2019 级

《大数据存储与管理》课程

课 程 报 告

基于 Bloom Filter 的设计

姓　名 刘劲涛

学　号 U201915101

班　号 计算机 1906 班

日　期 2022.04.20

目　　录

一、Bloom Filter 基本介绍

Bloom filter 是由 Howard Bloom 在 1970 年提出的二进制向量数据结构，它具有空间和时间效率，被用来检测一个元素是不是集合中的一个成员(百度百科)。这只是对于 Bloom filter 的大概描述。

对于 Bloom filter，人们将他提出最初是因为哈希存储，而哈希存储的应用又是一种利用空间换取时间的策略。Bloom filter 的出现，其实还是因为人们对于一个问题算法进行优化时想出来的策略。这个问题就是：元素在集合中的存在性判断。

对于元素在集合中是否存在，判断方法有很多。最初的很简单，即将所有元素存储起来，放到一个相应的数据库里，在判断一个元素是否存在的时候，只需要线性遍历整个数据库，看一看数据库里是否有对应的元素。这个方法毋庸置疑是准确无误，且永远不会出错的。但是，这个方法对于时间及空间上的消耗非常巨大，因此，人们开始慢慢探索新的方法。

进一步的，人们开始用抗碰撞性强的安全的哈希函数对每一个元素进行相应的取值，然后进行相应的存储。注意，这里仅仅是进行一个取值，使得元素的 size 大小在哈希过后进行一个减小，来减小相应的空间消耗，但是对于时间上来说，需要一个 n 次的元素哈希值计算以计算哈希值。这也是将时间换取空间。

再后来，人们想出了散列表这种方法。在对于哈希取值之后，用求得哈希值作为元素存储的位置地址，在相应的查询之后，如果那处有值，那么就能证明为该哈希值的元素存在。这样看来，人们对于元素查询的时间复杂度由 O(n)变为 O(1)。

而这种方法，我们称之为 Bit-Map，而 Bloom Filter 更是种个典型的 Bit-Map 方法。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。Bloom Filter 的这种高效是有一定代价的：在判断一个元素是否属于某个集合时，有可能会把不属于这个集合的元素误认为属于这个集合（false positive）。因此，Bloom Filter 不适合那些“零错误”的应用场合。而在能容忍低错误率的应用场合下，Bloom Filter 通过极少的错误换取了存储空间的极大节省。

二、Bloom Filter 原理和分析

2.1 原理

Bloom Filter 其实就是多个哈希函数的对应。在人们判断元素是否存在这个问题时，在随着数据持续性增大的条件下，总会出现两个元素的哈希值相等的情况，在这时，我们就无法通过观测这两个元素的哈希值对应的值是否为 1 来判断这两个元素其中一个是否存在（可能只有一个存在，或者两个都存在）。而 Bloom Filter 就是多个哈希函数对应哈希值的翻版。假设我们有着 S={x1, x2,…,xn}这 n 个元素，又有着 k 个相互独立的哈希函数。那么我们对 S 中每一个元素都进行相应的哈希求值，那么最多的情况下我们就需要 k*n = m 个元素的数组来进行哈希值的存储。因此 m <= k*n 的。图 2.1 时 BloomFilter 的初始示意图。

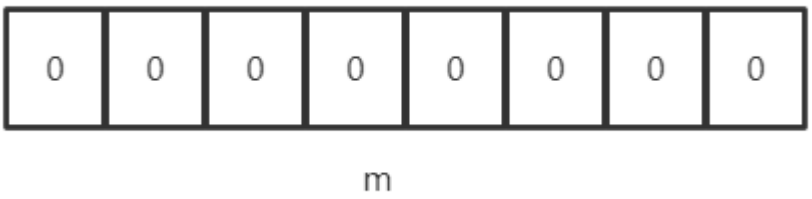


图 2.1 BloomFilter 初始示意图

在进行 k 个哈希值进行对应计算后，我们的 BloomFilter 中的值就要进行相应的值的改变。这里假设 k=3，如图 2.2。

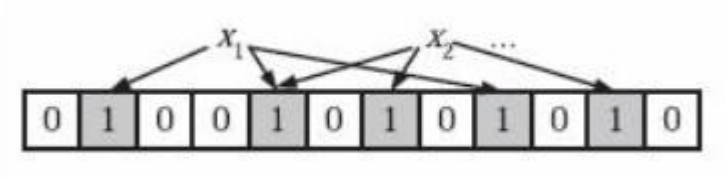


图 2.2

在图 2.2 中，x1 和 x2 中有着一处不同哈希计算可是最终得到的哈希值相等的情况，在 BloomFilter 中，位数组只对第一次出现的哈希值对应的位置进行置 1，之后出现这个哈希值我们都不需要其他操作，每有一处重复，我们的最大 kn 的大小都要进行减一。

但是尽管我们使用了多个哈希函数，还是有可能在元素数量巨大的情况下出现两个元素的 k 个哈希值一致的情况。所以，BloomFilter 并不能保证数据检测的完全正确。因此，我们有着这样的结论，如果我们在 BloomFilter 中找到一个元素对应的哈希值都存在，那我们只能认为他可能存在，即存在或者他是一个 false

positive。但是只要其中一个元素的对应哈希值不存在，那么这个元素肯定就不存在。如图 2.3。

2.3

2.3 中，y1 由于有两处哈希值对应为 0，那么它就肯定不存在，而 y2 对应的三个哈希值都为 1，我们也只能说它可能存在。

在海量的数据之中，利用 BloomFilter 具有非常有效的判断元素是否存在的效率，在允许一定的误差的条件下，BloomFilter 无疑是一个好的判断元素是否存在的选择。比如网络爬虫，我们判断一个网站的 URL 是否被爬取过，利用 BloomFilter 就有着很高的效率，因为我们允许有一些网站不被爬取。

2.2 分析

前面我们已经提到了，Bloom Filter 在判断一个元素是否属于它表示的集合时会有一定的错误率（false positive rate），下面我们就来估计错误率的大小。

首先我们对于 m 位的位数组，有一个哈希函数选中其中一位的概率为 1/m。而当集合 S={x1, x2,…,xn}的所有元素都被 k 个哈希函数映射到 m 位的位数组中时，这个位数组中某一位还是 0 的概率是：

$$p' = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}.$$

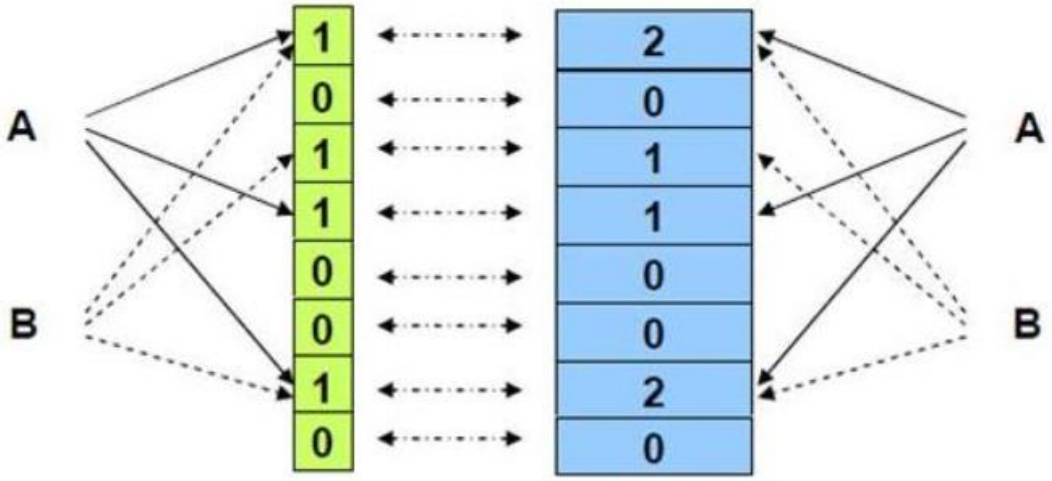
其中的不等式由相应的微积分的知识的计算得到，这里并不赘述。ρ 为位数组中 0 的比例，则 ρ 的数学期望 E(ρ)= p’。在 ρ 已知的情况下，要求的错误率（false positive rate）为：

$$(1-\rho)^k \approx (1-p')^k \approx (1-p)^k.$$

三、BloomFilter 的拓展

3.1 Counting Bloom Filter

标准的 Bloom Filter 只支持对应的插入数据和检测数据的操作。而在这个数据如此错综复杂的时代，没有对于数据的删除是万万不能的。而 Counting Bloom Filter 则是对于 Bloom Filter 的拓展。它与 Bloom Filter 其实并没有大的算法上的改变，实际上，他只是将每个元素经过哈希函数计算得到的哈希值对应位置进行相应的加 1 操作而已，相比于之前的置 1 之后不管的操作。而在删除一个元素时，也只需要将它对应的哈希值的每个位置减一即可。如图 3.1。



3.1

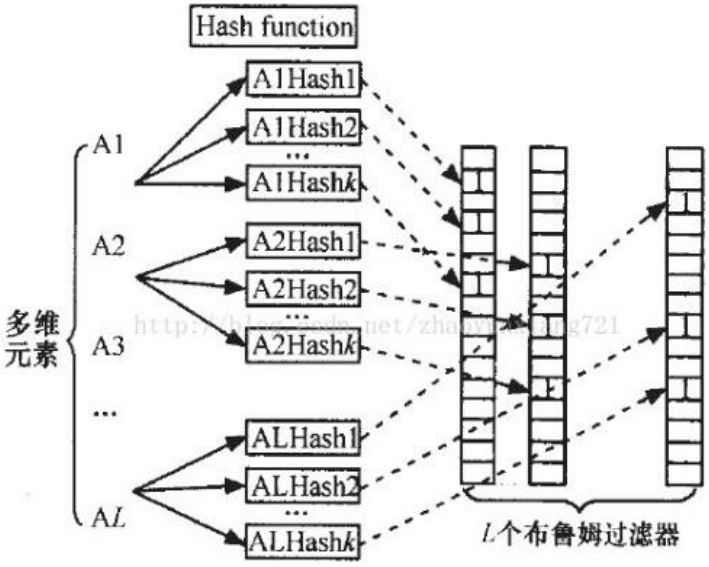
数据 A 和数据 B 对应的相同位置被置于 2，说明有两个元素对应这个位置。删除 A 时，并不会影响 B 存在这个事实。

而当数据增大时，我们的哈希值对应的位置的值到底该给多大的空间来供其使用呢，实际上，如果对于 n 个元素的数组，如果此数组里所有元素通过 k 个哈希函数计算都能得到同一个哈希值，那么该哈希值的数就要被置为 n，我们不可能对于每个位置都给 n 大小的数据位置。所以合适的哈希函数的选择也是十分重要的。通过相关研究，对于绝大多数的程序，4 位的大小足够。

3.2 Multi-dimension Bloom Filter

Multi-dimension Bloom Filter 由字面意义来看，其实就是多维 Bloom Filter。什么意思呢，对于一个元素本身。我们利用哈希值进行相应的对应时，如果这个元素有着不同的属性，比如一个人，他具有年龄，性别，名字，学号等属性。多种属性我们进行判断这个元素是否存在时，往往需要对多种属性进行判断。因此，对于一个属性复杂的元素，我们要对其每一个属性有着一个特地的位数组，用于

相应的判断该元素的这个属性是否存在，如果存在，那就对下一个属性进行判断。如果这个元素存在，那么其充分必要的条件，就是他的所有属性存在。MDBF 示意图。



MDBF 示意图

四、实验设计

这里我用 python 对于 BloomFilter 进行一个简单的实验模拟，使用语言为 python。

```
class BloomFilter(set):

    def __init__(self, size, hash_count):
        super(BloomFilter, self).__init__()
        self.bit_array = bitarray(size)
        self.bit_array.setall(0)
        self.size = size
        self.hash_count = hash_count

    def __len__(self):
        return self.size

    def __iter__(self):
        return iter(self.bit_array)

    def add(self, item):
        for seed in range(self.hash_count):
            index = mmh3.hash(item, seed) % self.size
            self.bit_array[index] = 1

        return self

    def __contains__(self, item):
        out = True
        for seed in range(self.hash_count):
            index = mmh3.hash(item, seed) % self.size
            if self.bit_array[index] == 0:
                out = False
```

简单标准的 BloomFilter 类其实除了初始化方法，只有添加元素和查询元素是否在位数组里面两个方法。add 很简单，其实就是对于一个元素进行相应的哈希操作，将目标位置置 1。而 contain 方法就是查询元素是否在其内部。只需要对元素进行哈希操作之后观测其所有哈希值是否为 1 即可，如果有一处不为 1，退出返回 false 即可。

```python
if __name__ == '__main__':
    bloom = BloomFilter(10000, 20)
    animals = ['dog', 'cat', 'giraffe', 'fly', 'mosquito', 'horse', 'eagle',
               'bird', 'bison', 'boar', 'butterfly', 'ant', 'anaconda', 'bear',
               'chicken', 'dolphin', 'donkey', 'crow', 'crocodile']
    # First insertion of animals into the bloom filter
    for animal in animals:
        bloom.add(animal)

    # Membership existence for already inserted animals
    # There should not be any false negatives
    for animal in animals:
        if animal in bloom:
            print('{} is in bloom filter as expected'.format(animal))
        else:
            print('Something is terribly went wrong for {}'.format(animal))
            print('FALSE NEGATIVE!')

    # Membership existence for not inserted animals
    # There could be false positives
    other_animals = ['badger', 'cow', 'pig', 'sheep', 'bee', 'wolf', 'fox',
                     'whale', 'shark', 'fish', 'turkey', 'duck', 'dove',
                     'deer', 'elephant', 'frog', 'falcon', 'goat', 'gorilla',
                     'hawk']
    for other_animal in other_animals:
        if other_animal in bloom:
            print('{} is not in the bloom, but a false positive'.format(other_animal))
        else:
            print('{} is not in the bloom filter as expected'.format(other_animal))
```

测试程序也很简单，先把一定的元素加入位数组，最后再将测试数据进行测试即可。

六、总结

这次对于 BloomFilter 的学习让我受益很多，我深刻地体会到人生学无止境这个道理。哈希表也是人类对于元素查找是否存在的一种智慧结晶，对其的使用节约了大量的时间消耗，在数据空间已经足够的今天，这无疑是一种十分合适妥当的替换。

參考文獻

- A Bloom Filter for High Dimensional Vectors Chunyan Shuai 1 , Hengcheng Yang 2 , Xin Ouyang 3,* and Zeweiyi Gong 2
- https://en.wikipedia.org/wiki/Bloom_filter
- F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, “Beyond B loom Filters: From Approximate Membership Checks to Approximate State Machine s,” Proc. ACM SIGCOMM, 2006.
- 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.