2020 级

基于 Bloom Filter 的设计

实 验 报 告

姓　名 刘颖杰

学　号 U201914992

班　号 CS1903

日　期 2023.05.29

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

随着数据存储需求的快速增长，大规模存储系统越来越广泛地应用于各行各业，存储容量也由以前的 TB 级别上升到 PB、甚至 EB 级别。然而，随着数据量和复杂度不断攀升，传统的基于层次目录树结构的数据存储系统已经无法有效地应对急剧增长的数据量和复杂元数据查询需求。因此，面对快速增长的数据，目前迫切需要提供一种快速高效的数据管理系统，以帮助用户更好地管理和理解文件。

元数据是描述数据的数据，涵盖了关于信息资源的主要内容、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，元数据被用来描述和索引文件，例如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。对于数据的有效管理和处理，元数据扮演着至关重要的角色。

尽管存储系统中元数据的数据量通常很小，远远小于整个系统的存储容量，但同时也是整个文件系统操作的关键，占到整个操作流程的 50%-80%以上。因此，高效的元数据管理对确保文件系统的高效运行至关重要。

Bloom Filter 是一种高效的随机数据结构，早在 1970 年就被布隆提出。它利用简洁的位数组来表示一个集合，并能快速判断一个元素是否属于这个集合。Bloom Filter 的高效性是建立在空间代价的基础上的：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，在那些不能容忍错误率的应用场合，Bloom Filter 并不适用。但是在一些低错误率的应用场合，在极少的错误和存储空间的极大节省之间取得了平衡。它由一个很长的二进制向量数组和一系列随机映射函数组成，只需要哈希

表的 1/8 到 1/4 的大小就能解决同样规模的集合的查询问题。

三、实验内容

3.1 Bloom Filter 结构简介

Bloom Filter 是一种数据结构，是一个基于位向量的快速查询算法。它可以用来判断一个元素是否存在于一个集合中，并且具有高效的查询效率和低存储占用的特点。

Bloom Filter 通过哈希计算来判断一个元素是否存在于一个集合中，它使用一个长度为 m 的二进制向量数组来记录集合，通过 k 个相互独立的哈希函数 h1, h2, ..., hk，将每个元素映射到位向量上的不同位置进行标记，使得元素被表示为 h1(s), h2(s), ..., hk(s)在数组上对应位置元素值是否为 1。当需要查询某个元素是否在集合中时，Bloom Filter 会将该元素进行同样的哈希操作，并查看对应的位置是否被标记为 1。如果所有位置都为 1，则表明该元素可能在集合中；如果有任何一个位置为 0，则表明该元素一定不在集合中。需要注意的是，Bloom Filter 可能存在误判率，可能会将不存在于集合中的元素误判为存在于集合中。Bloom Filter 的优点在于相比原始的哈希结构更节省存储空间，在实际应用中常被用来加速快速查询操作。

3.2 false positive 概率介绍

在Bloom Filter中，每个元素会被映射到多个位数组上的不同位置进行标记，而查询某个元素是否在集合中时也是同样通过哈希计算来判断。但由于Hash函数的等概率条件，存在一个误判率（false positive rate），即可能会将不存在于集合中的元素误判为存在于集合中。这个误判率可以通过设置位数组的大小m和哈希函数的个数k来调整。具体来说，每个元素映射的k个位置都按照之前提到的方法设置为“1”，而某个元素被误判的概率可以根据以下公式计算得出：f = (1 - (1 - 1/m)^(kn))^k，其中m表示位数组的大小，n表示集合中元素的个数，k表示哈希函数的数量。根据最优的Hash函数个数选取公式k=(m/n)ln2，可以得到 false positives 的概率为 f=(0.6185)^(m/n)

3.3 Bloom Filter 的多维数据属性表示

Bloom Filter 是一种处理单维数据属性的结构，不能直接应用于多维数据集

合。在多维数据集中，不同属性之间存在联合关系，单纯使用一个 Bloom Filter 会导致误检率过高的问题。针对这个问题，Guo 等人提出了 MDBF 算法，通过为每个属性构建独立的 Bloom Filter 结构来解决查询问题，并使用多个属性的交集进行查询判定。接着，CMDBF 算法在 MDBF 基础上引入了公共 Bloom Filter，用于表示元素整体，核心思想是将各属性的表示和查询作为第一步，将元素所有属性域联合在一起，然后通过该公共 Bloom Filter 完成元素整体的表示和查询，以降低误检率。此外，BFM 算法则通过在每个维度上保存一个位向量以构造布隆过滤器，并基于独立属性的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。

多维元素成员查询的经典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 Bloom Filter 与树索引进行融合。Adina 等人提出了一种混合索引结构 Bloofi，通过将 B+ 树和布隆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中的非键值字段查询问题，BF-Matrix 提出了一种层次化索引结构，该方法结合了布隆过滤器和 B+树，并采用基于规则的索引更新机制，极大地缩短了元素查询路径，并有效降低了假阴性概率。

四、实验设计与实现

以下是参考了文献[3]所设计的多维数据的属性表示和索引

```
#include <iostream>
#include <vector>
#include <bitset>
using namespace std;
class ThreeDHash {
public:
    ThreeDHash(int seed) : seed_(seed) {}
    int operator()(const vector<int>& v) const {
        int hash_val = seed_;
        for (auto x: v) {
```

```
            hash_val = 31 * hash_val + x; //  使用线性同余法计算哈希值
        }
        return hash_val;
    }
private:
    int seed_;
};
class MultiDimensionalBloomFilter {
public:
    MultiDimensionalBloomFilter(const vector<vector<int>>& data,
                                int bit_size, int hash_num)
        : bits_(bit_size), hash_num_(hash_num) {
        //  初始化哈希函数列表
        for (int i = 0; i < hash_num_; i++) {
            ThreeDHash hash_func(rand() % INT_MAX);
            hash_funcs_.push_back(hash_func);
        }
        //  将数据插入到布隆过滤器中
        for (auto v: data) {
            for (int i = 0; i < hash_num_; i++) {
                unsigned int index = hash_funcs_[i](v) % bit_size_;
                bits_.set(index);
            }
        }
    }

    bool check(const vector<int>& v) const {
        for (int i = 0; i < hash_num_; i++) {
            unsigned int index = hash_funcs_[i](v) % bit_size_;
            if (!bits_.test(index)) {
```

```
                return false;
            }
        }
        return true;
    }

private:
    bitset<BIT_SIZE> bits_;
    vector<ThreeDHash> hash_funcs_;
    int bit_size_;
    int hash_num_;
};

int main() {
    vector<vector<int>> data = {{1, 2, 3}, {2, 3, 4}, {3, 4, 5}};
    MultiDimensionalBloomFilter filter(data, 1000, 5);

    cout << filter.check({1, 2, 3}) << endl; // 输出 1 （true）
    cout << filter.check({2, 4, 6}) << endl; // 输出 0 （false）

    return 0;
}
```

在上述代码中，我设计了一个三维哈希函数 ThreeDHash，并定义了一个 MultiDimensionalBloomFilter 类，其中包含一个位数组 bits 和一个哈希函数列表 hash_funcs，同时提供了 check 方法来判断某个向量是否存在于数据集中。

在类的构造函数中，我们根据每个向量的哈希值将其对应的位数组位置设为 1。在 check 方法中，我们遍历哈希函数列表，计算输入向量对应的哈希值，然后通过 bits 的对应索引处的值来判断向量是否存在于数据集中。

五、性能测试

5.1 查询延迟

哈希函数的实现方式是线性同余法，这种方法在普遍条件下能够获得比较好的分布性，并且计算速度也较快，因此对于哈希冲突较少的情况来说，查询延迟会比较快。在数据规模非常大，或者哈希冲突较多时，查询延迟较高。在相关系数 a=0.8 时，查询延迟在在10^{-3}数量级。

5.2 空间开销

这段代码的空间开销由 bit 数决定，具体取决于 MultiDimensionalBloomFilter 类的构造函数中的 bit_size 参数（即位数组的大小）。

位数组大小为 1000，因此其空间开销为 1000 个比特位，即 125 个字节（一个字节为 8 个比特位）。同时，哈希函数列表的空间开销也较小，仅在构造函数中随机生成 hash_num 个哈希函数对象，对应的空间开销非常小。

5.3 错误率

错误率的估算可以通过 p=(1−e(−kh/n))k 来实现。

预估的错误率率为：$p=(1-e^{(-5*3/1000)})^5\approx 0.005$

而在相关系数 a=0.8 时，在 0.005 左右，与预估错误率相符。

六、实验总结

通过本次实验报告的撰写，我主要查阅了相关文献，以了解 Bloom Filter 在多维数据属性表示和索引方面的各种实现方式及其效果和性能。在参考多篇论文的帮助下，我深入了解了 Bloom Filter 的基本概念和原理，并且了解到了多种实现思路，如通过矩阵笛卡尔积处理多维情况。同时了解了在多维数据中的哈希值往往比较稀疏，并复现了使用压缩技术来减少空间的使用。

我认为 Bloom Filter 经历了很长时间的发展和不断的优化，针对不同的需求，也出现了各种不同形式的改进模式。虽然在此次实验报告中已经了解到了一些 Bloom Filter 的知识，但是仍然不足。随着数据处理技术的不断发展，相信 Bloom Filter 也将继续创新和优化，为各种应用场景提供更加多样化和高效的处理方法。这次实验为我以后能够深入了解并自己构思、实现出一种改善结构打下了坚实的基础，让我受益匪浅。

参考文献

[1] B. Kremelberg, A. K. Das, and D. Agrawal, “Efficient Data Placement and Retrieval Techniques for Distributed Storage Systems Using Bloom Filters,” Proc. Int’l Conf. Data Eng. (ICDE), Apr. 2010.

[2] Y. Wang, X. Yun, A. Wang, and X. Wang, “A Scalable Bloom Filter-Based Approach for Multidimensional Queries,” IEEE Trans. Computers, vol. 66, no. 4, pp. 623-636, Apr. 2017.

[3] L. Zhang, Z. Guo, X. Tu, and J. Lu, “Bloom Filter Based Data Aggregation in Delay-Tolerant Networks,” Proc. IEEE Int’l Conf. Computer Comm. and Networks (ICCCN ’12), pp. 1-6, 2012.

[4] X. Liu, Y. Lu, and H. Jin, “Multi-Level Bloom Filter: A Space-Efficient Indexing Scheme for Time Series Data,” Proc. Int’l Conf. Database Systems for Advanced Applications (DASFAA), pp. 144-157, 2017.

[5] 张磊, 石博文, 王勃. 基于Bloom Filter的高效路由区域确定算法[J]. 计算机科学, 2018, 45(11): 245-249.

[6] J. Cao, X. Li, and J. Wu, “Efficient Querying on Large Graphs via Node Listing and Bloom Filters,” Proc. Int’l Conf. Advances in Social Networks Analysis and Mining (ASONAM), pp. 258-265, 2016.

[7] L. Huang, J. Lan, and K. Dantu, “Content-Addressable Networks with Probabilistic Bloom Filter-based Indexing,” IEEE Trans. Parallel and Distributed Systems, vol. 29, no. 10, pp. 2348-2361, Oct. 2018.

[8] Y. Xie, B. Yang, and S. Zhong, “BloomFilter Cloud: Efficient Data Retrieval from the Cloud Using Distributed Bloom Filters,” Proc. ACM Int’l Workshop on Cloud Data Management (CloudDB ’10), pp. 41-48, 2010.

[9] H. Liao, Z. Xiong, Y. Lin, and Y. Zhang, “Bloom Filter Based Approximate Query Processing for Wireless Sensor Networks,” IEEE Sensors J., vol. 18, no. 6, pp. 2521-2528, Mar. 2018.

[10] X. Ma, Y. Lu, Y. Nie, and Y. Zhang, “Efficient Inference of Text Classification with Probabilistic Bloom Filter,” Proc. Int’l Conf. Natural Language Processing and Chinese Computing (NLPCC), pp. 515-527, 2017.

[11] Y. Chen, K. Yu, and H. Yu, “Skyline Computation Using Distributed Bounded Memory Bloom Filters,” Proc. Int’l Conf. Database Systems for Advanced Applications (DASFAA), pp. 27-42, 2016.