基于 Bloom Filter 的设计

实 验 报 告

姓　名　程丁丁

学　号　U201813729

班　号　CS1807

日　期　2022.04.20

目　　录

一、实验目的

1. 分析 bloom filter 的设计结构和操作流程；
2. 理论分析 false positive；
3. 多维数据属性表示和索引（系数 0.8）
4. 实验性能查询延迟，空间开销，错误率的分析。

二、实验背景

随着社会对信息存储需求的增长，大规模存储系统的应用越来越广泛，存储容量也从以前的 TB（Terabyte）级上升到 PB（Petabyte）级甚至 EB（Exabyte）级。查找和处理文件变得越来越困难。现有的基于层次目录树结构的数据存储系统的扩展性和功能性不能有效地满足大规模文件系统中快速增长的数据量和复杂元数据查询的需求。为了有效地处理这些快速增长的数据，迫切需要提供快速有效的数据管理系统来帮助用户更好的理解和处理文件。

元数据（metadata）是关于数据的数据，是关于信息资源的形式、主要内容、数据的特征和属性、数据的使用者、使用和修改记录、存放位置等信息的集合。在文件系统中元数据用于描述和索引文件，例如超级块信息，记录全局文件信息如文件系统的大小，可用空间等；索引节点信息，记录文件类型、文件的链接数目、用户 ID、组 ID、文件大小、访问时间、修改时间、文件使用的磁盘块数目等；目录块信息，记录文件名和文件索引节点号等。有效的管理这些元数据并提供各种查询接口，能帮助用户更好的理解和处理数据。

一般来讲，尽管存储系统中元数据的数据量远小于整个系统的存储容量，文件系统中元数据占用的空间往往不到 10%，但元数据操作是整个文件系统操作的 50%-80%，所以元数据的高效管理十分必要。

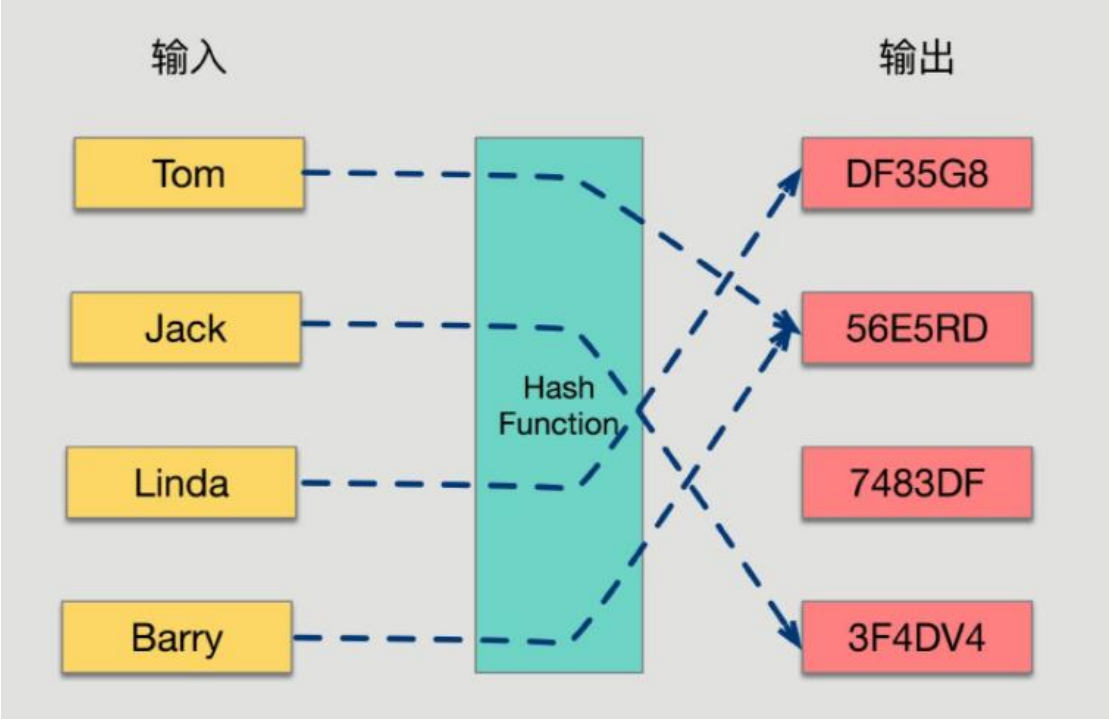
三、实验内容

3.1 Bloom Filter 简介

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。主要用于判断一个元素是否在一个集合中。

通常我们会遇到很多要判断一个元素是否在某个集合中的业务场景，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也会呈现线性增长，最终达到瓶颈。同时检索速度也越来越慢，上述三种结构的检索时间复杂度分别为 O(n)，

哈希函数的概念是：将任意大小的输入数据转换成特定大小的输出数据的函数，转换后的数据称为哈希值或哈希编码，也叫散列值。下面是一幅示意图 O(logn)，O(1)。



所有散列函数都有如下基本特性：

- 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。这个特性是散列函数具有确定性的结果，具有这种性质的散列函数称为单向散列函数。

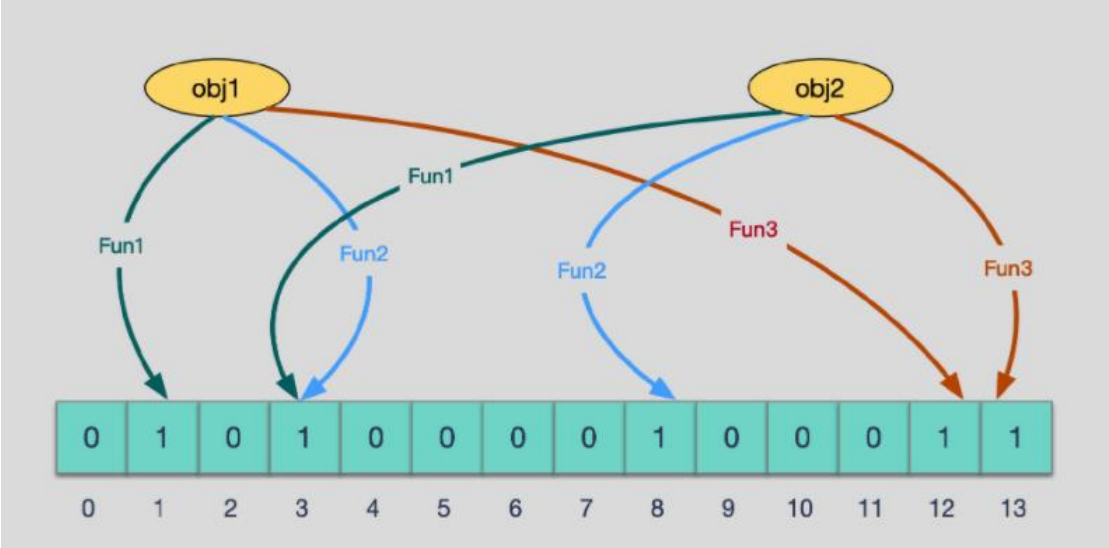
- 散列函数的输入和输出不是唯一对应关系的，如果两个散列值相同，两个输入值很可能是相同的，但也可能不同，这种情况称为“散列碰撞（collision）”。

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。

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：

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（假定有两个变量都通过 3 个映射函数）。



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

3.2 false positive 概率推导

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。

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。(比如上图中的第 3 位)

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 k 次 Hash 操作后，插入 n 个元素后，该位都仍然为 0 的概率是：$\left(1-\frac{1}{\mathrm{m}}\right)^{\mathrm{kn}}$

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：

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

根据推导可得，Hash 函数个数选取最优数目$k=\left(\frac{m}{n}\right) ln2$

此时 false positives 的概率为$f=(0.6185)^{m/n}$

3.3 Bloom Filter 的多维数据属性表示

多维数据属性也很常见，如，有具有两个属性的桌子，其中一个桌子是小的红色的，另一个桌子是大的蓝色的，现在将这些对应的属性对应的 hash 映射位置一，当再来一张大的红色的桌子的时候，由于对应的属性全为 1，所以会被误

判在该集合中是存在的，这就导致了多维数据 false positive 的存在。

标准的 Bloom Filter 不支持多维元素成员的查询，由 Guo 提出的 MDBF 算法核心思想是针对多维数据集的每一个属性，构建独立的布隆过滤器，当查询元素时，通过多维元素的各个属性是否都存在于相应的布隆过滤器中，来判断元素是否属于集合，CMDBF 在 MDBF 的基础上新增了一个用于表示元素整体的联合布隆过滤器 CBF 将各属性的表示和查询作为第一步，第二步联合元素所有的属性域，利用 CBF 完成元素整体的表示和查询。BFM 通过在每个维度上保存一个位向量以构造布隆过滤器，并基于独立属性笛卡尔乘积构造位矩阵，用于全属性查询，从根本上消除了组合误差率。

多维元素成员查询典型解决方案包括表索引、树索引及混合索引结构。

目前广泛应用的混合索引结构是将布鲁姆过滤器与树索引进行融合。Adina 等在 2015 年提出了一种混合索引结构 Bloofi，将 B+树和布鲁姆过滤器进行层次化融合，并在此基础上实现位级别的并行化算法 Flat-Bloofi，有效解决了多维数据集的元素查询问题。为了解决云存储环境下的非 KEY 字段查询问题，BF-Matrix 提出了一种层次化索引结构，结合了布鲁姆过滤器和 B+树，极大缩减了元素查询路径，并采用基于规则的索引更新机制，有效降低布鲁姆过滤器的假阴性概率。

四、实验设计

```
public class MyBloomFilter {

    /**
     * 一个长度为10 亿的比特位
     */
    private static final int DEFAULT_SIZE = 256 << 22;

    /**
     * 为了降低错误率，使用加法 hash 算法，所以定义一个 8 个元素的质数数组
     */
    private static final int[] seeds = {3, 5, 7, 11, 13, 31, 37, 61};

    /**
     * 相当于构建 8 个不同的 hash 算法
     */
```

```
    private static HashFunction[] functions = new
HashFunction[seeds.length];

    /**
     * 初始化布隆过滤器的 bitmap
     */
    private static BitSet bitset = new BitSet(DEFAULT_SIZE);

    /**
     * 添加数据
     *
     * @param value 需要加入的值
     */
    public static void add(String value) {
        if (value != null) {
            for (HashFunction f : functions) {
                //计算 hash 值并修改 bitmap 中相应位置为 true
                bitset.set(f.hash(value), true);
            }
        }
    }

    /**
     * 判断相应元素是否存在
     * @param value 需要判断的元素
     * @return 结果
     */
    public static boolean contains(String value) {
        if (value == null) {
            return false;
        }
        boolean ret = true;
        for (HashFunction f : functions) {
            ret = bitset.get(f.hash(value));
            //一个 hash 函数返回 false 则跳出循环
            if (!ret) {
                break;
            }
        }
        return ret;
    }

    /**
     * 模拟用户是不是会员，或用户在不在线。。。
```

```
     */
    public static void main(String[] args) {

        for (int i = 0; i < seeds.length; i++) {
            functions[i] = new HashFunction(DEFAULT_SIZE, seeds[i]);
        }

        // 添加1亿数据
        for (int i = 0; i < 100000000; i++) {
            add(String.valueOf(i));
        }
        String id = "123456789";
        add(id);

        System.out.println(contains(id));   // true
        System.out.println("" + contains("234567890"));  //false
    }
}

class HashFunction {

    private int size;
    private int seed;

    public HashFunction(int size, int seed) {
        this.size = size;
        this.seed = seed;
    }

    public int hash(String value) {
        int result = 0;
        int len = value.length();
        for (int i = 0; i < len; i++) {
            result = seed * result + value.charAt(i);
        }
        int r = (size - 1) & result;
        return (size - 1) & result;
    }
}
```

五、性能测试

下面讨论默认相关系数 a=0.8

5.1 误判率

全属性查询对比 MDBF、CMDBF、CBF 索引结构，随着相关系数增大，误判率趋近于 1。MDBF 的误判率＞CMDBF＞CBF，其中 CBF 最佳，在10^{-3}数量级，MDBF 在10^{-1}数量级。

5.2 空间开销

在相同内存条件下，BFM 误判率较高，CMDBF 仍然保持较低的误判率，原因是 BFM 存在较高的空间需求导致其位冲突率较高。。

5.3 查询延迟

CMF 查询延迟＜MDBF＜CMDBF，但差距不大。

六、课程总结

本课程是一门基础性课程，学习了很多前沿知识。对本次实验相关的 Bloom Filter，最直观的感受是启发了我的思考，此前没想到哈希表与存储还有这么多值得研究的内容，确实印证了学无止境，或者说其实才刚刚开始而已。

首先了解了 Bloom Filter 的前世今生以及面临的问题，在学习完毕之后回首来看，其实一切也来得顺理成章。对于多维属性，也了解了很多富有智慧的设计。具体代码实现方面仅仅实现了一个最为简单的 bloom filter，只略作学习使用，没有太多现实意义。之后简单比较了一下性能，这方面各有千秋。

虽然本实验只能算做入门，但也为之后的学习研究打下了基础，希望日后有机会深入学习。也感谢老师们的辛苦付出。

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.

[8]Yong WANG,Xiao-chun YUN,ANGShu-peng WANG,Xi WANG. CBFM:cutted Bloom filter matrix for multi-dimensional membership query[J]. Journal on Communications, 2016, 37(3): 139-147.

[9] CHENG X, LI H, WANG Y, et al. BF-matrix: a secondary index for the cloud storage[M]. Springer International Publishing, 2014. 384-396.