2021 级

《大数据系统存储与管理》课程

基于 BloomFilter 的多维数据属性表示和索引

姓　名　陈单睿

学　号　U202115659

班　号　计科 2110 班

日　期　2024.04.20

目　　录

一、Bloom Filter 技术简介

1. BloomFilter 的简介

Bloom Filter 是一种空间效率高、查询速度快的数据结构，最早由布隆（Burton Howard Bloom）在 1970 年提出。Bloom Filter 的主要思想是利用位数组和多个哈希函数来判断一个元素是否存在于集合中。Bloom Filter 的优势在于它可以高效地判断一个元素是否存在于集合中，而且可以使用很少的空间来存储数据。因此，Bloom Filter 被广泛应用于网络路由、缓存系统、搜索引擎等领域。

2. BloomFilter 的原理

Bloom Filter 的核心是一个位数组和多个哈希函数。位数组的每个元素只能取 0 或 1 两个值。多个哈希函数可以将一个元素映射到位数组的多个位置上。当一个元素被加入到 Bloom Filter 中时，它会被哈希函数映射到多个位置上，并将这些位置的值都设为 1。当需要查询一个元素是否存在于 Bloom Filter 中时，将该元素通过哈希函数映射到位数组的位置上，并判断这些位置的值是否都为 1。如果这些位置的值都为 1，则可以认为该元素可能存在于 Bloom Filter 中；如果这些位置的值有一个为 0，则可以确定该元素一定不存在于 Bloom Filter 中。

假设一个 Bloom Filter 包含一个位数组和三个哈希函数。当元素 A 被加入到 Bloom Filter 中时，它会被哈希函数映射到位数组的三个位置上，并将这些位置的值都设为 1。当需要查询元素 B 是否存在于 Bloom Filter 中时，将元素 B 通过哈希函数映射到位数组的三个位置上，并判断这些位置的值是否都为 1。由于元素 B 只被哈希函数映射到了两个位置上，其中一个位置的值为 0，因此可以确定元素 B 不存在于 Bloom Filter 中。

Bloom Filter 的哈希函数需要满足以下两个条件：

（1）哈希函数的结果必须是一个固定长度的值；

（2）哈希函数的结果必须均匀地分布在位数组的各个位置上。

常用的哈希函数有哈希链、MD5、SHA 等。

3. BloomFilter 的错误率分析

Bloom Filter 的错误率与位数组的大小和哈希函数的数量有关。假设位数组的大小为 m，哈希函数的数量为 k，元素的个数为 n，错误率为 p，则有以下公式：

$$p = (1 - e^{-k\frac{n}{m}})^k$$

其中，e 是自然对数的底数。当一个元素被查询时，它可能被误判为存在于 Bloom Filter 中的概率为 p。当位数组的大小和哈希函数的数量固定时，错误率 p 随元素的个数 n 增加而增加。

Bloom Filter 的错误率可以通过调整位数组的大小和哈希函数的数量来控制。如果需要降低错误率，可以增加位数组的大小和哈希函数的数量；如果需要提高空间利用率，可以减小位数组的大小和哈希函数的数量。

二、多维数据的 BloomFilter 的基本原理

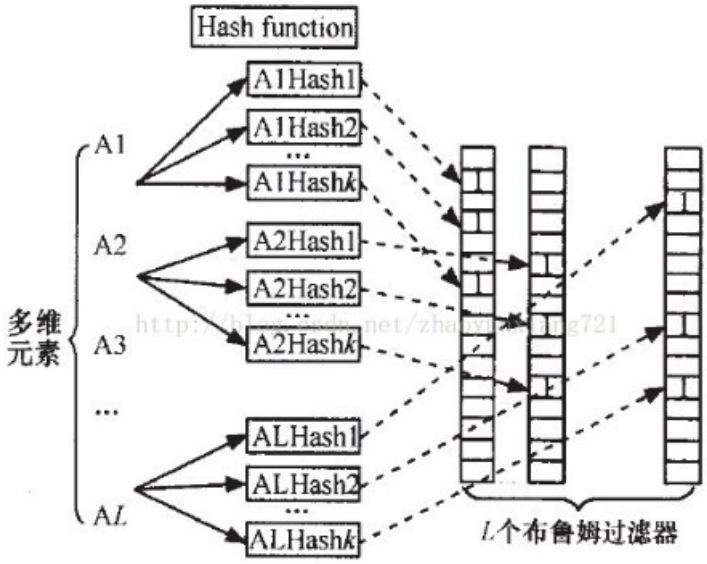
多维数据的 Bloom Filter（Multi-Dimensional Bloom Filter，MDBF）是 Bloom Filter 的扩展，用于处理多维数据的查询。MDBF 的基本原理是将多维数据映射到多个一维 Bloom Filter 中，每个一维 Bloom Filter 对应数据的一个维度。这样可以在查询时同时对多个维度进行判断，提高了查询的准确性和效率。

在 MDBF 中，每个维度都有一个对应的一维 Bloom Filter。当一个多维数据被加入到 MDBF 时，通过哈希函数将其映射到每个维度对应的一维 Bloom Filter 中，并将相应位置的值设为 1。因此，一个多维数据在 MDBF 中会在多个一维 Bloom Filter 中留下标记。

在查询时，将需要查询的多维数据通过哈希函数映射到每个维度对应的一维 Bloom Filter 中，并判断每个维度的一维 Bloom Filter 中对应位置的值是否都为 1。只有当所有维度的一维 Bloom Filter 中对应位置的值都为 1 时，才能确定该多维数据存在于 MDBF 中。

MDBF 的查询效率取决于维度的数量和每个一维 Bloom Filter 的错误率。增加维度的数量会增加查询的准确性，但也会增加错误率。因此，在设计 MDBF 时需要根据实际需求平衡维度数量和错误率。

下图展示了一个简单的多维数据的 Bloom Filter 示意图，其中有 L 个维度对应 L 个一维 Bloom Filter，多维数据被映射到每个维度的一维 Bloom Filter 中。



MDBF 是一种有效的数据结构，可用于处理多维数据的查询，提高查询效率和准确性。通过合理设计哈希函数和位数组大小，可以优化 MDBF 的性能，适应不同的应用场景。

三、实验设计与实现

1.总体结构设计

在本次实验中，我们假设每个元组有两个维度，使用两个 n 位 bitset 来记录信息。我们采用了 3 重 hash 函数，在本次实验中那个，我们的 hash 函数是根据 BKDRhash 函数进行改写的，用于生成哈希值。哈希函数采用初始值和字符串 s 进行计算，将字符串映射为哈希值。hash 函数的定义如下所示：

```
struct Hash {
    Hash(size_t initial) : initialValue(initial) {}
    size_t operator()(const string& s) const{
        size_t value = initialValue;
        for (auto e : s){
            value += e;
            value *= 131;
        }
        return value;
```

```
    }
    size_t initialValue;
};
```

本实验所采用的 3 维 Bloom Filter 的数据结构包括构造函数、添加元素方法和检查元素是否存在方法。布隆过滤器使用三个哈希函数进行计算，维护索引位集合和属性位集合。

```
class BloomFilter {
private:
    array<Hash, 3> hashes; // 哈希函数
    bitset<N> indexBitset; // 索引位
    bitset<N> attributeBitset; // 属性位
        double indexCoefficient; // 索引系数
}
```

2. Set 方法

```
        size_t i1 = Hash1()(key1) % N;
        size_t i2 = Hash2()(key1) % N;
        size_t i3 = Hash3()(key1) % N;
        size_t i4 = (Hash1()(key2) * i2) % N;
        size_t i5 = (Hash2()(key2) * i3) % N;
        size_t i6 = (Hash3()(key2) * i1) % N;
        bitset1.set(i1);
        bitset1.set(i2);
        bitset1.set(i3);
        bitset2.set(i4);
        bitset2.set(i5);
        bitset2.set(i6);
```

四、实验测试

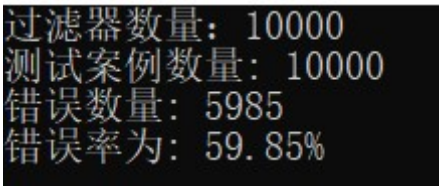
本次实验中，我选用的是来自 kaggle 的 covid_abstracts 数据集。在这个数据

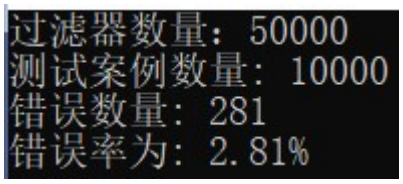
集中，每一项都包含了一篇文献的名称及 url 地址，相互不重复，总共 10000 项。

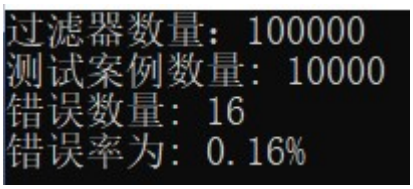
而针对测试程序，我选用的三个参数是 2021、5659、8968，索引系数设置为了要求的 0.8。

我选取的测试方法是一边插入一边检查，通过检查是否已经存在本数据，来检测错误率。

下面，我们通过调控 Bloom Filter 的数量来观察错误率的变化。







可以发现，随着 BloomFilter 数量的增加，错误率有着显著的降低。

五、实验总结

在本次实验中，我完成了多维 BloomFilter 的 C++实现。通过本次实验，我掌握了 BloomFilter 的原理，尽管实现的过程中可能存在一些冗余或繁琐的地方，但通过不断尝试和优化，我逐渐掌握了更简单、更高效的实现方式。

在构建二维数据展示时，我感受到了增加复杂性所带来的挑战，但同时也明白了该了如何处理更复杂的数据结构。这次实验不仅提高了我的编程技能，还让我更好地理解了数据结构和算法在实际应用中的价值。我也在实验的过程中掌握了自主收集数据集的途径。

总的来说，通过这次实验，我意识到了实践的重要性，只有通过动手实践并不断优化，才能真正掌握知识并取得进步。即使在实现过程中遇到困难或繁琐，也要保持耐心和积极的态度，因为每一次尝试都是对知识的巩固和提升。这种探

索和实践的过程让我收获良多，也激发了我对大数据存储系统更深入探索的兴趣。

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.