数据结构与算法（15）：布隆过滤器

一、引入

什么情况下需要布隆过滤器？我们先来看几个比较常见的例子：

- 字处理软件中，需要检查一个英语单词是否拼写正确
- 在 FBI，一个嫌疑人的名字是否已经在嫌疑名单上
- 在网络爬虫里，一个网址是否被访问过
- yahoo, gmail等邮箱垃圾邮件过滤功能

这几个例子有一个共同的特点： 如何判断一个元素是否存在一个集合中?

二、常规思路与局限

如果想判断一个元素是不是在一个集合里，一般想到的是将集合中所有元素保存起来，然后通过比较确定。链表、树、散列表（又叫哈希表，Hash table）等等数据结构都是这种思路。但是随着集合中元素的增加，我们需要的存储空间越来越大。同时检索速度也越来越慢。

- 数组
- 链表
- 树、平衡二叉树、Trie
- Map (红黑树)
- 哈希表

虽然上面描述的这几种数据结构配合常见的排序、二分搜索可以快速高效的处理绝大部分判断元素是否存在集合中的需求。但是当集合里面的元素数量足够大，如果有500万条记录甚至1亿条记录呢？这个时候常规的数据结构的问题就凸显出来了。

数组、链表、树等数据结构会存储元素的内容，一旦数据量过大，消耗的内存也会呈现线性增长，最终达到瓶颈。

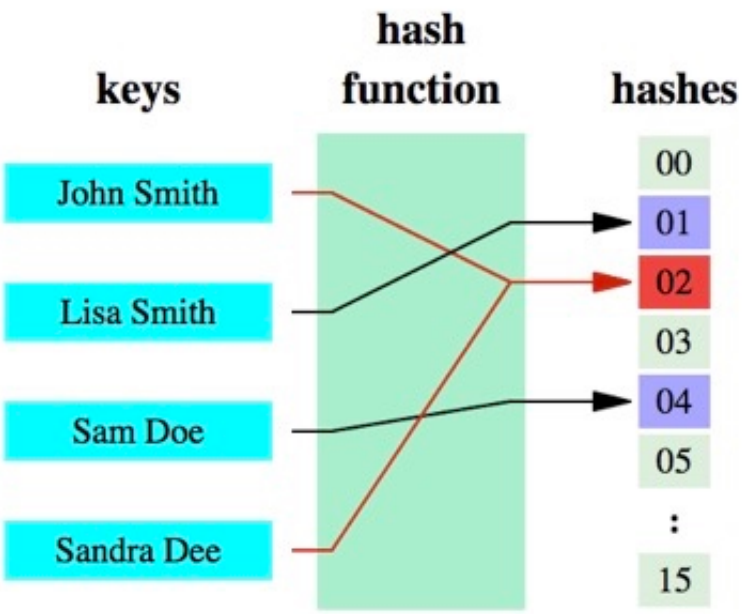
有的同学可能会问，哈希表不是效率很高吗？查询效率可以达到O(1)。但是哈希表需要消耗的内存依然很高。使用哈希表存储一亿 个垃圾 email 地址的消耗？哈希表的做法：首先，哈希函数将一个email地址映射成8字节信息指纹；考虑到哈希表存储效率通常小于50%（哈希冲突）；因此消耗的内存：8 * 2 * 1亿 字节 = 1.6G 内存，普通计算机是无法提供如此大的内存。这个时候，布隆过滤器（Bloom Filter）就应运而生。在继续介绍布隆过滤器的原理时，先讲解下关于哈希

函数的预备知识。

三、哈希函数

哈希函数的概念是：将任意大小的数据转换成特定大小的数据的函数，转换后的数据称为哈希值或哈希编码。

一个应用是Hash table（散列表，也叫哈希表），是根据哈希值 (Key value) 而直接进行访问的数据结构。也就是说，它通过把哈希值映射到表中一个位置来访问记录，以加快查找的速度。下面是一个典型的 hash 函数 / 表示意图：



可以明显的看到，原始数据经过哈希函数的映射后称为了一个个的哈希编码，数据得到压缩。哈希函数是实现哈希表和布隆过滤器的基础。

哈希函数有以下两个特点：

- 如果两个散列值是不相同的（根据同一函数），那么这两个散列值的原始输入也是不相同的。
- 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”（或者 “散列冲突”）。

缺点： 引用吴军博士的《数学之美》中所言，哈希表的空间效率还是不够高。如果用哈希表存储一亿个垃圾邮件地址，每个email地址 对应 8bytes, 而哈希表的存储效率一般只有50%，因此一个email地址需要占用16bytes. 因此一亿个email地址占用1.6GB，如果存储几十亿个email address则需要上百GB的内存。除非是超级计算机，一般的服务器是无法存储的。

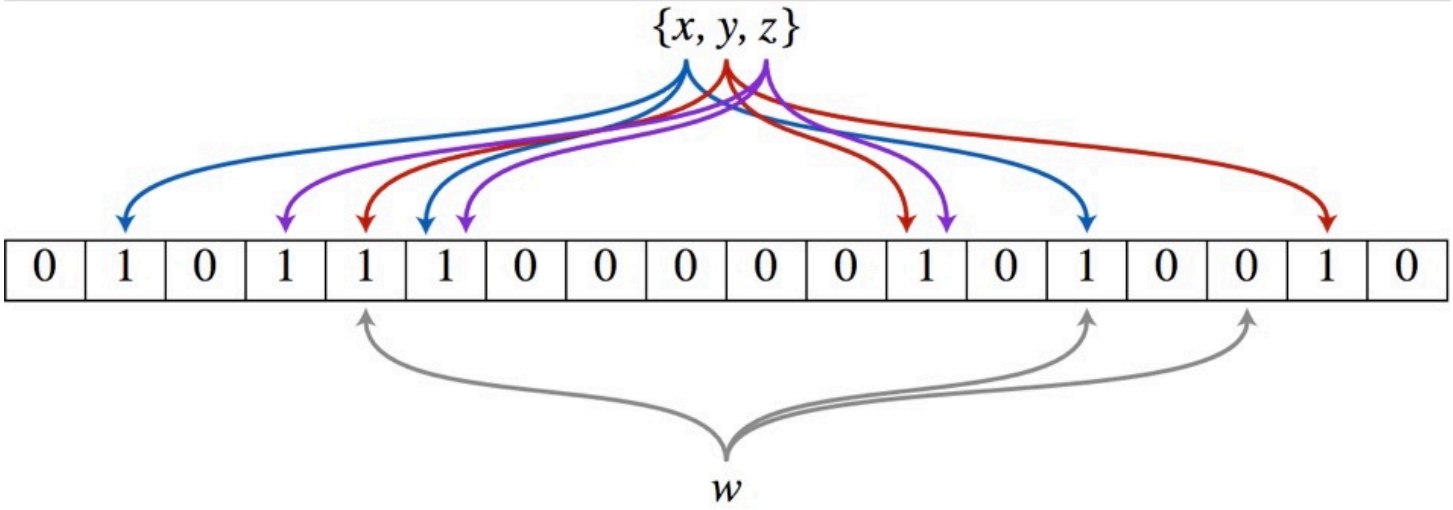
所以要引入下面的 Bloom Filter。

四、布隆过滤器（Bloom Filter）

布隆过滤器（英语：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。

4.1 原理

布隆过滤器（Bloom Filter）的核心实现是一个超大的位数组和几个哈希函数。假设位数组的长度为m，哈希函数的个数为k



以上图为例，具体的操作流程：假设集合里面有3个元素{x, y, z}，哈希函数的个数为3。首先将位数组进行初始化，将里面每个位都设置位0。

对于集合里面的每一个元素，将元素依次通过3个哈希函数进行映射，每次映射都会产生一个哈希值，这个值对应位数组上面的一个点，然后将位数组对应的位置标记为1。查询W元素是否存在集合中的时候，同样的方法将W通过哈希映射到位数组上的3个点。如果3个点的其中有一个点不为1，则可以判断该元素一定不存在集合中。反之，如果3个点都为1，则该元素可能存在集合中。

注意：此处不能判断该元素是否一定存在集合中，可能存在一定的误判率。可以从图中可以看到：假设某个元素通过映射对应下标为4，5，6这3个点。虽然这3个点都为1，但是很明显这3个点是不同元素经过哈希得到的位置，因此这种情况说明元素虽然不在集合中，也可能对应的都是1，这是误判率存在的原因。

4.2 添加与查询

> 布隆过滤器添加元素

- 将要添加的元素给k个哈希函数
- 得到对应于位数组上的k个位置
- 将这k个位置设为1

> 布隆过滤器查询元素

- 将要查询的元素给k个哈希函数
- 得到对应于位数组上的k个位置
- 如果k个位置有一个为0，则肯定不在集合中
- 如果k个位置全部为1，则可能在集合中

4.3 优点

> It tells us that the element either definitely is not in the set or may be in the set.

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；

4.4 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。

误判补救方法是：再建立一个小的白名单，存储那些可能被误判的信息。

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。

4.5 实例

可以快速且空间效率高的判断一个元素是否属于一个集合；用来实现数据字典，或者集合求交集。

> Google chrome 浏览器使用bloom filter识别恶意链接（能够用较少的存储空间表示较大的数据集合，简单的想就是把每一个URL都可以映射成为一个bit）

又如： 检测垃圾邮件

> 假定我们存储一亿个电子邮件地址，我们先建立一个十六亿二进制（比特），即两亿字节的向量，然后将这十六亿个二进制全部设置为零。对于每一个电子邮件地址 X，我们用八个不同的随机数产生器（F1,F2, ...,F8） 产生八个信息指纹（f1, f2, ..., f8）。再用一个随机数产生器 G 把这八个信息指纹映射到 1 到十六亿中的八个自然数 g1, g2, ...,g8。现在我们把这八个位置的二进制全部设置为一。当我们对这一亿个 email 地址都进行这样的处理后。一个针对这些 email 地址的布隆过滤器就建成了。

再如:

> A,B 两个文件，各存放 50 亿条 URL，每条 URL 占用 64 字节，内存限制是 4G，让你找出 A,B 文件共同的 URL。如果是三个乃至 n 个文件呢?

分析： 如果允许有一定的错误率，可以使用 Bloom filter，4G 内存大概可以表示 340 亿 bit。将其中一个文件中的 url 使用 Bloom filter 映射为这 340 亿 bit，然后挨个读取另外一个文件的 url，检查是否与 Bloom filter，如果是，那么该 url 应该是共同的 url（注意会有一定的错误率）。”

4.6 实现

下面给出python的实现，使用murmurhash算法

```
import mmh3
from bitarray import bitarray


# zhihu_crawler.bloom_filter

# Implement a simple bloom filter with murmurhash algorithm.
# Bloom filter is used to check wether an element exists in a collection, and it ha
s a good performance in big data situation.
# It may has positive rate depend on hash functions and elements count.



BIT_SIZE = 5000000

class BloomFilter:

    def __init__(self):
        # Initialize bloom filter, set size and all bits to 0
        bit_array = bitarray(BIT_SIZE)
        bit_array.setall(0)

        self.bit_array = bit_array
```

```
    def add(self, url):
        # Add a url, and set points in bitarray to 1 (Points count is equal to hash
funcs count.)
        # Here use 7 hash functions.
        point_list = self.get_postions(url)

        for b in point_list:
            self.bit_array[b] = 1

    def contains(self, url):
        # Check if a url is in a collection
        point_list = self.get_postions(url)

        result = True
        for b in point_list:
            result = result and self.bit_array[b]

        return result

    def get_postions(self, url):
        # Get points positions in bit vector.
        point1 = mmh3.hash(url, 41) % BIT_SIZE
        point2 = mmh3.hash(url, 42) % BIT_SIZE
        point3 = mmh3.hash(url, 43) % BIT_SIZE
        point4 = mmh3.hash(url, 44) % BIT_SIZE
        point5 = mmh3.hash(url, 45) % BIT_SIZE
        point6 = mmh3.hash(url, 46) % BIT_SIZE
        point7 = mmh3.hash(url, 47) % BIT_SIZE


        return [point1, point2, point3, point4, point5, point6, point7]
```