



华中科技大学

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
1.1	基本描述.....	1
1.2	BitSet 结构与 hash 函数	1
2	设计与实现.....	3
2.1	MultiSimpleHash 类	3
2.2	MultiBloomFilter 类	3
3	False Positive	6
4	测试结果.....	7

1 Bloom Filter 结构

1.1 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；

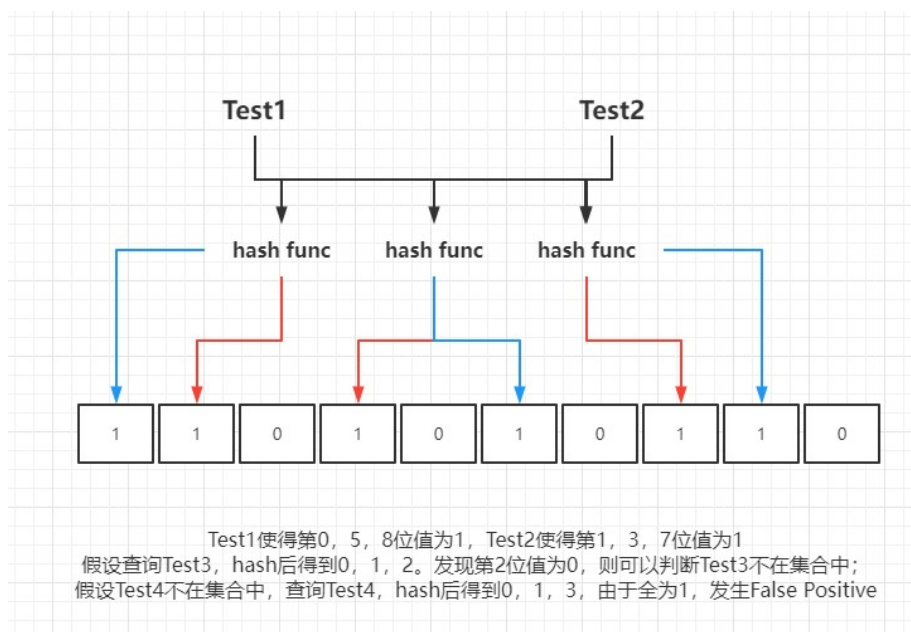


图 1 Bloom Filter 原理

1.2 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:

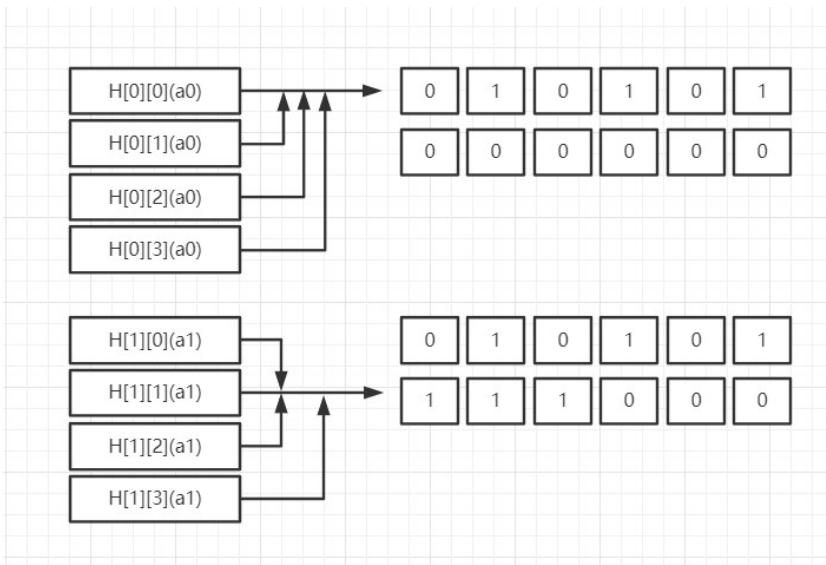


图 2 多维处理

2 设计与实现

2.1 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;  
}
```

2.2 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;
}

```

3 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：

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

则 k 个 hash func 插入时没有被置位为 1 的概率为： $\left(1 - \frac{1}{m}\right)^k$ ，插入 n 个元素后，没有被置为 1 的概率是 $\left(1 - \frac{1}{m}\right)^{kn}$ 。所以被置为 1 的概率为 $1 - \left(1 - \frac{1}{m}\right)^{kn}$ 。在查询某元素，得到所有位为 1 的概率为 $\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k$ 。当 m 很大时：

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k = \left(1 - \left(1 - \frac{1}{m}\right)^{\frac{knm}{m}}\right)^k \sim \left(1 - e^{-\frac{kn}{m}}\right)^k$$

设 $b = e^{-\frac{n}{m}}$ ，则 $f(k) = (1 - b^{-k})^k$ ，对等式两边取对数得到：

$$\ln f(k) = k \ln(1 - b^{-k})$$

随后对等式两边求导： $\frac{1}{f(k)} f'(k) = \ln(1 - b^{-k}) + k \frac{b^{-k} \ln b}{1 - b^{-k}}$ 。为求得其最值，

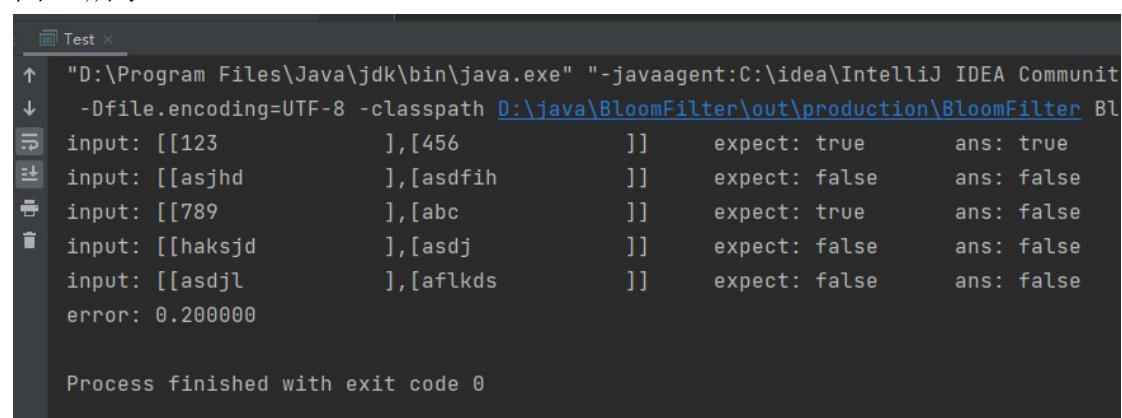
令 $f'(k)=0$ ，即 $\ln(1 - b^{-k}) + k \frac{b^{-k} \ln b}{1 - b^{-k}} = 0$ ，得出当 $k = \ln 2 \frac{m}{n}$ 时取得最大值，此时 $p(\text{error}) = \left(1 - \frac{1}{2}\right)^k \approx 0.6185^{\frac{m}{n}}$ 。

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。

4 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 $\ln 2 \frac{m}{n}$ 。

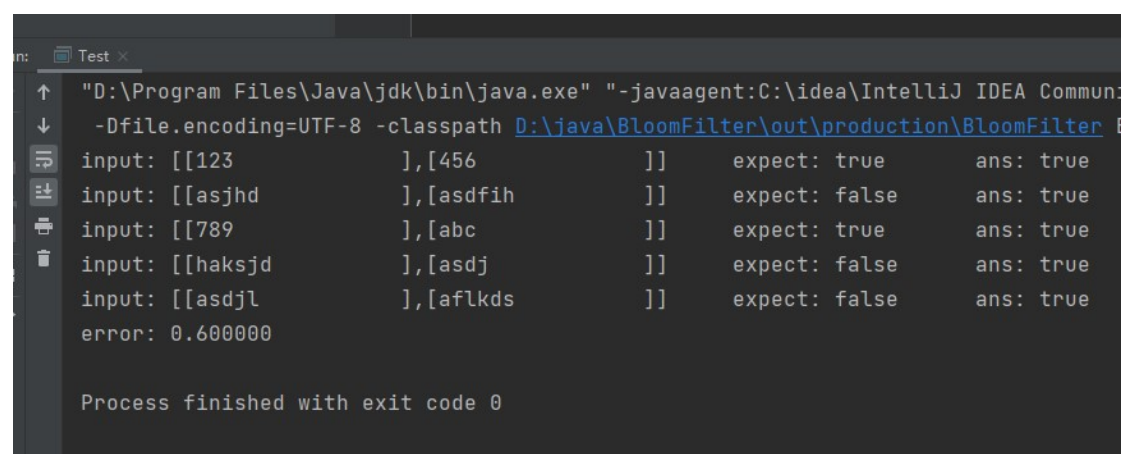
由于缺少完善的测试集，我模拟了一些简单的情况，通过固定 BitSet 大小和已添加元素，或者固定 hash 函数数量，进行了一些简单的测试。例如，当 hash 函数数量 $k=3$ ，BitSet 大小 $m=10$ 时，通过改变已经插入的数据 n 的数量，形成不同的 Bloom Filter。对这些 Bloom Filter 进行简单的查询测试，其结果如图 3 和图 4 所示：



```
Test
↑ "D:\Program Files\Java\jdk\bin\java.exe" "-javaagent:C:\idea\IntelliJ IDEA Communit
↓ -Dfile.encoding=UTF-8 -classpath D:\java\BloomFilter\out\production\BloomFilter BL
input: [[123          ],[456          ]]      expect: true      ans: true
input: [[asjhd        ],[asdfih       ]]      expect: false     ans: false
input: [[789          ],[abc          ]]      expect: true      ans: false
input: [[haksjd       ],[asdj        ]]      expect: false     ans: false
input: [[asdjl        ],[aflkds      ]]      expect: false     ans: false
error: 0.200000

Process finished with exit code 0
```

图 3 $k=3, m=10, n=1$ 测试结果



```
Test
↑ "D:\Program Files\Java\jdk\bin\java.exe" "-javaagent:C:\idea\IntelliJ IDEA Communit
↓ -Dfile.encoding=UTF-8 -classpath D:\java\BloomFilter\out\production\BloomFilter BL
input: [[123          ],[456          ]]      expect: true      ans: true
input: [[asjhd        ],[asdfih       ]]      expect: false     ans: true
input: [[789          ],[abc          ]]      expect: true      ans: true
input: [[haksjd       ],[asdj        ]]      expect: false     ans: true
input: [[asdjl        ],[aflkds      ]]      expect: false     ans: true
error: 0.600000

Process finished with exit code 0
```

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：

	A	B	C	D	E	F
1	$\frac{m}{n}$	1	2	3	4	5
2	1	0.640000	0.360000	0.390000	0.640000	0.800000
3	2	0.360000	0.320000	0.320000	0.640000	0.640000
4	3	0.280000	0.200000	0.320000	0.400000	0.400000
5	4	0.160000	0.080000	0.120000	0.200000	0.240000
6	5	0.120000	0.060000	0.040000	0.080000	0.080000

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 $\ln 2m/n$ 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