



# 华中科技大学

## 大数据存储系统与管理课程报告

姓 名：刘正舆  
学 院：计算机科学与技术学院  
专 业：计算机本硕博  
班 级：2101 班  
学 号：U202115667  
指导教师：华宇

分数	
教师签名	

2024 年 4 月 21 日

# 一、实验背景

布隆过滤器（Bloom Filter）是由布隆在 1970 年提出的一种数据结构，用于快速检查一个元素是否属于一个集合中。它以牺牲一定的精确性为代价，来提高查询的速度和减少内存消耗。Bloom Filter 是一种空间效率非常高的数据结构，在许多实际应用中都有着广泛的应用，例如网络路由器、分布式系统、缓存系统等。

在介绍布隆过滤器之前，首先要理解一个常见的问题：如何判断一个元素是否属于一个集合中？一种最直接的方法是使用哈希表，但是哈希表会占用大量的内存，特别是当集合非常大时，内存消耗会成为一个问题。而且，对于大规模的数据集，哈希表需要进行频繁的内存分配和哈希计算，导致性能下降。

为了解决这个问题，布隆过滤器应运而生。

布隆过滤器基于一个简单的思想：通过多个哈希函数将元素映射到一个位数组中，并将对应位置置为 1。当查询一个元素时，布隆过滤器会使用相同的哈希函数计算出位数组中的位置，如果所有位置都为 1，则可以判断元素存在于集合中；如果任何一个位置为 0，则元素肯定不在集合中。

优点：高效的空间利用率：布隆过滤器只需要占用少量的内存空间，不随存储元素数量的增加而线性增长，这使得它在大规模数据场景下具有明显的优势；快速查询速度：查询元素的时间复杂度是固定的，与存储元素数量无关，通常是常量时间；灵活性：布隆过滤器可以根据需要调整大小和哈希函数的数量，以平衡精确性和性能。

缺点：误报率：布隆过滤器可能会误报元素存在，但绝不会误报元素不存在。这是因为如果一个元素被哈希到的所有位置都已经被置为 1，那么任何其他元素查询到这些位置时都会被误报为存在。但是，通过调整哈希函数的数量和位数组的大小，可以降低误报率；无法删除元素：由于布隆过滤器不存储实际的元素数据，所以无法直接删除元素。一旦元素被添加到布隆过滤器中，就无法再将其删除。

在本文中，我通过对单一布隆过滤器按维度数划分实现了基于 Bloom Filter

的多维数据属性表示，并测试了误判率及时间开销。

## 二、布隆过滤器设计

通过二维数组，我得以将一维布隆过滤器长度等分，实现多维度数据表示（如：

图 1 布隆过滤器数据结构）

图 1 布隆过滤器数据结构

```
// 定义多维布隆过滤器结构
typedef struct {
    bool filter[FILTER_SIZE][NUM_DIMENSIONS];
} MDBF;
```

设计相关函数：布隆过滤器初始化函数、添加元素函数、判断函数以及测试所用的检查元素是否真正添加函数（如：）

图 2 相关函数定义

```
// 初始化多维布隆过滤器
void initMDBF(MDBF *mdbf) {
    for (int i = 0; i < FILTER_SIZE; i++) {
        for (int j = 0; j < NUM_DIMENSIONS; j++) {
            mdbf->filter[i][j] = false;
        }
    }
}

// 添加元素到多维布隆过滤器
void addToMDBF(MDBF *mdbf, int *element) {
    for (int i = 0; i < NUM_DIMENSIONS; i++) {
        int index = element[i] % FILTER_SIZE;
        mdbf->filter[index][i] = true;
    }
}

// 检查元素是否存在于多维布隆过滤器中
bool isInMDBF(MDBF *mdbf, int *element) {
    for (int i = 0; i < NUM_DIMENSIONS; i++) {
        int index = element[i] % FILTER_SIZE;
        if (!mdbf->filter[index][i]) {
            return false;
        }
    }
    return true;
}

// 检查元素是否为所有数据（确定）
bool isInMDBF_ofcourse(int element_repository[NUM_ELEMENTS][NUM_DIMENSIONS], int *element) {
    for (int i = 0; i < NUM_ELEMENTS; i++) {
        for (int j = 0; j < NUM_DIMENSIONS; j++) {
            if (element[j] == element_repository[i][j])
                return true;
        }
    }
    return false;
}
```

## 三、理论分析

布隆过滤器不存在 false negative 问题（向量各位只会从 0 到 1，因此其他数据的插入不会影响原有 1 的变化，故对于不存在的判断一定是正确的）。

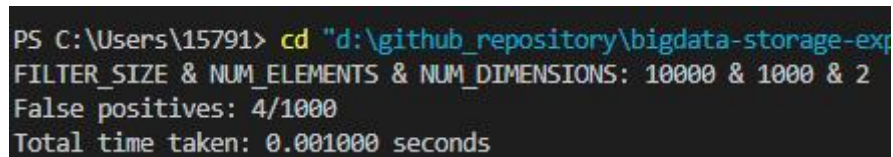
而对于 false positive 问题，根源来自于 hash 冲突，因此可通过增加位数组大小、采用更均匀的 hash 函数、降低存储量等方式缓解。

## 四、性能测试

本实验通过将原本随机生成的数据保存并与新随机生成的测试数据进行比较，来确认是否发生 false positive 问题，进而统计 false positive 发生率；同时计时得出总用时。

最终性能结果如：图 3 性能测试结果

图 3 性能测试结果



```
PS C:\Users\15791> cd "d:\github_repository\bigdata-storage-exp
FILTER_SIZE & NUM_ELEMENTS & NUM_DIMENSIONS: 10000 & 1000 & 2
False positives: 4/1000
Total time taken: 0.001000 seconds
```

也即对于 1000 次测试，仅产生了 4 次 false positive 错误，总用时 0.001s。