2020 级

《物联网数据存储与管理》课程

课 程 报 告

姓　名 杨晨

学　号 U202015596

班　号 CS2010

日　期 2023.05.29

目　　录

一、目标概述

1. 初步了解 Bloom Filter，明白其结构和原理。
2. 基于理论，分析 Bloom Filter 的 false positive 的概率。
3. 基于 Bloom Filter 的操作流程，实现一个能保证和实现所提出的设计目标的建议 Bloom Filter。
4. 基于传统 Bloom Filter，实现多维度属性数据和索引的 Bloom Filter。

二、初入 Bloom Filter

2.1 布隆过滤器简介

Bloom Filter（布隆过滤器）是一种高效的数据结构，用于判断一个元素是否属于一个集合。它使用位数组（Bit Array）和一系列哈希函数来实现快速的成员查询。

Bloom Filter 最初由布隆（Burton H. Bloom）于 1970 年提出，广泛应用于计算机科学和软件工程领域，特别适用于那些对查询速度要求高、可以容忍一定程度误判率的场景。

Bloom Filter 的核心思想是通过多个哈希函数将元素映射到位数组的不同位置上，每个位置用一个二进制位表示。当元素被添加到 Bloom Filter 中时，对应的位数组位置被置为 1。在查询阶段，如果一个元素经过多个哈希函数的映射后，对应的位数组位置都为 1，则可以认为该元素可能存在于集合中；否则，可以确定该元素一定不存在于集合中。

Bloom Filter 具有以下特点：

1. 快速查询：由于 Bloom Filter 只需要进行位数组的读取操作，而不涉及具体元素的存储和比较，所以查询速度非常快。
2. 节省空间：Bloom Filter 只需要存储位数组和哈希函数，不需要存储具体元素，因此占用的空间通常比较小。
3. 可能的误判：由于多个元素可能被映射到同一个位数组位置上，所以在查询结果为存在时，可能存在一定的误判率。

由于 Bloom Filter 的误判率是可以控制的，可以通过调整位数组的大小和哈希函数的数量来平衡误判率和空间占用。同时，Bloom Filter 也支持动态添加和删除元素的操作，但删除操作会导致误判率的增加。

Bloom Filter 在许多实际应用中发挥着重要作用，例如：

- 网络缓存：用于判断请求的数据是否已经缓存，避免重复下载；
- 分布式系统：用于减少网络传输和存储开销，过滤掉无效的请求；
- 数据库系统：用于快速判断某个元素是否已经存在于数据库中，从而避免昂贵的查询操作。

需要注意的是，Bloom Filter 不适合对数据进行精确的查询和检索，它只能提供可能的存在性判断。在关键应用场景中，需要综合考虑误判率和性能需求，选择合适的参数配置和数据结构。其结构示意图如图 1。

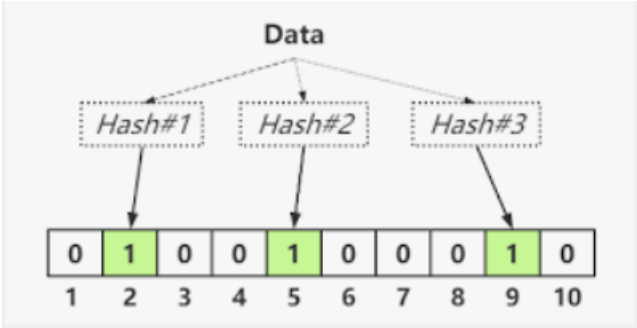

图 1 布隆过滤器示意图

2.2 布隆过滤器原理

Bloom Filter 的原理相对简单，其核心思想是使用位数组和多个哈希函数来实现快速的成员查询。下面是 Bloom Filter 的工作原理：

1. 初始化：创建一个位数组（Bit Array）并将所有位都置为 0。同时确定需要使用的哈希函数数量和参数。

2. 添加元素：当一个元素要被添加到 Bloom Filter 中时，将该元素输入到多个哈希函数中，并根据哈希函数的输出结果将位数组中对应的位置置为 1。

3. 查询元素：当需要查询一个元素是否存在于 Bloom Filter 中时，将该元素输入到相同的多个哈希函数中，并检查位数组中对应的位置是否都为 1。如果所有位置都为 1，则可以认为该元素可能存在于集合中。如果有任何一个位置为 0，则可以确定该元素一定不存在于集合中。

需要注意的是，Bloom Filter 存在一定的误判率，即在查询结果为存在时，实际上该元素可能并不存在于集合中。这是由于多个元素可能被映射到同一个位数组位置上，从而导致冲突。

误判率受到位数组的大小和哈希函数数量的影响。通过增加位数组的大小和增加哈希函数的数量，可以降低误判率，但同时也会增加空间占用和查询时间。因此，在实际应用中需要根据具体需求和资源限制进行权衡。

总结起来，Bloom Filter 的原理可以简单概括为：通过多个哈希函数将元素映射到位数组的不同位置上，利用位数组的位来表示元素的存在与否，从而实现快速的成员查询。当一个元素被加入集合时，通过 K 个哈希函数将这个元素映射成 K 个不同的索引，然后将 bit 数组中对应的 K 个不同索引的值置为 1。检索时，我们只要看看这些点是不是都是 1 就（大约）知道集合中有没有它了：如果这些点有任何一个 0，则被检元素一定不在；如果都是 1，则被检元素很可能在。这就是布隆过滤器的基本思想。

2.3 布隆过滤器的假阳性

过滤器的假阳性（False Positive）是指在查询阶段，过滤器错误地将一个不存在于集合中的元素判断为存在于集合中的情况。

假阳性是由于 Bloom Filter 的特性所致，即多个元素可能映射到同一个位数组位置上，导致冲突。当查询一个元素时，如果该元素经过多个哈希函数的映射后，对应的位数组位置都为 1，就会被误判为存在于集合中，即出现了假阳性。

假阳性的概率受到多个因素的影响，包括位数组的大小、哈希函数的数量、哈希函数的质量以及已添加元素的数量等。增加位数组的大小和增加哈希函数的数量可以降低假阳性的概率，但同时也会增加空间占用和查询时间。

可以通过以下方式来减少假阳性的概率：

1. 增加位数组的大小：增加位数组的长度可以减少冲突的可能性，从而降低假阳性的概率。
2. 增加哈希函数的数量：使用更多的哈希函数可以增加位数组中被置为 1 的位置数量，减少假阳性的概率。
3. 选择良好的哈希函数：使用高质量的哈希函数可以减少冲突，降低假阳性的概率。
4. 控制已添加元素的数量：当已添加的元素数量超过一定阈值时，假阳性的概率会增加。可以根据实际需求定期清空过滤器或重新构建过滤器来控制假阳性的影响。

需要注意的是，虽然 Bloom Filter 可以提供高效的成员查询和节省空间的优势，但假阳性是其固有的特性。因此，在使用 Bloom Filter 的场景中，需要权衡假阳性的概率和性能需求，确保误判率在可接受范围内。假设一个元素 A 已经在集合中，其哈希之后对应的 bit 位是 1，3，6，而这时我想要查询的元素 B（不

在集合中），其哈希之后对应的 bit 位也是 1，3，6，在这种情况下，布隆过滤器判断元素 B 在这个集合中，因此会产生假阳性（false positive）。

三、Bloom Filter 理论分析

3.1 False positive 分析

布隆过滤器（Bloom Filter）是一种空间有效的概率型数据结构，它用于测试一个元素是不是集合的一部分。但是，布隆过滤器存在一定的错误率，这就是假阳性（False Positive）。也就是说，有时候布隆过滤器会错误地判定一个元素在集合中，虽然实际上它并不在集合中。

假设布隆过滤器由 m 位的比特数组以及 n 个元素组成，分析如何选定哈希函数的个数 k，能够使得布隆过滤器的假阳性概率最低。（假设对于哈希函数而言，映射到 0 至 m-1 的概率是相等的。

当在进行一次元素插入集合中的时候，对于任一一个 bit 位而言，被置 1 的概率是

$$\frac{1}{m}$$

因此没有被置为 1 的概率是

$$1-\frac{1}{m}$$

那么在进行 k 次不同 hash 之后，依旧没有被置为 1 的概率是

$$\left(1-\frac{1}{m}\right)^k$$

插入 n 个元素之后，对于任一一个 bit 位，为 1 的概率为

$$1-\left(1-\frac{1}{m}\right)^{kn}$$

因此对于一个不存在与集合中的元素，被误判断为在集合中的概率为

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

使得错误率最小，对于给定的 m 和 n，当 $k=\frac{m}{n}*ln2$ 时候取值最小。

3.2 Bit 数组大小分析

在给定 n（集合中元素的个数）和错误率（最优函数个数 k 的的错误率）的情况下，位数组 M 的大小计算，在最优 k 的情况下

$$p=\left(1-e^{-\frac{kn}{m}}\right)^k$$

经过简单的代入和转换可以得到

$$m=-\frac{(n*lnp)}{(ln2)\text{^}2}$$

四、Bloom Filter 操作流程

传统的布隆过滤器的主要操作包括了增加元素，查询元素，以及清空过滤器。而主要组成部分包括 k 个哈希函数，以及一个 m 位的 bit 数组。

插入操作：

给定要插入布隆过滤器的元素。

使用布隆过滤器的所有哈希函数对这个元素进行哈希运算，得到一组哈希值。布隆过滤器可以使用多个哈希函数，通常为 2 个或以上。

对于每一个哈希值，将其作为索引，将布隆过滤器位数组（通常是一个比特数组）中对应的位设置为 1。

这就完成了元素的插入操作。布隆过滤器不实际存储元素本身，而是存储了元素经过哈希运算后的信息。

流程如图 2 所示。

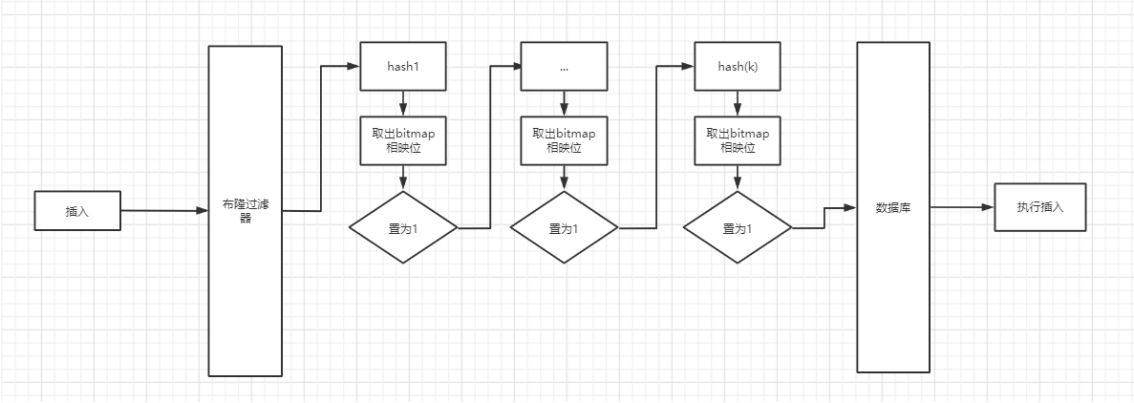

图 2 插入操作流程

查询操作：

给定要查询的元素。

同样使用布隆过滤器的所有哈希函数对这个元素进行哈希运算，得到一组哈希值。

对于每一个哈希值，检查布隆过滤器位数组中对应的位是否为 1。如果所有对应的位都是 1，那么布隆过滤器就会返回"可能在集合中"。如果有任何一位不是 1，那么布隆过滤器就会返回"肯定不在集合中"。

流程图如图 3 所示。

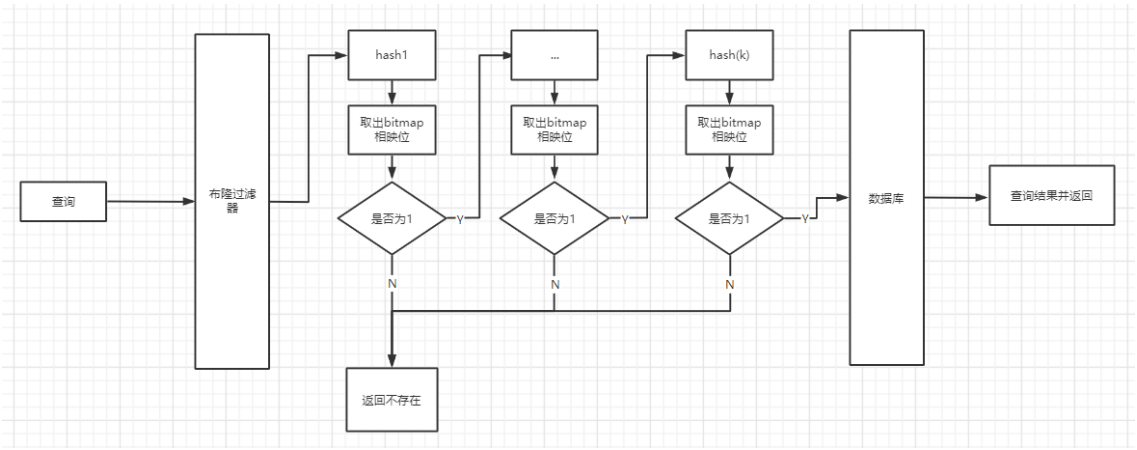

图 3 查询操作流程

五、Bloom Filter 简易实现

5.1 结构设计

设计核心思想就是对于一个 Bloom Filter 类，我们根据给定的容纳的最大元素个数以及最大的错误率，运用第四节中的两个公式来计算出 bit 数组的大小以及 hash 函数的个数，来使得 false positive 最小。

而 Bloom Filter 类中有的核心方法有：加入元素，查询元素，清空过滤器这三个方法。

因此可以根据上述思想来构造一个建议的 Bloom Filter，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;
}
```

5.2 测试结果

这里我随机生成了 20w 个不重复的字符串作为测试用例，其中 15w 用作存入数据库的数据，剩余 5w 作为查询数据，值得注意的是，在存入数据的时候，bloom filter 会先判断数据是否在数据库中，如果再，则不存入，这里也会有一个 false positive 的存在；而对于查询数据，本来这 5w 的数据就不再数据库中，因此只要知道过滤器判断多少个元素在存储中，即可计算出 false positive 的值。而另外一点，false positive 的值也同样取决于 hash 函数的选择以及 hash 种子的选择。以下是几组测试数据。

图 4 简易 bloom filter 测试结果

六、课程感悟

总的来说，布隆过滤器是一个有趣且独特的数据结构，它的概率特性和空间效率使其在处理大数据和资源有限的环境中非常有用。然而，也需要明白并接受其假阳性的限制，以及选择和优化适当的参数以达到最佳效果。

这次的课程报告对我产生了深远的影响。它不仅驱使我深入研读了许多国际学术论文，从而提升了我逐渐消退的英语水平，更为重要的是，它让我对大数据存储和处理有了更深入的理解。在我们所生活的这个时代，单日产生的信息量已经远超过过去几百年甚至几千年的信息总量。如何有效并高效地管理和存储这些数据，这正是我们当前面临的巨大挑战。通过编写这份课程总结报告，我对布隆过滤器有了深度的理解。然而，我认为课程中所学习到的知识始终仅是皮毛。从个人发展的视角来看，如果我决定投身于大数据相关行业，我不应仅满足于课堂上的学习，而应积极实践，亲自编写代码。唯有通过亲手创造，我才能迅速提升自己的技能，并真正做到学以致用。

参考文献

[1] 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.

[2] 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.

[3] D. Guo, J. Wu, H. Chen, and X. Luo, “Theory and Network Application of Dynamic Bloom Filters,” Proc. IEEE INFOCOM, 2006.