華中科技大學

课 程 实 验 报 告

课程名称： 物联网数据存储与管理

选题名称： 基于 Bloom Filter 的
多维数据属性表示和索引

专业班级： 物联网 1801 班

学　　号： U201814597

姓　　名： 曹金羽

指导教师： 华宇

报告日期： 2021 年 6 月 21 日

计算机科学与技术学院

目　录

1 选题背景与意义

Bloom filter（布隆过滤器）是 Howard Bloom 在 1970 年提出的二进制向量数据结构，具有良好的空间和时间效率，用于检测某元素是否为集合的成员。

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，不会漏判（召回率 100%），但可能误判。因此 Bloom Filter 不适合要求“零错误”的应用场合，但在能容忍低错误率的应用场合下，可以通过极少的错误换取存储空间的极大节省。

Bloom Filter 给出检测结果时，若判断结果为否，则该元素一定不在集合中；若判断结果为是，该元素可能并不在集合中。这种误判的情况被称为 false positive（假阳性，假正例）。当插入的元素增多到一定程度时，false positive 的概率将快速增长到不可接受的程度。

通过本课题研究，尝试设计可扩展的 Bloom Filter 结构，使 Bloom Filter 在大量元素插入的条件下能将 false positive 维持在一定的可接受范围，并给出一个具体的实验样例。

2 总体设计

2.1 Bloom Filter 原理

2.1.1 Bloom Filter 基本流程

为判断某元素是否在某集合中，Bloom filter 采用哈希函数的方法，将一个元素映射到一个长度为 m 的阵列上的一点，将其标志置为 1。进行判断时，若对应点标志为 1，表明对应元素在集合内，反之则表明其不在集合内。

该方法的缺点来自于哈希函数的特性，多个不同的元素经过哈希可能映射到同一个点，也即哈希冲突，如图 2.1 所示。当集合插入元素逐渐增多，产生冲突的频率将明显提高。

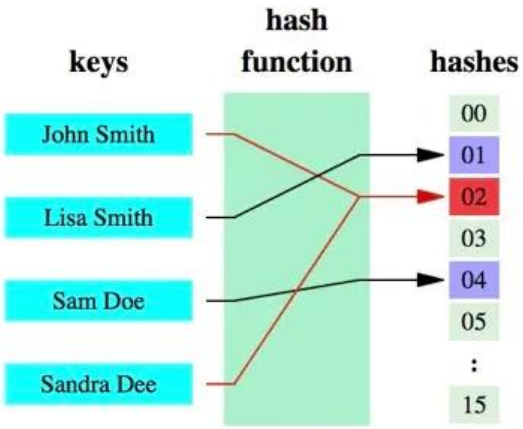

图 2.1 哈希冲突示意图

为解决上述冲突问题，Bloom Filter 采用多哈希法。

假定 Bloom Filter 通过一个大小为 m 的序列保存对应点信息。起始时，Bloom Filter 对应一个长度为 m、内部元素均为 0 的二进制向量。当一个元素 x 插入时，Bloom Filter 通过 k 个不同的哈希函数对 x 进行计算，得到多个 1 位的计算结果，将向量中对应位置的标志置 1，由此将 x 映射到$\{1,2 \ldots m\}$范围内的多个位置。如图 2.2 所示。

进行判断时，若某一对应点的标志值不为 1，则可以确定目标元素不在集合中；所有对应点的标志值均为 1 时，可以认为目标元素在集合中。

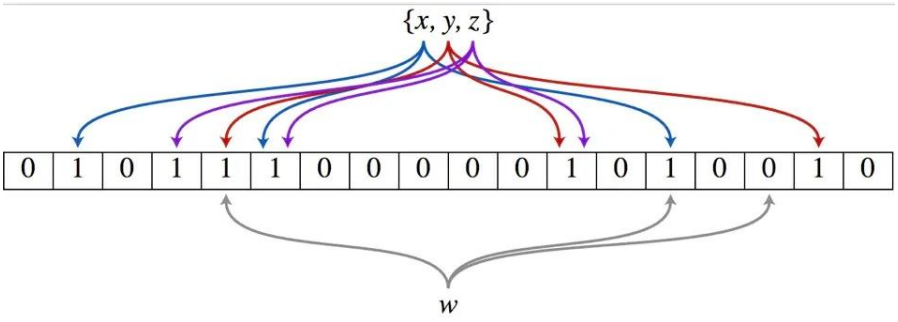

图 2.2 Bloom Filter 多哈希算法示意图

插入元素逐渐增多时，有一定概率出现以下情况：对于某个未插入的元素 y，其值通过 k 个哈希函数计算得到的多个结果，在向量中对应位置的标志均已被此前插入的元素置为 1。此时 Bloom Filter 检查 w 对应的 k 个对应位置，发现标志均为 1，认为 w 在集合内，由此产生误判，也即 false positive。

2.1.2 Bloom Filter 误判率分析

考虑分析 Bloom Filter 的误判率（以下记为 FP）。

起始时，二进制向量 m 各位置均为 0，如图 2.3 所示。

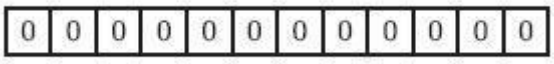

图 2.3 Bloom Filter 向量起始状态

假设哈希函数计算结果为{1,2 ... m}各值的概率均等，此时对某插入元素进行一次哈希计算，某特定位置值仍为 0 的概率是$\left(1-\frac{1}{m}\right)$。由于 k 个函数彼此无关，对 n 个元素进行 k 次哈希计算后，某特定位置仍为 0 的概率 p 为：

$$\mathrm{p}=\left(1-\frac{1}{m}\right)^{nk}=\left(1-\frac{1}{m}\right)^{m\cdot\frac{nk}{m}}\approx e^{-\frac{nk}{m}}$$

考虑 Bloom Filter 发生误判的情形：某不在集合中的元素 x，其 k 个哈希计算结果对应位置值均为 1。由于 k 个哈希计算结果在{1,2 ... m}随机分布，该事件概率 fp（也即插入 n 个元素后产生 false positive 的概率）等价于在向量中随机选取 k 个位置，其值均为 1 的概率，其值为：

$$\mathrm{fp}=(1-p)^k\approx\left(1-e^{-\frac{nk}{m}}\right)^k=e^{\mathrm{k}\ln\left(1-e^{-\frac{nk}{m}}\right)}=e^{-\mathrm{k}\cdot\frac{m}{nk}\ln e^{-\frac{nk}{m}}\ln\left(1-e^{-\frac{nk}{m}}\right)}\quad\cdots\cdots\ ①$$

$$\mathrm{fp}=e^{-\frac{m}{n}\cdot\ln(p)\cdot\ln(1-\mathrm{p})}\quad\cdots\cdots\ ②$$

式②中，指数函数e^x的指数$-\frac{m}{n}\cdot\ln(p)\cdot\ln(1-\mathrm{p})$取其最小值、也即$\ln(p)\ln(1-\mathrm{p})$取其最大值时，误判概率 fp 有最小值。此时：

$$\mathrm{p}=e^{-\frac{nk}{m}}=\frac{1}{2}$$

也即：

$$\mathrm{k}=\frac{m}{n}\ln 2\approx 0.7\frac{m}{n}$$

此时 fp 取其最小值：

$$\mathrm{f}=\left(\frac{1}{2}\right)^k\approx 0.6185^{\frac{m}{n}}$$

2.1.3 Bloom Filter 的不足

经过前文对 Bloom Filter 结构原理的分析与误判率的计算，可以发现 Bloom Filter 误判率 fp 与哈希函数数量 k、集合容量 m、当前已插入元素数量 n 有关。在实际应用中，限制插入元素的数量是违背实际需求的。因此需要考虑设计哈希函数数量 k 与集合容量 m。

此外，Bloom Filter 在插入元素时，若哈希函数计算结果对应位置的值已被置为 1，则不做操作。因此，向量中为 1 的标志位可能对应多个元素的映射，却无法保存这一信息。尝试删除元素时，若直接将对应位置的标志置 0，可能影响多个其他元素的判断。

2.2 Bloom Filter 结构设计

2.2.1 Counter Bloom Filter 结构

Counter Bloom Filter 是以 Bloom Filter 初始结构为基础设计的多维数据属性表示结构。Counter Bloom Filter 不再使用二进制向量保存元素信息，而是为向量的每一个位置维护一个计数器（Counter），用以记录该位置上元素映射的个数。每当新元素插入、进行某一次哈希函数计算后，计算结果对应位置的计数器值加一。

这一结构在总体保留了 Bloom Filter 高时间、空间效率的基础上，通过增加一定的存储空间，解决了后者难以删除已有元素的问题。

此外，其误判率特性与传统结构的 Bloom Filter 相同。

2.2.2 Scalable Bloom Filter 结构

对于传统结构的 Bloom Filter 与 Counter Bloom Filter，存储向量的长度固定；当插入元素数量 n 增大时，向量中值为 1 的位置比例增大，此时出现误判情形的概率随之上升。由 2.1.2 节式②可知，哈希函数数量 k、向量容量 m 固定时，误判率 fp 将随着插入元素数量 n 增大而以指数级速度增大。

为将误判率 fp 控制在应用场景可接受的范围内，提出可拓展容量的 Bloom Filter 结构——Scalable Bloom Filter。它能在误判率 fp 超出一定限度时，自动拓展存储向量的容量 m 以降低 fp，使其控制在预设的标准附近。

Scalable Bloom Filter 具体结构设计如下：

1. 起始时创建一个容量为 m 的 Bloom Filter，限制其误判率不超过 fp，并将该预设值带入 2.1.2 节式①，计算出该条件下允许插入的最大元素数量 N。
2. 当前已插入元素数量 n 达到 N 时，认为误判率同样达到限制值 fp，于是创建一个容量为 2m、误判率限制值仍为 fp 的 BF 结构作为新一层。
3. 同理类推，每当此前创建的 BF 结构达到预设插入量 N 时，创建一个新的 BF 结构，其容量 m'为前一结构容量的 m 的 2 倍，误判率限制值仍为 fp。

设起始时 BF 结构层数为 0，则第 i 层 BF 结构容量为$2^i \cdot m$。带入 2.1.2 节式①可知，该层最多容纳插入个数

$$N = -\frac{m}{k}\ln\left(1 - f_p^{\frac{1}{k}}\right) \quad \cdots\cdots \ ③$$

2.3 Scalable Bloom Filter 流程设计

2.3.1 Scalable Bloom Filter 插入流程

1. 即将插入第 n+1 个元素时，检查 n 是否已达到当前层 BF 结构允许插入的最大元素量 N；
2. 若 n 达到 N，先按照前述结构规律对 BF 进行扩容；
3. 在新一层 BF 结构中插入第 n+1 个新元素，也即根据 k 个哈希函数的计算结果将对应位置值置 1；

2.3.2 Scalable Bloom Filter 查找流程

1. 根据目标元素经 k 个哈希函数的计算结果，在最顶层（最新创建的 BF 层）对应位置检验目标元素是否存在；
2. 若不存在，依次在前一层 BF 重复上述步骤，查找目标元素；
3. 若达到初始化层，仍未查找到目标元素，则判断其不在集合中。

最坏情况下，需要查找所有 i+1 层 SBF，则需要进行的哈希计算次数为：

3 理论分析

1. 分析经过 i 次扩展后，整个 SBF 允许容纳的元素个数。

根据 2.2.2 节式③，考虑第一次拓展 SBF 结构，添加$SBF_1 = \{n_1, m_1, k\}$。

其中$m_1 = 2m_0$。继而得到$n_1 = 2n_0$。

经等比递推，有$n_i = 2^i \cdot n_0$。

将n_0到n_i各层可容纳元素个数求和（等比求和），得到整个 SBF 允许容纳的元素个数$N = 2^{i+1}n_0$.

2. 考虑分析，为表示含有 n 个元素的集合，SBF 需要经过 i 次拓展。

设最终的 SBF 包含 L 个二进制向量，其中最后一层SBF_i代表 t 个元素；整个 SBF 占据M_{SBF}位，对应误判率为f_{SBF}。

$$i = [\log_2(n/n_0 + 1)]$$

$$t = n - n_0 \cdot (2^{[\log_2(n/n_0+1)]} - 1)$$

$$M_{SBF} = (2^i - 1)m = \left(2^{\left\lfloor \log_2 \frac{n}{n_0} + 1 \right\rfloor} - 1\right)m$$

$$f_{SBF}(m,k,n_0,n) = 1 - (1 - (1 - e^{-kn_0/m})^k)^{[\log_2(n/n_0+1)]}(1 - (1 - e^{-kt/m_i})^k)$$

以下证明：

要表示 n 个元素，SBF 需要满足如下的公式，其中的 i 为需要的 SBF 拓展次数：$(2^i - 1)n_0 = n_{i-1\_\max} < n \le n_{i\_\max} = (2^{i+1} - 1)n_0$

则需要的拓展次数 i 为：$\log_2(n/n_0 + 1) - 1 \le i < \log_2(n/n_0 + 1)$

容易看出，在经过 i 次拓展之后，SBF 数组的大小变为：

$$L = i + 1 = \left\lfloor \log_2 \frac{n}{n_0} + 1 \right\rfloor + 1$$

那么 SBF 整体需要占用的位大小为：

$$M_{SBF} = m_0 + m_1 + \cdots + m_i = m + 2m + 4m + \cdots 2^i m = m(2^{i+1} - 1)$$

需要注意的是，SBF_j表示了$(2^j)n_0$个元素。除去最后一层SBF_i表示的 t 个元素之外，$t = n - n_0(2^j - 1)$，前 j 个 SBF 误判率 f 为：

$$f^{BF}\left(m_j,k,n_j\right)=\left(1-e^{-k\left(2^j n_0\right)/\left(2^j m\right)}\right)^k=\left(1-e^{-kn_0/m}\right)^k=f^{BF}\left(m,k,n_0\right)$$

最后 t 个元素在最后一层SBF_i中表示，其错误率 f 为：

$$f^{BF}\left(m_i,k,t\right)=\left(1-e^{-kt/m_i}\right)^k$$

进而得到整个 SBF 误判率f_{SBF}为

$$f^{SBF}\left(m,k,n_0,n\right)=1-\prod_{j=0}^{j=i-1}\left(1-f^{BF}\left(m_j,k,n_j\right)\right)\left(1-f^{BF}\left(m_i,k,t\right)\right)$$

4 实验测试

4.1 实验设计

4.1.1 测试参数

实验参数符号、含义、配置如表 4.1 所示。

表 4.1 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	10^5
t	查找元素个数	10^4
k	哈希函数个数	5（尽量保证独立）

4.1.2 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：

1. 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。
2. 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。
3. 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{
private:
    vector<bool> bits;
    int len;
    static int hash1(int v, int m) {
        return (((v >> 3) ^ (v << 5)) & 0x7fffffff) % 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;
    }
```

```
        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.resize(1);
            bits[0].resize(len);
        }
        void resize(){
            depth += 1;
            bits.resize(depth+1);
            bits[depth].resize(len << 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;
        }
```

```
    int cap() const { return (len << (depth + 1)) - len; }
    int resizeTime() const { return depth; }
};
```

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;
    }
```

4.2 实验结果

1. 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	0.6457	0.0189	1400000	2
100000	0.9666	0.0292	1500000	3
50000	0.9999	0.0405	1550000	4

2. 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

对两组测试结果，分别绘制 fp1、fp2 与 m 关系散点图。如图 4.1、图 4.2 所示。

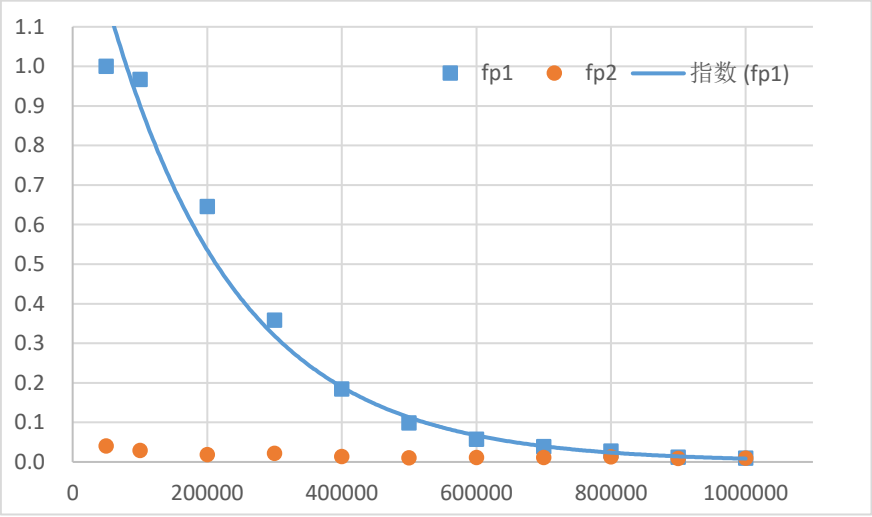

图 4.1 测试结果（fp = 0.01）

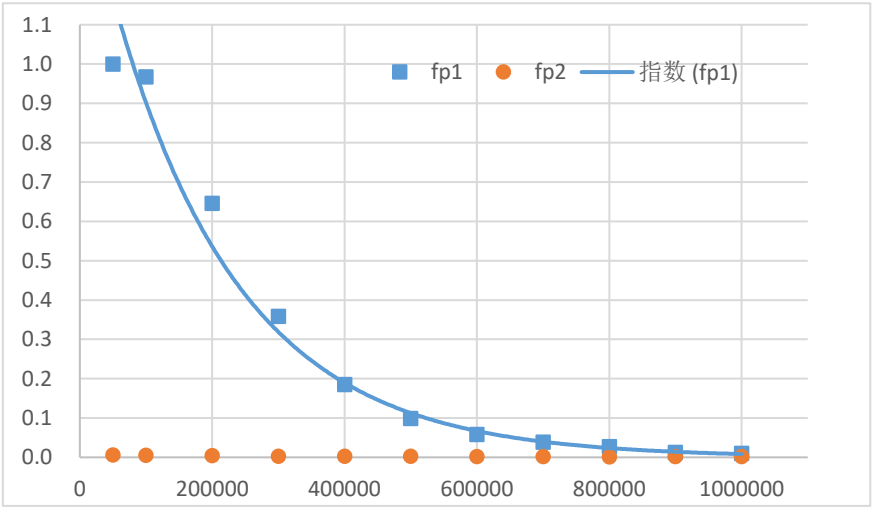

图 4.2 测试结果（fp = 0.001）

由图 4.1、图 4.2 可见，对于固定的插入元素数量与查询元素数量，随着 Bloom Filter 存储数组初始容量减小，存储数组相对饱和度增大，基础 Bloom Filter 结构的实际总体误判率 fp1 以指数级别速度增长；插入元素数量接近数组容量时，误判率 fp1 接近 1。

与之相比，存储数组初始容量减小时，Scalable Bloom Filter 结构的总体误判率 fp2 变化不明显；存储数组初始容量足够时，SBF 结构能将误判率 fp2 控制在预设最大允许误判率 fp 水平附近；预设误判率 fp 要求越严格，SBF 限制实际误判率的特性优势越明显。

另一方面，SBF 限制误判率的代价是占用更大的存储空间。当插入饱和度较高、误判率限制较严格时，SBF 在插入过程中将进行多次扩容操作，最终空间占用量 capacity 可达初始容量的数倍乃至数十倍。其中，由于 SBF 的每次扩容，其新存储层的容量均为前一层容量的 2 倍，最新层（也即容量最大的一层）往往并未达到最大容许插入量，造成大量存储空间浪费。

此外，插入元素数量接近 SBF 数组初始容量时，最终总体误判率 fp2 仍然明显超过预设容许最大误判率 fp。可能的原因是：计算当前误判率、最大容许插入量的推导过程存在近似，以及 k 个哈希函数不一定完全独立。

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。

综上所述，Scalable Bloom Filter 以可接受的存储空间占用为代价，换取有效的误判率控制能力，同时基本不影响 Bloom Filter 本身在插入、删除、查找方面良好的空间和时间效率，是一种针对 Bloom Filter 性能的有效优化方案。

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