2019 级

《大数据存储系统与管理》课程

课 程 报 告

姓　名　赵英举

学　号　U201915093

班　号　物联网 1901 班

日　期　2022.04.20

计算机科学与技术学院

目录

一、 实验内容

基于 Bloom Filter 的多维数据属性表示和索引：设计算法实现 Bloom Filter 算法进行多维数据的索引，并对索引算法的查询延迟、空间开销、错误率等性能进行分析优化。

二、 实验原理

针对多维元素的表示和查询问题，目前存在一种多维布鲁姆过滤器(MDBF)解决方案。MDBF 采用和元素维数相同的多个标准布鲁姆过滤器组成，直接将多维元素的表示和查询分解为单属性值子集合的表示查询，元素的维数有多少，就采用多少个标准的布鲁姆过滤器分别表示各自对应的属性。进行元素查询时，通过判断多维元素的各个属性值是否都在相应的标准布鲁姆过滤器中来判断元素是否属于集合，如图 2.1 所示。

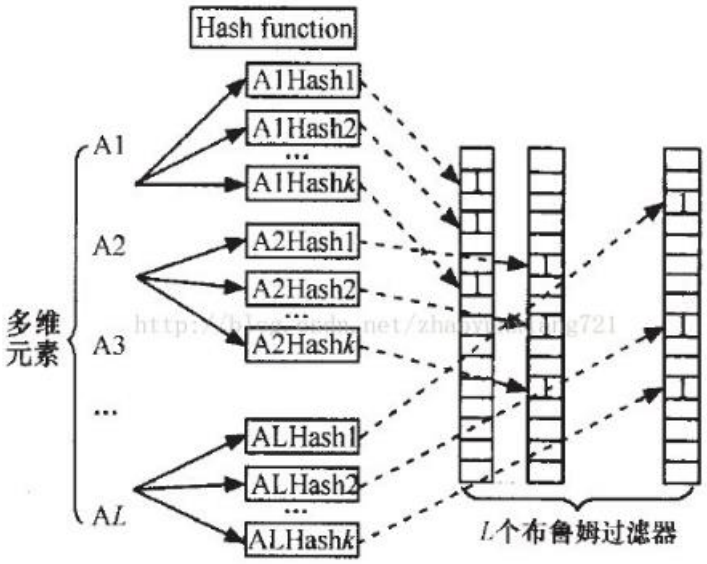

图 2.1 多维 Bloom Filter 判断原理

而 Bloom Filter 是一个判断集合中是否包含特定元素的算法，

他无需存储数据本身，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,首先将位数组进行初始化，将里面每个位都设置为 0。通过哈希将输入的字符串映射到数据的下标，然后把下标对应的值设置为 1，第二次相同字符串进行计算时也会被哈希函数映射到相同的下标则可以判断映射内容已经存在。

三、 结构设计

创建一个 m 位的位数组，先将所有位初始化为 0。然后选择 k 个不同的哈希函数。第 i 个哈希函数对字符串 str 哈希的结果记为 h（i，str），且 h（i，str）的范围是 0 到 m-1。如图 3.1 所示，将一个字符串经过 k 个哈希函数映射到 m 位数组中。

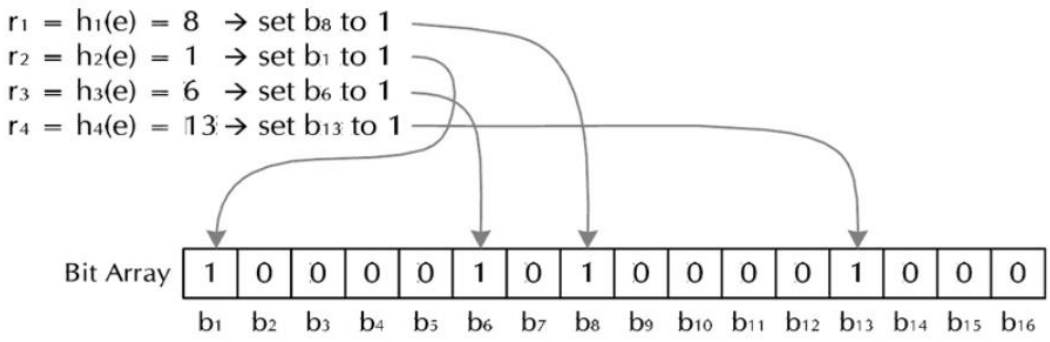

图 3.1 哈希映射到数组

从图中我们可以看到，字符串经过哈希函数映射成介于 0 到 m-1 之间的数字，并将 m 位位数组中下标等于这个数字的那一位置为 1 ，这样就将字符串映射到位数组中的 k 个二进制位了。

对于多维，设二维元素为{A1，A2)，其中 A1 和 A2 是 2 个不同的属性。使用 MDBF 需要 2 个标准布鲁姆过滤器 BF1、BF2 用来分别表

示属性 A1 和属性 A2。单属性域的布鲁姆过滤器向量都取作 8 bit。每次映射和查找的散列函数的个数为 2 个，2 个属性值映射的散列函数取一致，简单定义这 2 个散列函数为：h1(x)=x mod 8 和 h2(x)=(2x+3) mod 8。比如数据集合是{(9，7)，(11，9)}，需要查询的元素是(11，15)。表示集合之前，2 个向量都需要初始化。元素(9，7)插入后，第一维布鲁姆过滤器向量 BF1[1]和 BF1[5]置位，第二维布鲁姆过滤器向量 BF2[7]和 BF2[1]置位，两向量状态分别如图 3.2 第 2 行所示，那么元素(11，9)插入后的向量状态如图 3.2 第 3 行所示。

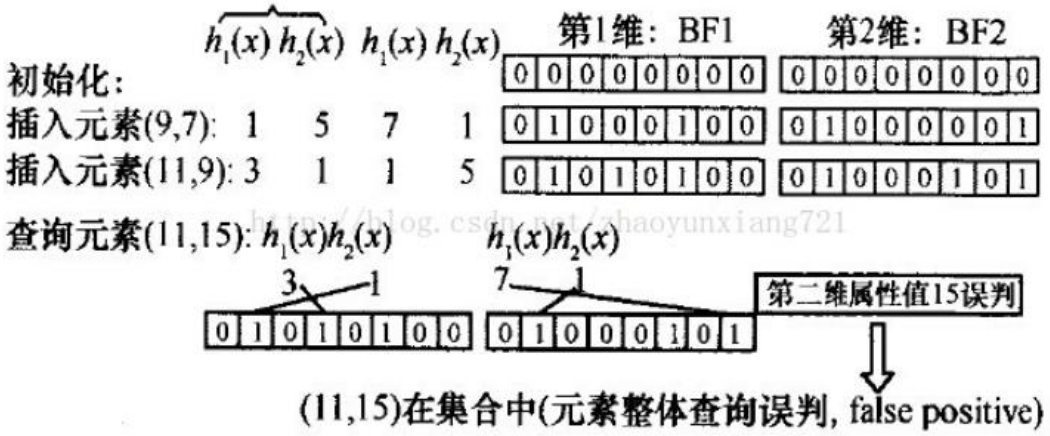

图 3.2 元素查询

查询(11，15)是否在集合中，首先检查 11 是否在第一个属性集合中，11 对应的 2 个散列地址 3、1，发现 BF1[3]，BF1[1]都已置位，说明 11 在属性 A 集合中。然后检查 15 的 2 个散列地址，发现 BF2[7]，BF2[1]也已经置位，说明 15 也在属性 A2 的集合中，得出(11，15)在集合中的错误结论。MDBF 算法在任何一维误判断时都会导致元素的误判断。

四、 实验过程

1. 实验步骤

首先使用 K 个相互独立、随机的散列函数将 x 映射到长度为 m 的位数组上，将散列函数得到的结果记作位置索引，将位数组改位置变为 1。我们用 k 个散列函数映射 k 个位置将对应的值变为 1。定义一个 HashMap 类，构造函数中，m 表示数组的位数，seed 表示种子的值，不用的散列函数要有不同的 seed，这样散列函数才不会碰撞。结构体如图 4.1 所示。

```
7   class SimpleHash(object):
8       def __init__(self, cap, seed):
9           self.cap = cap
10          self.seed = seed
11
12      def hash(self, value):
13          ret = 0
14          for i in range(len(value)):
15              ret += self.seed * ret + ord(value[i])
16          return (self.cap - 1) & ret
```

图 4.1 哈希散列函数

当 k 个散列函数在一起就实现了 Bloom Filter，我们对这几个散列函数指定相同的 m 不同的 seed。如图 4.2 所示。

```
19  class BloomFilter(object):
20      # 随机种子
21      SEEDS = [543, 460, 171, 876, 796, 607, 650, 81, 837, 545, 591, 946, 846, 521, 913, 636, 878, 735, 414, 372,
22               344, 324, 223, 180, 327, 891, 798, 933, 493, 293, 836, 10, 6, 544, 924, 849, 438, 41, 862, 648, 338,
23               465, 562, 693, 979, 52, 763, 103, 387, 374, 349, 94, 384, 680, 574, 480, 307, 580, 71, 535, 300, 53,
24               481, 519, 644, 219, 686, 236, 424, 326, 244, 212, 909, 202, 951, 56, 812, 901, 926, 250, 507, 739, 371,
25               63, 584, 154, 7, 284, 617, 332, 472, 140, 605, 262, 355, 526, 647, 923, 199, 518]
26
27      def __init__(self, capacity=100000000, error_rate=0.0000001, redis_con=None, key="bloomfilter"):
28          self.bit_size = math.ceil(capacity * math.log2(math.e) * math.log2(1 / error_rate))  # 所需位数
29          self.hash_time = math.ceil(math.log1p(2) * self.bit_size / capacity)  # 最少hash次数
30          self.memery = math.ceil(self.bit_size / 8 / 1024 / 1024)  # 占用多少M内存
31          self.block_num = math.ceil(self.memery / 512)  # 需要多少个512M的内存块,value的第一个字符必须是ascii码，最多有256个内存块
32          self.seeds = self.SEEDS[0:self.hash_time]
33          self.key = key
34          self.N = 2 ** 31 - 1
35          self.hash_func = [SimpleHash(self.bit_size, seed) for seed in self.seeds]
36          self.redis_con = redis_con
```

图 4.2 seed 部分结构

接下来实现插入和判断重复的方法。如图 4.3 所示。

```
    def is_contains(self, str_input):
        try:
            if not str_input:
                return False
            m5 = md5()
            m5.update(str_input)
            str_input = m5.hexdigest()
            ret = True
            name = self.get_key(str_input)
            for f in self.hash_func:
                loc = f.hash(str_input)
                ret = ret & self.redis_con.getbit(name, loc)
            return ret
        except Exception as e:
            raise

    def insert(self, str_input):
        try:
            m5 = md5()
            m5.update(str_input)
            str_input = m5.hexdigest()
            name = self.get_key(str_input)
            for f in self.hash_func:
                loc = f.hash(str_input)
                self.redis_con.setbit(name, loc, 1)
        except Exception as e:
            raise
```

图 4.3 判重和插入操作结构

然后进行存储数据的调用和时间、空间性能的测试。如图 4.4 所示。

```
#main
if __name__ == '__main__':
    bloom_filter_redis_conn_args = {
        "host": "127.0.0.1",
        "port": "6379",
        "db": 0
    }
    redis_con = redis.StrictRedis(connection_pool=redis.ConnectionPool(**bloom_filter_redis_conn_args))
    bloom_filter = BloomFilter(redis_con=redis_con)

    start_time = time.time()
    for i in range(6100000, 6101000):
        bloom_filter.insert(str(i).encode("utf-8"))
    print(time.time() - start_time)
    print(bloom_filter.is_contains("6100101".encode("utf-8")))

```

图 4.4 时间、空间等性能测试

最终运行结果如图 4.5 所示。

图 4.5 实验运行结果

2. 性能分析

MDBF 使用每个单独的 BF 来表示元素的每个单独的属性值，没有相应的结构将各属性值合成的元素表达出来。MDBF 分割了各个属性值属于元素一体的特点，仅通过单独判断元素的各个属性值是否在对应的子集合来进行元素的从属判断，不可避免会发生将不属于集合的元素误判为属于集合。对于{n，m，k，L}的 MDBF，判断元素是否从属集合，需要判断所有的属性值是否在对应的属性子集合，多维布鲁姆过滤器误判率为：

$$f^{MDBF}(m,k,n,l)=\prod_{i=1}^{L} f^{BF}(m,k,n)=(f^{BF}(m,k,n))^{L}$$

我们分析一下布隆过滤器的效率，在初始化时我们需要确认数组的大小,k 的数值,这些参数其实可以在给定容忍错误率后推算出来，由于哈希函数的构造，以及 k 个下标的技术都依赖于数组大小和 k 的数值，布隆过滤器在初始化时需要把数组 m 个比特设置为 0，以及构建 k 个哈希函数，因此时间复杂度为 O(k+m)。在本次实验中所花费的时间为 4.267839670187204s，和预期差不多。

Bloom Filter 解决冲突率高的有效方法就是使用多个 hash 函数把去重对象映射到多个位上。因为 hash 函数有一定的几率出现冲突，假设冲突的概率为 P1，其实 P1 是一个很小的概率，但是当待去重数

据量很大时 P1 也就跟着变大导致冲突变多。Bloom Filter 使用多个 hash 函数冲突概率分别为 P1,P2…Pn，不同的 hash 函数处理同一个去重对象是独立的，所以使用多个 hash 函数后的冲突概率通过乘法得到为 P1P2…Pn，这样冲突的概率就变得很小很小了。

五、 实验总结

在本实验中，Bloom Filter 会发生误判，但不会发生漏判。即如果某元素不在集合中，可能判断为在；但如果某元素在集合中，则一定会做出正确判断。比如结构设计中的例子两维元素(11，15)进行查询判断时，第二维 15 却不在第二维子集中，但是使用 BF2 误判断 15 在该子集中，此时就出现了两维元素由于一维的误判而导致元素整体的误判，成为 MDBF 的缺陷。但是进行实验测试时并没有出现误判的情况，且查找索引延迟较短只有 4 秒多，实验结果比较理想。