



2019 级

# 《物联网数据存储与管理》课程 课 程 报 告

姓 名 王子义

学 号 U201915085

班 号 物联网 1901 班

日 期 2022.04.19

# 目 录

一、实验目的 .....	2
二、实验背景 .....	错误!未定义书签。
2.1 Bloom filter 的提出 .....	3
2.2 Bloom filter 的原理 .....	3
2.3 Bloomfilter 的误判性 .....	4
2.4 Bloom filter 的优缺点 .....	4
三、Bloom Filter 理论分析 .....	5
3.1 Bloom Filter 结构 .....	5
3.2 False positive 分析 .....	5
四、实验设计 .....	6
4.1 实验环境 .....	6
4.2 实验设计 .....	7
4.3 实验代码 .....	7
五、性能测试 .....	9
5.1 实验运行结果 .....	9
5.2 结果分析 .....	9
六、应用场景和实例 .....	9
七、课程感悟 .....	错误!未定义书签。
参考文献 .....	11

# 一、实验目的

- 1、了解 bloom filter 的设计结构和流程；
- 2、通过理论，对 false positive 的概率进行分析；
- 3、基于它的存储结构，设计一个符合要求的存储系统，实现多维数据属性表示和索引；
- 4、对实验性能查询延迟的分析。

# 二、实验背景

## 2.1 Bloom filter 的提出

布隆过滤器（Bloom Filter）是 1970 年由布隆提出的。它实际上是一个很长的二进制向量和一系列随机映射函数。布隆过滤器可以用于检索一个元素是否在一个集合中。它的优点是空间效率和查询时间都远远超过一般的算法，缺点是有一定的误识别率和删除困难。

布隆过滤器它是一种 space efficient 的概率型数据结构，用于判断一个元素是否在集合中。在垃圾邮件过滤的黑白名单方法、爬虫(Crawler)的网址判重模块中等等经常被用到。哈希表也能用于判断元素是否在集合中，但是布隆过滤器只需要哈希表的 1/8 或 1/4 的空间复杂度就能完成同样的问题。布隆过滤器可以插入元素，但不可以删除已有元素。其中的元素越多，false positive rate(误报率)越大，但是 false negative（漏报）是不可能的。

## 2.2 Bloom filter 的原理

BloomFilter 是由一个固定大小的二进制向量或者位图(bitmap)和一系列映射函数组成的。在初始状态时，对于长度为  $m$  的位数组，它的所有位都被置为 0，如下图所示：



图 1

为了表达  $S=\{x_1, x_2, \dots, x_n\}$  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  $\{1, \dots, m\}$  的范围中。对任意一个元素  $x$ ，第  $i$  个哈希函数映射的位置  $h_i(x)$  就会被置为 1 ( $1 \leq i \leq k$ )。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）。



图 2

在判断  $y$  是否属于这个集合时，我们对  $y$  应用  $k$  次哈希函数，如果所有  $h_i(y)$  的位置都是 1 ( $1 \leq i \leq k$ )，那么我们就认为  $y$  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*。



图 3

## 2.3 Bloom filter 的误判性

查询某个变量的时候我们只要看看这些点是不是都是 1 就可以大概率知道集合中有没有它了

- 如果这些点有任何一个 0，则被查询变量一定不在；
- 如果都是 1，则被查询变量很可能存在

为什么说是可能存在，而不是一定存在呢？那是因为映射函数本身就是散列函数，散列函数是会有碰撞的。

【散列函数的输入和输出不是唯一对应关系的，如果两个散列值相同，两个输入值很可能是相同的，但也可能不同，这种情况称为“**散列碰撞** (collision)”。】

布隆过滤器的误判是指多个输入经过哈希之后在相同的 bit 位置 1 了，这样就无法判断究竟是哪个输入产生的，因此误判的根源在于相同的 bit 位被多次映射且置 1。

这种情况也造成了布隆过滤器的删除问题，因为布隆过滤器的每一个 bit 并不是独占的，很有可能多个元素共享了某一位。如果我们直接删除这一位的话，会影响其他的元素。

## 2.4 Bloom filter 的优缺点

**优点：** 相比于其它的数据结构，布隆过滤器在空间和时间方面都有巨大的优势。布隆过滤器存储空间和插入/查询时间都是常数  $O(K)$ ，另外，散列函数相互之间没有关系，方便由硬件并行实现。布隆过滤器不需要存储元素本身，在某些对保密要求非常严格的场合有优势。布隆过滤器可以表示全集，其它任何数据结构都不能；

**缺点：** 但是布隆过滤器的缺点和优点一样明显。误算率是其中之一。随着存入的元素数量增加，误算率随之增加。但是如果元素数量太少，则使用散列表足矣。另外，一般情况下不能从布隆过滤器中删除元素。我们很容易想到把位数组变成整数数组，每插入一个元素相应的计数器加 1，这样删除元素时将计数器减掉就可以了。然而要保证安全地删除元素并非如此简单。首先我们必须保证删除的元素的确在布隆过滤器里面。这一点单凭这个过滤器是无法保证的。另外计数器回绕也会造成问题。在降低误算率方面，有不少工作，使得出现了很多布隆过滤器的变种。

## 三、Bloom filter 理论分析

### 3.1 Bloom filter 结构

为了表示一个具有  $n$  个元素的集合  $S=\{s_1, s_2, \dots, s_n\}$ ，需要一个长度为  $m$  的二进制向量数组来记录 Bloom filter( $m>n$ )，初始化数组的每一个元素的值为 0；并使用  $k$  个相互独立的哈希函数  $h_1, h_2, \dots, h_k$ ，它们的域值均为  $\{0, 1, \dots, m-1\}$ 。对于每一个元素  $s \in S$ ，将数组中对应于  $h_1(s), h_2(s), \dots, h_k(s)$  的地址位置置成 1。

这样，描述一个元素  $s \in S$  就可以用它的哈希值  $h_1(s), h_2(s), \dots, h_k(s)$  在数组上对应位置的元素值是否全为 1 来表示，只要有一个对应的元素值为 0，则表示这个元素  $s$  不在集合  $S$  上。由上可见，Bloom filter 的本质是哈希计算，不同之处在于 Bloom filter 对同一数据使用多个哈希函数进行多次哈希，将结果保存在同一个向量数组中，所以 Bloom filter 在达到相同的功能的情况下比原始的哈希结构更节约存储空间。Bloom filter 算法的一个缺点在于查询一个元素是否在集合  $S$  上可能存在失误定位(False Positive)。失误定位是指可能存在元素  $b$  且  $b \notin S$ ，但是 Bloom filter 算法判断  $b \in S$ 。

### 3.2 False positive 分析

假设 Hash 函数以等概率条件选择并设置 Bit Array 中的某一位， $m$  是该位数组的大小， $k$  是 Hash 函数的个数，那么位数组中某一特定的位在进行元素插入时的 Hash 操作中没

有被置位的概率是：

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

那么在所有  $k$  次 Hash 操作后该位都没有被置 "1" 的概率是：

$$\left(1 - \frac{1}{m}\right)^k.$$

如果我们插入了  $n$  个元素，那么某一位仍然为 "0" 的概率是：

$$\left(1 - \frac{1}{m}\right)^{kn};$$

因而该位为 "1" 的概率是：

$$1 - \left(1 - \frac{1}{m}\right)^{kn}.$$

现在检测某一元素是否在该集合中。标明某个元素是否在集合中所需的  $k$  个位置都按照如上的方法设置为 "1"，但是该方法可能会使算法错误的认为某一原本不在集合中的元素却被检测为在该集合中（False Positives），该概率由以下公式确定：

$$\left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k.$$

其实上述结果是在假定由每个 Hash 计算出需要设置的位（bit）的位置是相互独立为前提计算出来的，不难看出，随着  $m$ （位数组大小）的增加，假正例（False Positives）的概率会下降，同时随着插入元素个数  $n$  的增加，False Positives 的概率又会上升，对于给定的  $m$ ， $n$ ，如何选择 Hash 函数个数  $k$  由以下公式确定：

$$\frac{m}{n} \ln 2 \approx 0.7 \frac{m}{n},$$

此时 False Positives 的概率为：

$$2^{-k} \approx 0.6185^{m/n}.$$

而对于给定的 False Positives 概率  $p$ ，如何选择最优的位数组大小  $m$  呢，

$$m = -\frac{n \ln p}{(\ln 2)^2}.$$

上式表明，位数组的大小最好与插入元素的个数成线性关系，对于给定的  $m$ ， $n$ ， $k$ ，假正例概率最大为：

$$\left(1 - e^{-k(n+0.5)/(m-1)}\right)^k.$$

## 四、实验设计

### 4.1 实验环境

操作系统: Windows10

处理器: Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz、

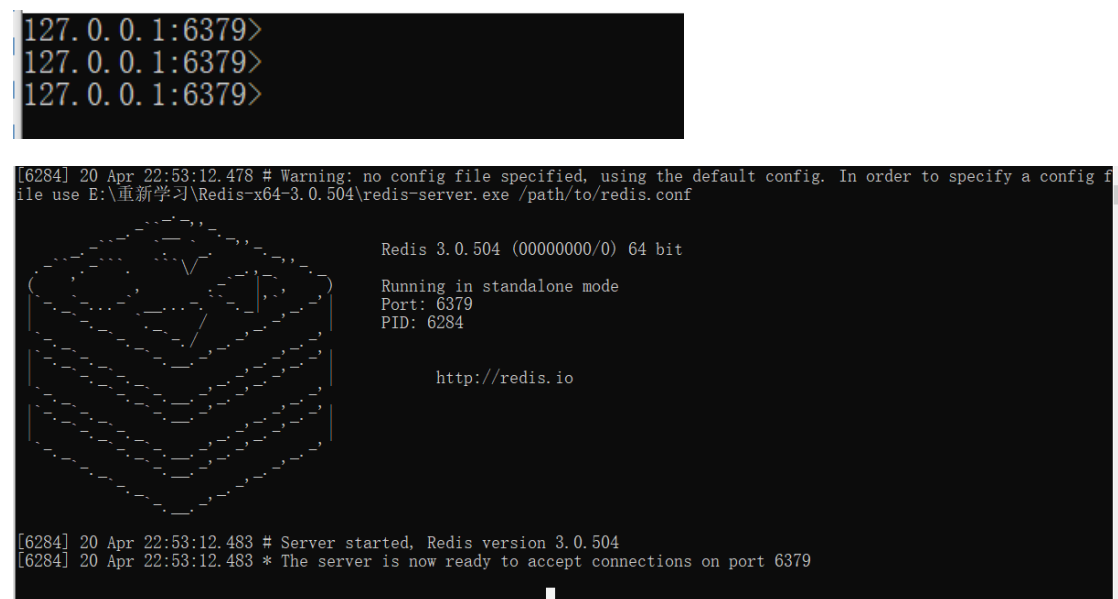
内存: 8GB

工具: Python: 3.9.0

### 4.2 实验设计

本次实验, 旨在实现一个自主设计的 Bloom filter 设计器, 并检验其误判率。

首先在自己的电脑环境中, 下载安装 redis, 然后与自己的电脑进行连接, 获得端口号



```

127.0.0.1:6379>
127.0.0.1:6379>
127.0.0.1:6379>

[6284] 20 Apr 22:53:12.478 # Warning: no config file specified, using the default config. In order to specify a config file use E:\重新学习\Redis-x64-3.0.504\redis-server.exe /path/to/redis.conf

Redis 3.0.504 (00000000/0) 64 bit

Running in standalone mode
Port: 6379
PID: 6284

http://redis.io

[6284] 20 Apr 22:53:12.483 # Server started, Redis version 3.0.504
[6284] 20 Apr 22:53:12.483 * The server is now ready to accept connections on port 6379
  
```

然后, 运行 py 文件, 进行测试。

### 4.3 实验代码

```

import time
from redisbloom.client import Client

rb = Client(host='127.0.0.1', port=6379)
  
```

```
def insert(size, key='book'):
    """插入数据"""
    # for i in range(size):
    #     rb.bfAdd(key, f'book{i}')
    s = time.time()
    step = 1000 # 每次插入 1000 条数据
    for start in range(0, size, step):
        stop = start + step
        if stop >= size:
            stop = size
        rb.bfMAdd(key, *range(start, stop))
    print('插入结束... 花费时间: {:.4f}s'.format(time.time() - s))
```

```
def select(size, key='book'):
    """查询数据"""
    # 统计误判个数
    count = 0

    s = time.time()

    # 单条查询速度太慢了。。。
    # for i in range(size, size * 2):
    #     count += rb.bfExists(key, i)

    step = 1000 # 每次查 1000 条数据
    for start in range(size, size * 2, step):
        stop = start + step
        if stop >= size * 2:
            stop = size * 2
        count += rb.bfMExists(key, *range(start, stop)).count(1) # 返回值[1, 0, 1, ...]统计 1 的个数
    print('size: {}, 误判元素个数: {}, 误判率 {:.4%}'.format(size, count, count / size))
    print('查询结束... 花费时间: {:.4f}s'.format(time.time() - s))
    print('*' * 30)
```

```
def _test1(size, key='book'):
    """测试 size 个不存在的"""
    rb.delete(key) # 先清空原来的 key
    insert(size, key)
    select(size, key)
```



```

def _test2(size, error=0.001, key='book'):
    """指定误差率和初始大小的布隆过滤器"""
    rb.delete(key)

    rb.bfCreate(key, error, size) # 误差率为 0.1%， 初始个数为 size

    insert(size, key)
    select(size, key)

if __name__ == '__main__':
    # The default error rate is 0.01 and the default initial capacity is 100.
    # 这个是默认的配置， 初始大小为 100， 误差率默认为 0.01
    _test1(1000)
    _test1(10000)
    _test1(100000)
    _test2(500000)

```

## 五、性能测试

### 5.1 实验运行结果

```

插入结束... 花费时间: 0.0409s
size: 1000, 误判元素个数: 14, 误判率1.4000%
查询结束... 花费时间: 0.0060s
*****
插入结束... 花费时间: 0.1389s
size: 10000, 误判元素个数: 110, 误判率1.1000%
查询结束... 花费时间: 0.0628s
*****
插入结束... 花费时间: 0.5372s
size: 100000, 误判元素个数: 1419, 误判率0.4190%
查询结束... 花费时间: 0.4318s
*****
插入结束... 花费时间: 1.9484s
size: 500000, 误判元素个数: 152, 误判率0.0304%
查询结束... 花费时间: 2.2177s

```

### 5.2 结果分析

因为误判率的默认我们设置的为 0.01，而随着插入数量 size 的增大，可以看到误判率是逐渐下降的，所以可以得知，当样本数量足够大时，Bloomfilter 具有足够好的效果，方便我们

在这个大数据时代进行数据的处理。

## 六、应用场景和实例

在程序的世界中，布隆过滤器是程序员的一把利器，利用它可以快速地解决项目中一些比较棘手的问题。

如网页 URL 去重、垃圾邮件识别、大集合中重复元素的判断和缓存穿透等问题。

布隆过滤器的典型应用有：

- 数据库防止穿库。Google Bigtable, HBase 和 Cassandra 以及 Postgresql 使用 BloomFilter 来减少不存在的行或列的磁盘查找。避免代价高昂的磁盘查找会大大提高数据库查询操作的性能。
- 业务场景中判断用户是否阅读过某视频或文章，比如抖音或头条，当然会导致一定的误判，但不会让用户看到重复的内容。
- 缓存宕机、缓存击穿场景，一般判断用户是否在缓存中，如果在则直接返回结果，不在则查询 db，如果来一波冷数据，会导致缓存大量击穿，造成雪崩效应，这时候可以用布隆过滤器当缓存的索引，只有在布隆过滤器中，才去查询缓存，如果没有查询到，则穿透到 db。如果不在布隆器中，则直接返回。
- WEB 拦截器，如果相同请求则拦截，防止重复被攻击。用户第一次请求，将请求参数放入布隆过滤器中，当第二次请求时，先判断请求参数是否被布隆过滤器命中。可以提高缓存命中率。Squid 网页代理缓存服务器在 cache digests 中就使用了布隆过滤器。Google Chrome 浏览器使用了布隆过滤器加速安全浏览服务
- Venti 文档存储系统也采用布隆过滤器来检测先前存储的数据。
- SPIN 模型检测器也使用布隆过滤器在大规模验证问题时跟踪可达状态空间。

## 七、课程感悟

通过本次课程，我对于 Bloomfilter 的结构和性能有了更加深入的了解和认识，它在应对大数据处理时具有良好的性能，在本次实验当中，我在网上查找了很多的资料来了解 BloomFilter，但是本身我的其他语言基础学习的并不是很好，没有达到老师要求的实现多维数据属性表示和索引，只是实现了最基本的 Bloom Filter 结构的创建与运用，利用 python 的额 redisBloom 库进行了简单的实践，感到很遗憾。不过在这个过程中，还是学习到了很多关于数据存储的知识，更认识到了 BloomFilter 在面对传统的 hash 表相比，具有的优势。这是我在以后学习的过程中，应当努力的方向，虽然课程结束了，但是我对于大数据存储与管理的学习还没有结束，甚至可以说只是刚刚有了个开始！

## 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.