom Filter引入了错误，s能够接受的不仅仅是X中的元素，它还能够ϵ（u - n)个false positive。因此，对于一个确定的位数组来说，它能够接受总共n + ϵ（u - n)个元素。在n + ϵ（u - n)个元素中，s真正表示的只有其中n个，所以一个确定的位数组可以表示

$$\binom{n+\epsilon(u-n)}{n}$$

个集合。*m*位的位数组共有2^m个不同的组合，进而可以推出，*m*位的位数组可以表示

$$2^m\binom{n+\epsilon(u-n)}{n}$$

个集合。全集中*n*个元素的集合总共有

$$\binom{u}{n}$$

个，因此要让m位的位数组能够表示所有n个元素的集合，必须有

$$2^m\binom{n+\epsilon(u-n)}{n} \geq \binom{u}{n}$$

即：

$$m \geq \log_2 \frac{\binom{u}{n}}{\binom{n+\epsilon(u-n)}{n}} \approx \log_2 \frac{\binom{u}{n}}{\binom{\epsilon u}{n}} \geq \log_2 \epsilon^{-n} = n\log_2(1/\epsilon).$$

上式中的近似前提是*n*和*ϵu*相比很小，这也是实际情况中常常发生

的。根据上式，得出结论：在错误率不大于ϵ的情况下，m至少要等于$n\ log_2(1/\epsilon)$才能表示任意n个元素的集合。

上一小节中我们曾算出当$k = ln2 \cdot (m/n)$时错误率f最小，这时$f = (1/2)^k = (1/2)^{mln2/n}$。现在令$f \leq \epsilon$，可以推出

$$m \geq n\frac{\log_2(1/\epsilon)}{\ln 2} = n \log_2 \mathrm{e} \cdot \log_2(1/\epsilon).$$

这个结果比前面算得的下界$n\ log_2(1/\epsilon)$大了$log_2 e \approx 1.44$倍。这说明在哈希函数的个数取到最优时，要让错误率不超过ϵ，m至少需要取到最小值的1.44倍。

实验测试

实验设计

实验参数符号、含义、配置：

m 哈希数组基础长度 10^5~10^6 步进10^5 (5000 一次单独测试)

fp 可容许的最大误判率 A 组限制为 0.01 B 组限制为 0.001

n 插入元素个数 105

t 查找元素个数 10^4

k 哈希函数个数 5

流程设计

考虑对比分析：基础 Bloom Filter 与Scalable Bloom Filter 在大规模、饱和元素插入条件下的误判率（false positive）。

具体流程如下：

对某一梯度的哈希数组长度 m，以 m 分别初始化一个 Bloom Filter 结构与一个 Scalable Bloom Filter 结构。

对BF 与SBF 结构，分别插入相同数量（n）的元素；其中对于 SBF，当其实际误判率达到预设最大误判率 fp，会自动扩容。

对BF 与SBF 结构，分别查找相同数量（t）的元素，统计查找过程中的误判数 error，计算误判率 fp1、fp2。

基础 Bloom Filter 类代码如下：

```
#include
<vector> using
namespace std;
class
BloomFilter{ priv
ate:
    vector<bool>
    bits; int len;
    static int hash1(int v, int m) {
    }
    static int hash2(int v, int m) {
        return (((v >> 7) ^ (v << 11)) & 0x7fffffff) % m;
    }
    static int hash3(int v, int m) {
        return (((v >> 13) ^ (v << 17)) & 0x7fffffff) % m;
    }
    static int hash4(int v, int m) {
```

```
        return (((v >> 19) ^ (v << 23)) & 0x7fffffff) % m;
    }
    static int hash5(int v, int m) {
        return (((v >> 29) ^ (v << 2)) & 0x7fffffff) % m;
    }

public:
    BloomFilter(int len=200000) :
        len(len){ bits.resize(len);
    }
    void insert(int v){
        bits[hash1(v, len)] = bits[hash2(v, len)] = bits[hash3(v, len)] =
bits[hash4(v, len)] = bits[hash5(v, len)] = true;
    }
    bool find(int v){
        return bits[hash1(v, len)] & bits[hash2(v, len)] & bits[hash3(v, len)] &
bits[hash4(v, len)] & bits[hash5(v, len)];
    }
    int cap() const { return len; }
};
```

Scalable Bloom Filter 类代码如下：

```
#include
<vector>
#include
<cmath> using
namespace std;
class
ScalableBloomFilter{
private:
    vector<vector<bool>> bits;
    int depth; // 过滤器层数，从 0 开始
    int len; // 第一层容量，第 i 层容量为
    2^(i- 1)*len int num; //  当前层数据量
    double fp; // 可容许 false positive
```

```
    rate static int hash1(int v, int m) {
        return (((v >> 3) ^ (v << 5)) & 0x7fffffff) % m;
    int cap() const { return (len << (depth + 1))
    - len; } int resizeTime() const { return
    depth; }
};
    static int hash2(int v, int m) {
        return (((v >> 7) ^ (v << 11)) & 0x7fffffff) % m;
    }
    static int hash3(int v, int m) {
        return (((v >> 13) ^ (v << 17)) & 0x7fffffff) % m;
    }
    static int hash4(int v, int m) {
        return (((v >> 19) ^ (v << 23)) & 0x7fffffff) % m;
    }
    static int hash5(int v, int m) {
        return (((v >> 29) ^ (v << 2)) & 0x7fffffff) % m;
    }

public:
    ScalableBloomFilter(int len=200000, double fp=1e- 2) :
        len(len), fp(fp){ depth = num = 0;
        bits.resiz
        e(1);
        bits[0].re
        size(len);
    }
    void resize(){
        depth += 1;
        bits.resize(depth+1);
        bits[depth].resize(le
        n << depth); num = 0;
    }
    void insert(int v){
        if(num >= (len << depth) * log(1.0 / (1.0 - pow(fp, 0.2))) / 5) // n = -
        m * ln(1 - f^0.2)
```

```
/ 5
                    resize();
                int m = len << depth;
                bits[depth][hash1(v, m)] = bits[depth][hash2(v, m)] =
                bits[depth][hash3(v, m)] =
bits[depth][hash4(v, m)] = bits[depth][hash5(v,
                m)] = true; num++;
            }
            bool find(int v){
                for(int i=depth;
                    i>=0;
                    i- - ){ int m
                    = len << i;
                    bool ok = bits[i][hash1(v, m)] & bits[i][hash2(v, m)] &
                    bits[i][hash3(v, m)] &
bits[i][hash4(v, m)] & bits[i][hash5(v, m)];
                    if(ok)
                        return true;
                }
                return false;
            }
```

测试程序 test.cpp 代码如下：

```
#include <iostream>
#include <random>
#include
<unordered_set>
#include
"BloomFilter.cpp"
#include
"ScalableBloomFilter.cpp" using
namespace std;

const int SEED = 1024;
int m = 1000000; // bloom filter 哈希数组长度
int n = 100000; // 插入元素个数
```

```
int t = 10000; //  查找元素个数
double fp = 1e- 3; //  可容许 false positive rate


int main() {
    //随机创建插入元素集、查找元素集
    default_random_engine
    e(SEED); unordered_set<int>
    insert_us, find_us; for(int i=0;
    i<n; i++)
        insert_us.insert(
    e()); for(int i=0; i<t;
    i++)
        find_us.insert(e());
    for(; m>=100000; m- =100000){
        cout << "- - -  m = " << m << " - - - " << endl;
        // test BloomFilter
        cout << "TEST BLOOM FILTER..." << endl;
        BloomFilter
        bf(m); for(int v :
        insert_us)
            bf.insert(v);
        int error  = 0;
        for(int v :
        find_us){
            if(!insert_us.count(v) &&
                bf.find(v)) error++;
        }
        cout << "False Positive Rate = " << 1.0 * error / t
        << endl; cout << "Capacity = " << bf.cap() <<
        endl;
        cout << endl;
        // test ScalableBloomFilter
        cout << "TEST SCALABLE BLOOM FILTER..." << endl;
        ScalableBloomFilter
        sbf(m, fp); for(int v :
```

```
            insert_us)
                sbf.insert(
            v); error = 0;
            for(int v : find_us){
                if(!insert_us.count(v) &&
                    sbf.find(v)) error++;
            }
            cout << "False Positive Rate = " << 1.0 * error / t
            << endl; cout << "Capacity = " << sbf.cap() <<
            endl;
            cout << "Resize Time = " << sbf.resizeTime() << endl;
        }
        return 0;
    }
```

实验结果

第一组测试：预设最大误判率 fp = 0.01

测试结果如表 4.2 所示。

其中 m 为初始 BF 哈希数组容量；fp1、fp2 分别为基础 BF 结构与 SBF 结构最终查询的实际总体误判率；capacity 为SBF 最终空间占用，对应 SBF 在插入过程中进行自动扩容的次数。

表 4.2 测试结果（fp = 0.01）

m	fp1	fp2	capacity	扩容次数
1000000	0.0099	0.0099	1000000	0
900000	0.0119	0.0088	2700000	1
800000	0.0271	0.0132	2400000	1
700000	0.0385	0.0114	2100000	1
600000	0.0576	0.0112	1800000	1
500000	0.0985	0.0105	1500000	1
400000	0.1847	0.0133	1200000	1
300000	0.3585	0.0215	2100000	2
200000	0.645	0.0189	1400000	2

	7			
100000	0.9666	0.0292	1500000	3
50000	0.9999	0.0405	1550000	4

第二组测试：预设最大误判率 fp = 0.001

测试结果如表 4.3 所示。各结果参数含义与第一组测试相同。

表 4.3 测试结果（fp = 0.001）

m	fp1	fp2	capacity	扩容次数
1000000	0.0099	0.0017	3000000	1
900000	0.0119	0.0013	2700000	1
800000	0.0271	0.0009	2400000	1
700000	0.0385	0.0017	2100000	1
600000	0.0576	0.0015	1800000	1
500000	0.0985	0.0022	3500000	2
400000	0.1847	0.0025	2800000	2
300000	0.3585	0.0028	2100000	2
200000	0.6457	0.0040	3000000	3
100000	0.9666	0.0045	3100000	4
50000	0.9999	0.0056	3150000	5

结语

本文分析了传统 Bloom Filter 结构原理与流程，针对其在大规模数据插入情况下误判率激增的缺点，提出改进结构 Scalable Bloom Filter，实现可自动扩容的多层数据存储与查找。

通过理论分析与实验测试，对比上述两种结构在大规模数据插入情况下实际误判率变化趋势，论证了 Scalable Bloom Filter 通过自动扩容控

制误判率的有效性与优越性。

参考文献

[1] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, “Beyond Bloom Filters: From Approximate Membership Checks to Approximate State Machines,” Proc. ACM SIGCOMM, 2006.

[2] 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.

[3] S. Dharmapurikar, P. Krishnamurthy, and D.E. Taylor, “Longest Prefix Matching Using Bloom Filters,” Proc. ACM SIGCOMM, pp. 201- 212, 2003.

[4] L. Fan, P. Cao, J. Almeida, and A. Broder, “Summary Cache: A Scalable Wide- Area Web Cache Sharing Protocol,” IEEE/ACM Trans. Networking, vol. 8, no. 3, pp. 281- 293, June 2000.

[5] B. Xiao and Y. Hua, “Using Parallel Bloom Filters for Multi- Attribute Representation on Network Services,” IEEE Trans. Parallel and Distributed Systems, vol. 21, no. 1, pp. 20- 32, Jan. 2010.

[6] Y. Hua, Y. Zhu, H. Jiang, D. Feng, and L. Tian, “Scalable and Adaptive Metadata Management in Ultra Large- scale File Systems,” Proc. 28th Int’l Conf. Distributed Computing Systems (ICDCS ’08), pp. 403- 410, 2008.

[7] D. Guo, J. Wu, H. Chen, and X. Luo, “Theory and Network Application of Dynamic Bloom Filters,” Proc. IEEE INFOCOM, 2006.

[8] K. Xie, Y. Min, D. Zhang, J. Wen, and G. Xie, “A Scalable Bloom

Filter for Membership Queries," IEEE Global Telecommunications Conference, 2007