2020 级

《物联网数据存储与管理》课程

课 程 报 告

基于 Bloom Filter 的设计

姓　名　郭雯

学　号　U202015585

班　号　计算机 2010 班

日　期　2023.05.20

目　　录

一、课设目的

1. 分析 bloom filter 数据结构的设计；
2. 操作流程的分析，即如何保证和实现所提出的设计目标；
3. 进行 false positive 的理论分析；
4. 多维数据属性表示和索引（系数 0.8）
5. 实验性能的测试。

二、课设背景

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：由于可能出现哈希碰撞，不同元素计算的哈希值有可能一样，导致一个不存在的元素有可能对应的比特位为 1，这就是所谓“假阳性”（false positive）。相对地，“假阴性”（false negative）在 BF 中是绝不会出现的。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。

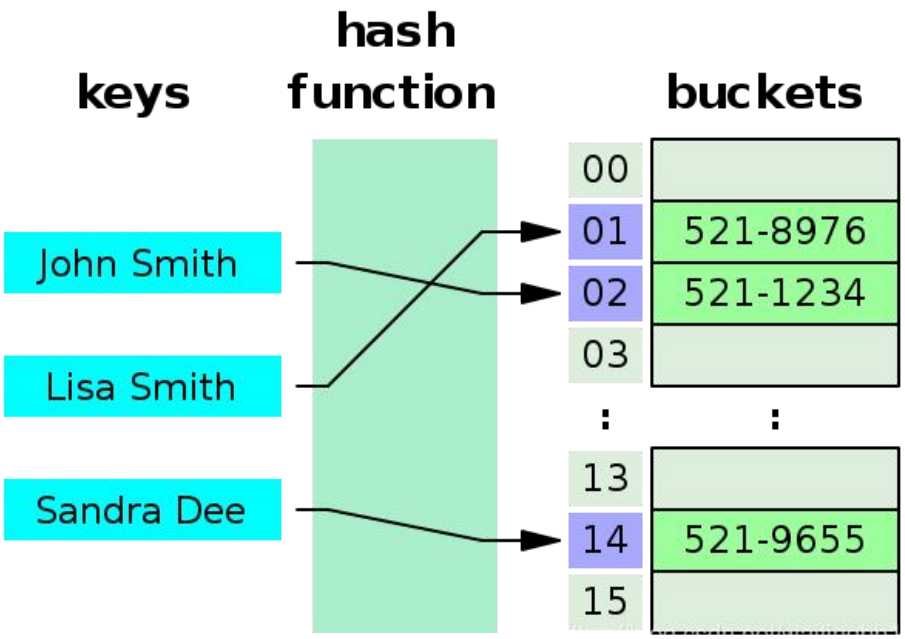
本次课程设计通过分析 Bloom Filter 的原理，最终提出对性能进行一定的优化的方法。

三、课设内容

3.1 Bloom Filter 原理

如果想判断一个元素是不是在一个集合里，一般想到的是将所有元素保存起来，然后通过比较确定。链表、树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，检索速度也越来越慢(O(n),O(logn))。这时候我们可以利用哈希表这种数据结构，基于哈希函数的特性，它在理想情况下(不发生哈希冲突)，检索速度可以达到 O(1)。

一张哈希表的示意图如下所示：



如果找到的哈希函数足够完美，那么理想状态下可以做到每个 key 对应一个唯一的 hashcode，但实际上往往会出现哈希冲突，即两个不同的 key 对应同一个 hashcode，即发生了碰撞。

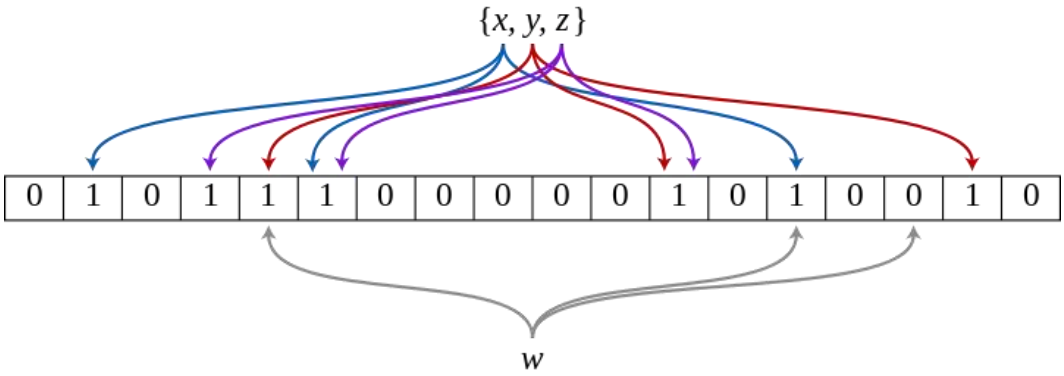
布隆过滤器使用了上面的思路，即利用哈希表这个数据结构，通过一个 Hash 函数将一个元素映射成一个位阵列（Bit array）中的一个点（每个点只能表示 0 或者 1），这样一来，我们只要看看这个点是不是 1 就知道在集合中有没有它了。

但是用 hash 表存储大数据量时，空间效率还是很低，当只有一个 hash 函数时，还很容易发生哈希碰撞。在哈希冲突的情况下，我们无法使用一个哈希函数来判断一个元素是否存在于集合之中，解决方法就是使用多个哈希函数，如果其中有一个哈希函数判断该元素不在集合中（元素经过 Hash 之后映射在位阵列中的点为 0），则不在。如果它们都判断存在，也有一定判断错误可能，不过这要比只用一个哈希函数来判断“一个元素存在于集合之中”的可靠性要高很多。这种多个 Hash 组成的数据结构就叫 Bloom Filter。

Bloom Filter 是基于一个 m 位的位阵列（b1,…bm），这些位阵列的初始值为 0。另外，还有一系列的 hash 函数（h1,…hk），这些 hash 函数的值域属于 1~m。

当有变量被加入集合时，通过 K 个映射函数将这个变量映射成位图中的 K 个点，把它们置为 1。

下图是一个 bloom filter 插入 x,y,z 并判断某个值 w 是否在该数据集的示意图（m=18,k=3）：



查找时，如果三个 hash 值对应的位向量都为 1，则判断可能在此数据集中。但是如果有任意 hash 值对应的位向量为 0，因此判断不在必定此集合中。

而事实上，查询所得位向量都为 1 的查询变量也可能不在此集合中，即会出现误报。显然，插入数据越多，1 的位数越多，错误率越大。

3.2 false positive 分析

易知由于哈希函数存在碰撞，而哈希碰撞导致的巧合会将不同的元素存储在相同的比特位上，这样会导致无法判断究竟是哪个输入产生的 1，因此会有一定的错误率。而这种错误率的大小是可以估计的。下面我们来估计这个数值。

为了简化计算，假设我们的哈希函数选择位数组中的比特时，都是等概率的。

在位数组长度 m 的 bloom filter 中插入一个元素，它的其中一个哈希函数会将某个特定的比特置为 1。因此，在插入元素后，该比特仍然为 0 的概率是：$1-\frac{1}{m}$ 。

现有 k 个哈希函数，并插入 n 个元素，则该比特仍然为 0 的概率是：$\left(1-\frac{1}{\mathrm{m}}\right)^{\mathrm{kn}}$。

易得它已被置为 1 的概率是：$1-(1-\frac{1}{\mathrm{m}})^{\mathrm{kn}}$。

即在插入 n 个元素后，如果查询变量不在集合中，那么被误认为在集合中的概率（也就是所有哈希函数对应的比特都为 1 的概率）为：$[1-(1-\frac{1}{\mathrm{m}})^{\mathrm{kn}}]^{k}$。

当 n 比较大时，这个概率近似为：$(1-\mathrm{e}^{-\mathrm{kn}/m})^{k}$。

最终我们发现，最优的 k 值为$\left(\frac{m}{n}\right) ln2$，此时错误率为$(0.6185)^{m/n}$。

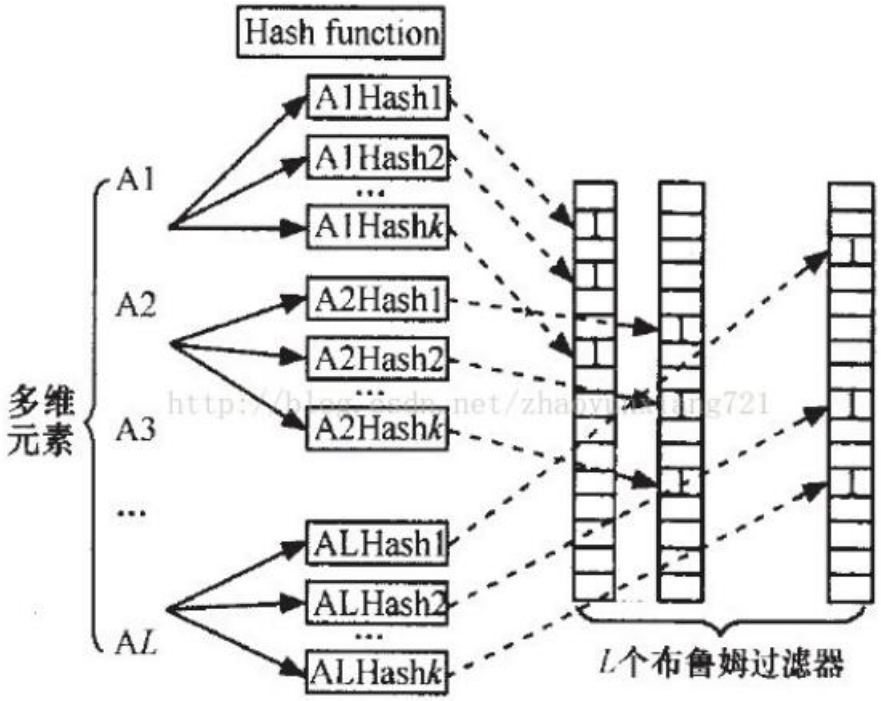
同时易得，在哈希函数的个数 k 一定的情况下：

（1）位数组长度 m 越大，错误率越低。

（2）已插入元素的个数 n 越大，错误率越高。

3.3 Bloom Filter 的多维数据属性表示

对于多重属性的元素，当我们判断该元素是否存在时，需要对这些属性分别进行判断。即每一种属性都对应一个位数组，假设数据有 n 维，则采用 n 组 hash 函数对每一维进行处理。将结果取 AND 运算，只有每个属性都为 1 时才能判断该种元素有可能存在，否则一旦有一个属性的位数组为 0，那么该元素必定不存在。同时易知，多维数据属性显然也存在 false positive。



四、实验设计

本实验实现了一个简易 Bloom Filter，并实现了插入元素、查询元素功能。代码如下：

```cpp
#include <bitset>
#include <iostream>
#include <string>

using namespace std;

// Bloom Filter类
class BloomFilter {
    private:
    bitset<1000000> bloom;  // 位向量
    int hash(string key, int seed) {  // 计算哈希值
        int hash = seed;
        for (int i = 0; i < key.length(); i++) {
            hash = hash * 131 + key[i];
        }
        return hash;
    }

    public:
    void insert(string key) {  // 插入元素
        int seed[8] = {3, 5, 7, 11, 13, 31, 37, 61};
        for (int i = 0; i < 8; i++) {
            int hashcode = hash(key, seed[i]);
            bloom.set(hashcode % bloom.size());  // 将对应位置设为1
        }
    }

    bool contains(string key) {  // 判断元素是否存在
        int seed[8] = {3, 5, 7, 11, 13, 31, 37, 61};
        for (int i = 0; i < 8; i++) {
            int hashcode = hash(key, seed[i]);
            if (!bloom.test(hashcode % bloom.size())) {  // 如果对应位置为0，则元素不存在
                return false;
            }
        }
        return true;
    }
};

int main() {
    BloomFilter bf;
    // 初始化 10000 条数据到过滤器中
    for (int i = 0; i < 10000; i++) {
        string s = "" + i;
        bf.insert(s);
    }
    // 判断值是否存在过滤器中
    int count = 0;
    for (int i = 0; i < 10000 + 100; i++) {
        string s = "" + i;
        if (bf.contains(s)) {
            count++;
        }
    }
    cout<<"已匹配数量 " <<count;
    return 0;
}
```

五、性能测试

5.1 误判率

改变数据大小，通过对比发现，当哈希函数个数为 8 时：

数据量为 1000 时误判率约为 0.278；

数据量为 10000 时误判率约为 0.083；

数据量为 100000 时误判率约为 0.028；

观察结果可知，当插入数据量不断增大的时候，误判率有所降低，易知 Bloom Filter 很适合大数据时代的使用。

六、课程总结

通过本次课程设计，我对于 Bloomfilter 的结构和性能有了更加深入的了解和认识。同时我认识到了 Bloomfilter 广大的应用场景以及它的优缺点。比如它在字典存储、数据库、分布式缓存、P2P/Overlay 网络应用等方面都有很丰富的应用。

在本次实验中，由于除了在课堂上，以前并没有接触过 BloomFilter，所以翻阅理解资料用了我很多时间，但是收获也是很客观的。但最终也只是实现了基础的要求，我感到十分遗憾，希望日后有时间可以进一步学习相关内容，进一步体会这些优势。

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.