

**大数据系统管理与处理课程报告**

|  |  |
| --- | --- |
| 姓 名： | 陆云龙 |
| 学 院： | 计算机学院 |
| 专 业： | 计算机科学与技术 |
| 班 级： | CS1903 |
| 学 号： | U201914987 |
| 指导教师： | 华宇 |

|  |  |
| --- | --- |
| 分数 |  |
| 教师签名 |  |

2022年4月18日

目 录

[1 Bloom Filter结构 1](#_Toc101342419)

[1.1 基本描述 1](#_Toc101342420)

[1.2 BitSet结构与hash函数 1](#_Toc101342421)

[2 设计与实现 3](#_Toc101342422)

[2.1 MultiSimpleHash类 3](#_Toc101342423)

[2.2 MultiBloomFilter类 3](#_Toc101342424)

[3 False Positive 6](#_Toc101342425)

[4 测试结果 7](#_Toc101342426)

# Bloom Filter结构

## 基本描述

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。其核心实现是一个超大的位数组和几个哈希函数。

默认情况下，初始化比特数组BitSet时，集合中的所有位最初都为值false 。然后需要引入k个不同的哈希函数，新增元素通过这k个哈希函数处理之后，把BitSet中命中映射的k个比特位设置为1，产生一个均匀的随机分布。

这个时候如果需要判断一个元素是否存在于Bloom Filter中，只需要通过k个散列函数处理得到比特数组的k个下标，然后判断比特数组对应的下标所在比特是否为1。如果这k个下标所在比特中至少存在一个0，那么这个需要判断的元素必定不在Bloom Filter代表的集合中；如果这k个下标所在比特全部都为1，那么这个需要判断的元素可能存在于布隆过滤器代表的集合中或者出现失误定位(False Positive)。具体过程如图 1所示；

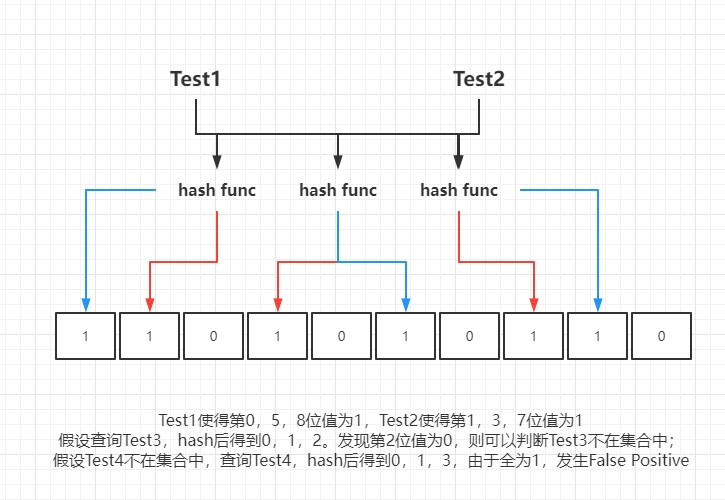


图 1 Bloom Filter原理

## BitSet结构与hash函数

对于简单的Bloom Filter，其BitSet可以使用一维数组实现。为了能够满足处理多维数据的需求，使用BitSet数组的形式，即数组的每个元素都为一个BitSet，对应存储数组一个维度的hash处理情况。

假设数据有n维，则采用n组hash函数对每一维进行处理。每组hash函数中有数个各不相同的函数，用它们对每一维数据进行处理，随后将结果映射到相应维度的BitSet。处理过程如图 2所示：

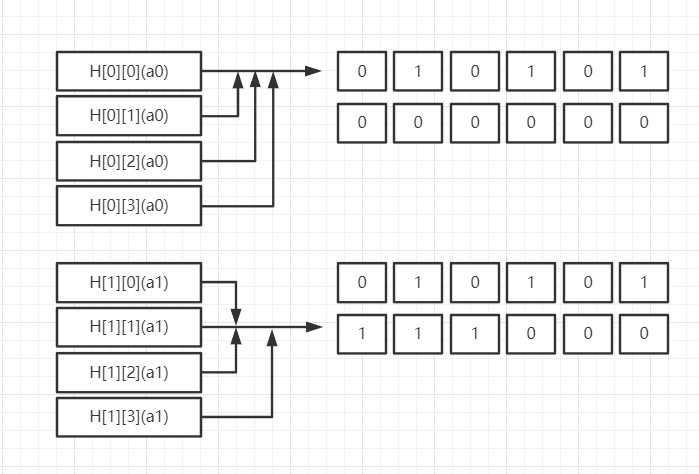


图 2 多维处理

# 设计与实现

## MultiSimpleHash类

MultiSimpleHash类实现初始化hash函数，并且可以通过hash函数处理数据。简易的hash函数算法为：设定一个初值为0的变量result，result首先乘以一个特定的整数seed，随后加上输入数据的一个字符，循环直至遍历输入数据的所有字符。最后，需要对result进行处理，使其小于BitSet的容量，这样即可通过hash函数将数据映射到BitSet上。主要代码如下：

public int hash(String val){  
 int result=0;  
 int len=val.length();  
 for(int i=0;i<len;++i){  
 result=seed\*result+val.charAt(i);  
 }  
  
 //与运算两位同时为1，结果才为1。确保计算出来的结果不会超过size  
 return (space-1)&result;  
}

通过设置不同的seed值，可以方便的得到不同的hash函数。故该类的私有变量及构造函数如下：

//过滤器容量  
private int space;  
private int seed;  
  
public MultiSimpleHash(int space,int seed){  
 this.space=space;  
 this.seed=seed;  
}

## MultiBloomFilter类

对于Bloom Filter，它需要实现的主要功能为构造BitSet，插入数据以及查询。

对于每一维数据，需要一个BitSet，故构造一个BitSet来实现存储。可通过引入java.util.BitSet，使用BitSet类实现。同时，为每一维数据分配一组hash函数，使用MultiSimpleHash类中实现的hash函数，只需分配不同的seed值即可。支持三维数据处理的Bloom Filter需要的变量如下：

//设置容量  
private static final int *SIZE*=9;  
//各个hash函数的应当有不同的seed，选取素数  
private static final int[][] *seeds*={{2,3,5,7,11,13,17,19,23},  
 {29,31,37,41,43,47,53,59,61},{67,71,73,79,83,89,91,101,103}};  
  
private BitSet[] bits=new BitSet[*seeds*.length];  
private MultiSimpleHash[][] func=new MultiSimpleHash[*seeds*.length][*seeds*[0].length];

为方便测试，规定Bloom Filter规模为9，使用素数作为hash函数seed值，若动态生成seed则可实现任意多维数据处理。其构造如下：

public MultiBloomFilter(){  
 for(int i=0;i<*seeds*.length;++i){  
 bits[i]=new BitSet();  
 }  
  
 //建立各个hash函数  
 for(int i=0;i<*seeds*.length;++i){  
 for(int j=0;j<*seeds*[0].length;++j){  
 func[i][j]=new MultiSimpleHash(*SIZE*,*seeds*[i][j]);  
 }  
 }  
}

插入一个数据时，需要对数据的每一维使用相应的hash函数组处理，将结果写入对应每一维的BitSet中。在外层循环遍历数据的每一维，在内层循环遍历该维度对应的每一个hash函数，同时记录结果。具体实现如下：

public void add(String[] val){  
 int n=val.length;  
 for(int i=0;i<n;++i){  
 //第i维数据hash存储到第i个BitSet  
 for(MultiSimpleHash f:func[i]){  
 bits[i].set(f.hash(val[i]),true);  
 }  
 }  
}

对一个数据进行查询时，对其进行相应的hash处理。每一维度都会得到一个结果，类似于简易的Bloom Filter，将得到的结果与BitSet相应位置的值进行判断。如果有一处为false，则此数据必定不在集合中；若全为true，则此数据有可能在集合中也有可能发生False Positive。具体实现如下：

public boolean contains(String[] val){  
 boolean ans=true;  
 int n=val.length;  
 for(int i=0;i<n;++i) {  
 for (MultiSimpleHash f : func[i]) {  
 ans = ans && bits[i].get(f.hash(val[i]));  
 }  
 }  
 return ans;  
}

# False Positive

假设Bloom Filter中的hash func使每个元素都等概率地hash到m个slot中的任何一个，与其它元素被hash到哪个slot无关（独立性）。若m为bit数，则对某一特定bit位在一个元素由某特定hash func插入时没有被置位为1的概率为：

则k个hash func插入时没有被置位为1的概率为：，插入n个元素后，没有被置为1的概率是。所以被置为1的概率为。在查询某元素，得到所有位为1的概率为。当m很大时：

设，则，对等式两边取对数得到:

随后对等式两边求导：。为求得其最值，令f’(k)=0，即，得出当时取得最大值，此时。

所以，可以通过控制m和n来使得Bloom Filter的误判率达到最低，只要这种可能性足够的小以至于应用能容忍这种误差，由于其哈希查找的常数时间和少量的存储空间开销，Bloom Filter具有非常高的实用价值。

# 测试结果

由False Positive的推导可以发现，对于给定hash函数的Bloom Filter而言，增加m或者减少n会使得失误定位的概率下降。而对于给定m/n的情况， hash函数的数量最好接近。

由于缺少完善的测试集，我模拟了一些简单的情况，通过固定BitSet大小和已添加元素，或者固定hash函数数量，进行了一些简单的测试。例如，当hash函数数量k=3，BitSet大小m=10时，通过改变已经插入的数据n的数量，形成不同的Bloom Filter。对这些Bloom Filter进行简单的查询测试，其结果如图 3和图 4所示：

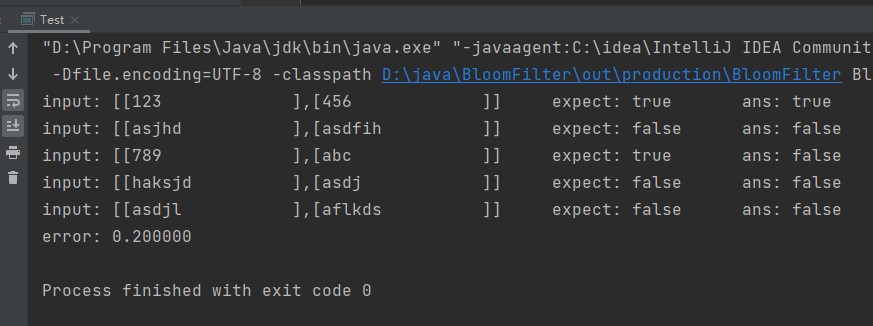


图 3 k=3,m=10,n=1测试结果

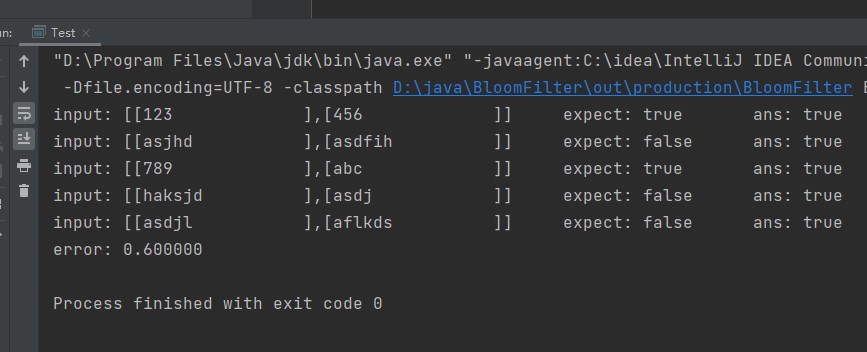


图 4 k=3,m=10,n=3测试结果

通过分析测试结果可以发现，在m/n减小的时候，失误定位的概率在增大。为了更好的验证结论，我随后增加了数据的总量，进行了其他多种情况的模拟，得到的结果如图 5所示：

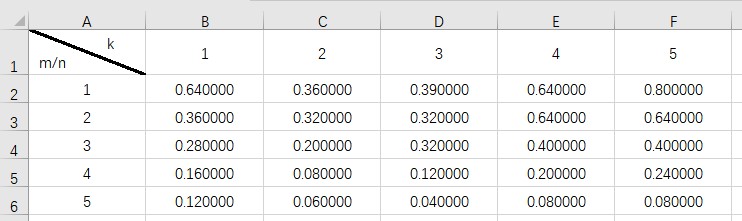


图 5 各个参数对错误率的影响

由于总体的数据较少，在某一些测试中，结果展示的规律并不严格符合预期，但还是可以大致发现：在m/n不变的情况下，hash函数的数量k的取值在ln2m/n附近时，错误率最低。而在hash函数数目固定时，随着m/n的增大，错误率逐渐减小，即在构造Bloom Filter时，选取尽量大的BitSet会带来更好的效果。同时，若Bloom Filter固定，即m与k固定，通过填充合适数量的数据，会使得查询结果更好。

# 参考文献

1. 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.
2. A stand-alone Bloom filter implementation written in Java https://github.com/MagnusS/Java-BloomFilter
3. 布隆过滤器(Bloom Filter)算法的实现原理，https://blog.csdn.net/qq\_39140300/article/details/118246591