

《大数据存储系统与管理》课程报告

**基于Bloom Filter的设计**

**姓 名 武桐羽**

**学 号 U202015658**

**班 号 计算机2110班**

**日 期 2024年4月18日**

**目 录**

[一、实验目的 1](#_Toc164726023)

[二、实验背景 1](#_Toc164726024)

[三、实验内容 1](#_Toc164726025)

[3.1 Bloom Filter原理 1](#_Toc164726026)

[3.2 操作流程分析 2](#_Toc164726027)

[3.3 理论分析 3](#_Toc164726028)

[四、实验设计 4](#_Toc164726029)

[4.1多维数据属性表示和索引 4](#_Toc164726030)

[4.2设计数据结构 4](#_Toc164726031)

[4.3实验设置 5](#_Toc164726032)

[五、实验测试 6](#_Toc164726033)

[六、实验总结 6](#_Toc164726034)

[参考文献 8](#_Toc164726035)

# 一、实验目的

通过设计和实现基于Bloom Filter的系统，深入理解Bloom Filter 的原理和特点，掌握其在数据存储和查询中的优势和限制，并能够进行相应的性能评估和理论分析。

实验基本结构如下：

1. 对Bloom Filter的数据结构进行设计；

2. 对操作流程进行分析：如何保证和实现所提出的设计目标；

3. 进行理论分析，例如false positive和false negative；

4. 多维数据属性表示和索引（系数0.8）；

5. 测试性能：查询延迟，空间开销，错误率等性能指标。

# 二、实验背景

在传统的数据存储中，哈希函数是一种有效的存储方式，它能够将元素映射到存储空间中的具体位置，并插入相应数据。然而，这种方法存在两个主要的不足：当数据量变得庞大时，需要很大的存储空间；在搜索过程中，虽然能够实现精确匹配，但可能会非常耗时。

而Bloom Filter可以解决此类问题。Bloom Filter是一种空间效率比较高的数据表示和查询结构，其使用了多个哈希函数来降低冲突，用于判断一个元素是否存在于一个集合中。这种数据结构适合应用在能容忍低错误率的场合。

该算法由Burton H. Bloom 于1970 年提出，它突破了传统哈希函数的映射和存储元素的方式，通过一定的错误率换取了空间的节省和查询的高效。由于它具有空间效率高、查询速度快等优点，其在大数据中应用广泛，如网络路由器、分布式系统、缓存系统等。

# 三、实验内容

## 3.1 Bloom Filter原理

Bloom filter是一种空间效率较高的数据结构，用于表示一个集合并支持成员查询操作。其核心思想是利用多个哈希函数将集合中的元素映射到位数组中，从而实现对集合的编码。

在标准的 Bloom filter 中，首先定义了一个长度为 m 的位向量，以及 k 个相互独立的哈希函数。这些位向量的初始值为0，以及哈希函数的值域位于1~m之间。对于形如｛x1，x2，…，xn｝的集合 S 中的每个元素 x，使用k 个哈希函数将其映射到位向量的 k 个不同位置上，并将这些位置上的值设置为 1。如果某个位置已经为 1，则保持不变。

当需要查询某个元素 zi 是否属于集合 S 时，同样使用这 k 个哈希函数将其映射到位向量中。如果所有映射位置上的值都为 1，则认为 zi可能属于集合 S，否则可以确定 zi 不属于集合 S。值得注意的是，由于Bloom filter 存在假阳性错误率，因此只能为可能属于。Bloom filter的工作原理图如图3.1所示。

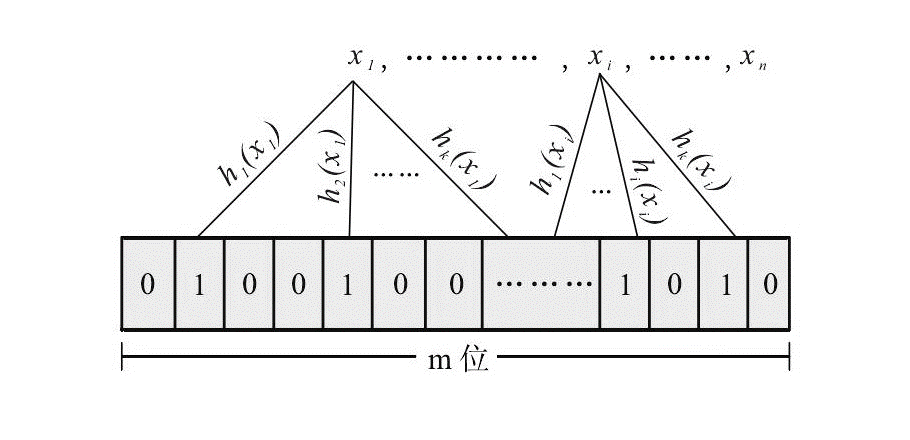


图3.1 Bloom filter工作原理图

Bloom filter 与传统的哈希函数相比，优势在于其高效的空间利用率和快速的查询速度。由于它不需要处理哈希冲突，因此无论集合中有多少元素，或者已经有多少元素被添加到位数组中，添加和查询操作的时间复杂度都仅取决于哈希函数的计算时间。此外，由于 Bloom filter 对集合元素进行了编码，它还提供了一种保护隐私的方式，使得直接查看集合元素变得困难。

然而，Bloom filter 的一个主要缺点是存在假阳性错误率即false positive，该部分将在3.3部分中阐述。

综上所述，Bloom filter 是一种高效且实用的数据结构，特别适用于需要快速判断元素是否属于某个集合的场景。在使用时需要注意其假阳性错误率的存在，并根据具体需求调整参数以优化性能。

## 3.2 操作流程分析

实现Bloom filter主要为两步：

1. 数据装入。

设置长度为m的向量V，k个相互独立均匀分布的哈希函数集合H，n个元素组成的集合S。

用哈希函数分别将集合S中的n个元素映射到向量V中的相应位置，将V中相应位置为1。

1. 数据判断。

当新元素y到来时,对y进行k次哈希运算，检查所有的h1( y)、h2(y) 、…、hk(y) 对应向量V中的位是否全部为1，是的话则说明元素y属于集合S，否则说明y不属于集合S。

## 3.3 理论分析

在使用Bloom filter判断一个元素是否属于集合时，可能会出现假阳性（false positive）的情况，即出现误判，把不属于该集合的元素误判为属于该集合。这是由于地址冲突不能避免，因此Bloom filter算法可能对位向量中同一个位多次置1，从而导致误判。当然，该算法不可能发生false negative，即属于该集合的元素误判为不属于该集合。

这种错误率可以通过概率方法进行计算，并且可以通过调整参数来优化。

具体来说，当使用 k 个哈希函数将 n 个元素映射到长为m的向量V中时，对于一个键值在m个空间的向量来说，被映射为1的概率是1/m，被映射为0的概率是1 - 1/m，即k \* n个键值都被映射为0的概率为 (1 - 1/m)kn。

当集合中所有元素都映射完毕后，V中任意一位为0的概率p为(1 - 1/m)kn，即：

当m趋于无穷大时，的极限为 ，即：

即若出现误判的情况，需满足y在V向量的k个映射位上的值都为1。即错误率fp为(1 - p)k，即：

令g =，根据对称性可知当p=1/2时，g取到最小值，此时fp也为最小值。此时有：

在Bloom filter中，参数m和n的比值是已知的，因此为了让错误率最小，需要求。在这种情况下，错误率fp可以近似为。

当m与n的比值越大时，则要求哈希函数的个数越多，此时错误率更小。当哈希函数取最优个数时，错误率与集合中元素个数n以及向量大小m相关。

# 四、实验设计

## 4.1多维数据属性表示和索引

在Bloom filter中，多维数据属性表示指的是将多个属性组合在一起，作为一个整体进行哈希和插入。这样做可以更全面地表示一个数据项，以便后续进行查询和检索。对于具有多维数据属性的元素，需要对所有属性一起加以判断。Bloom filter常常用于检查某个项是否存在于数据集中，而在这种情况下，一对相关的字符串可能代表一个项目的不同属性或标识。

举例来说，如果元素为网站用户的登录信息，那么两个字符串可能表示用户名和密码。在这种情况下，需要同时将用户名和密码添加到Bloom filter中，并在后续的查询中同时检查它们。

在实际实验中，每一种属性都应当对应一个位数组，如果数据共为n维，则需要n个对应的位数组，并且用n组hash函数对每一维进行处理。只有当每一个属性对应的映射值都为1时，才能说明该元素可能存在。

而在Bloom filter中，还需考虑每个属性的位被设置为1的概率，即索引系数。在本实验中，索引系数设置为0.8。

## 4.2设计数据结构

哈希函数的结构体为：

struct Hash {

    Hash(size\_t initial) : initialValue(initial) {}

    size\_t operator()(const string& s) const {

        size\_t hash = initialValue;

        for (char c : s) {

            hash = (hash \* 131) + c;

        }

        return hash;

    }

    size\_t initialValue;

};

其中，initialValue为哈希函数的初始值，用于设置相互独立的不同哈希函数。

Bloom filter的数据结构为：

class BloomFilter {

public:

    BloomFilter(size\_t initial1, size\_t initial2, size\_t initial3, double indexCoefficient)

        : hashes({Hash(initial1), Hash(initial2), Hash(initial3)}), indexCoefficient(indexCoefficient) {

        initializeIndexBits();

    }

    void add(const string& key1, const string& key2) { …

bool contains(const string& key1, const string& key2) const {…

private:

    array<Hash, 3> hashes; // 三个哈希函数

    bitset<N> indexBitset; // 索引位集合

    bitset<N> attributeBitset; // 属性位集合

    double indexCoefficient; // 索引系数

    // 根据索引系数设置属性位

    void setAttributeBits(size\_t index) {

        size\_t numAttributeBits = static\_cast<size\_t>(ceil(indexCoefficient \* log(N)));

        for (size\_t i = 0; i < numAttributeBits; ++i) {

            attributeBitset[(index + i) % N] = true;

        }

    }

    // 初始化索引位集合

    void initializeIndexBits() {

        indexBitset.reset();

    }

};

其中，add方法用来添加元素，设置了数据索引位和属性位的映射值，contains方法则用来检查索引位和属性位是否为1，从而判断该数据是否已经存在。由于该部分篇幅过长且较为重复，在此处不全部展开。

## 4.3实验设置

本实验的数据来源网址为：

<https://archive.ics.uci.edu/dataset/967/phiusiil+phishing+url+dataset>

选取了其中前17428项数据，每一项包含一个文件名和一个网址，每一项数据均不相同。文件名作为数据的索引位，网址作为数据的属性位。

实验中设置了三个哈希函数，初始值分别为123、456、789。索引系数设置为0.8。

在本实验中，插入数据和查找数据同时进行，从而方便测试错误率。每次插入一组数据，都将检查是否已存在本数据，如果存在则说明发生了误测。

# 五、实验测试

当Bloom filter大小为50000时，得到测试结果如图5.1所示。

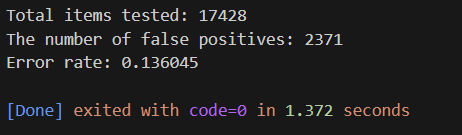


图5.1 大小为50000时的测试结果

当Bloom filter大小为75000时，得到测试结果如图5.2所示。

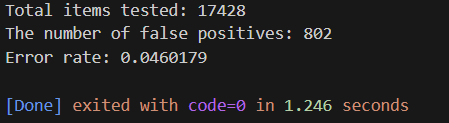


图5.2 大小为75000时的测试结果

当Bloom filter大小为100000时，得到测试结果如图5.3所示。

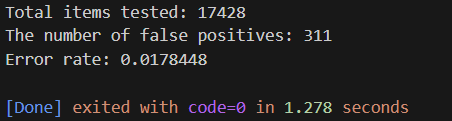


图5.3 大小为100000时的测试结果

由测试结果可以看出，当哈希函数的个数和测试数据个数一定时，Bloom filter的大小越大，误测的数目越少，错误率越低。

# 六、实验总结

在本次课程设计实验中，我深入学习了Bloom filter的相关知识。Bloom filter是一种重要的数据结构，可以用于解决大规模数据集中的重复元素检测和数据查询等问题。它通过多个哈希函数和位数组来实现，具有高效的空间利用率和快速的查询速度。

实验中我使用c++代码简单实现了基础的Bloom filter算法，包括添加元素和检查元素是否存在。我尝试简单构建了二维数据的展示，相比单一属性的数据又增添了一些复杂性，但不多。我最终实现的Bloom filter并不是很复杂，自我认为代码也有些繁琐，可能还有更简单而高效的实现方式来优化，不过得到的成果符合理论，从中我收获许多。

Bloom filter的许多参数调节都需要根据具体的需求来进行，包括选择合适的哈希函数，设置适当的位数组大小，从而可以优化Bloom filter的性能，其错误率也与集合中元素个数以及数据大小相关，然而如果为了降低错误率而增加Bloom filter的大小，也需要考虑空间的开销。总之，合理地使用Bloom filter，可以有效减少数据存储和查询的时间和空间成本，提高大数据存储系统的性能和可扩展性。

# 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.