

**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 1](#_Toc75188825)

[2 总体设计 2](#_Toc75188826)

[2.1 Bloom Filter原理 2](#_Toc75188827)

[2.2 Bloom Filter结构设计 4](#_Toc75188828)

[2.3 Scalable Bloom Filter流程设计 5](#_Toc75188829)

[3 理论分析 7](#_Toc75188830)

[4 实验测试 9](#_Toc75188831)

[4.1 实验设计 9](#_Toc75188832)

[4.2 实验结果 13](#_Toc75188833)

[5 结语 16](#_Toc75188834)

[参考文献 17](#_Toc75188835)

# 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。

# 总体设计

## Bloom Filter原理

### 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。

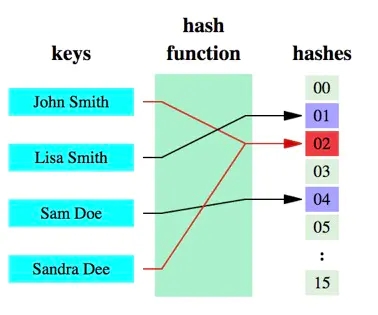


图 2.1 哈希冲突示意图

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。

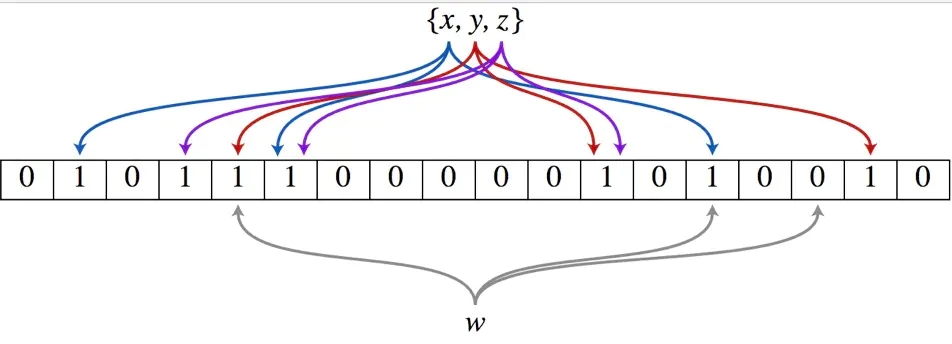


图 2.2 Bloom Filter多哈希算法示意图

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。

### 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。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：

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：

…… ①

…… ②

式②中，指数函数的指数取其最小值、也即取其最大值时，误判概率fp有最小值。此时：

也即：

此时fp取其最小值：

### 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。

## Bloom Filter结构设计

### 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。

### 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。带入2.1.2节式①可知，该层最多容纳插入个数

…… ③

## Scalable Bloom Filter流程设计

### 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；

### 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：

# 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。

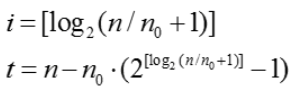
其中。继而得到。

经等比递推，有。

将到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.

1. 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，其中最后一层代表t个元素；整个SBF占据位，对应误判率为。



=



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：

则需要的拓展次数i为：

容易看出，在经过i次拓展之后，SBF数组的大小变为：

+ 1

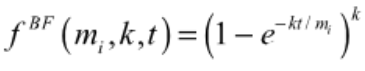
那么SBF整体需要占用的位大小为：



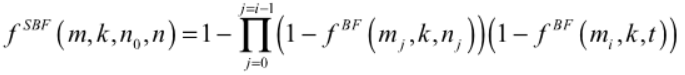
需要注意的是，表示了个元素。除去最后一层表示的t个元素之外，，前j个SBF误判率f为：



最后t个元素在最后一层中表示，其错误率f为：



进而得到整个SBF误判率为



# 实验测试

## 实验设计

### 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 | ~步进  （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 |  |
| t | 查找元素个数 |  |
| k | 哈希函数个数 | 5（尽量保证独立） |

### 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;

}

## 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 |

1. 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。

# 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