

**2021** 级

《大数据存储与管理》课程

**课 程 报 告**

基于Bloom Filter的设计

**姓 名 邹扬**

**学 号 U202115421**

**班 号 大数据2102班**

**日 期 2024.04.15**

**目 录**

[一、课设目的 2](#_Toc19423)

[二、课设背景 2](#_Toc32324)

[三、课设内容 2](#_Toc8934)

[3.1 Bloom Filter原理 2](#_Toc6782)

[3.2 false positive分析 4](#_Toc30835)

[3.3 Bloom Filter的多维数据属性表示 5](#_Toc28875)

[四、实验设计 5](#_Toc28620)

[五、 性能测试 6](#_Toc7839)

[5.1查询延迟 6](#_Toc8913)

[5.2空间开销 7](#_Toc26539)

[5.3错误率 7](#_Toc23160)

[六、课程总结 7](#_Toc23794)

[参考文献 8](#_Toc10262)

# 一、课设目的

1. 分析bloom filter数据结构的设计；

2. 操作流程的分析，即如何保证和实现所提出的设计目标；

3. 进行false positive的理论分析；

4. 多维数据属性表示和索引（系数0.8）

5. 实验性能的测试。

# 二、课设背景

Bloom Filter是一种空间效率很高的随机数据结构，它利用位数组很简洁地表示一个集合，并能判断一个元素是否属于这个集合。Bloom Filter的这种高效是有一定代价的：由于可能出现[哈希](https://so.csdn.net/so/search?q=%E5%93%88%E5%B8%8C&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/qq_55624813/article/details/_blank)碰撞，不同元素计算的哈希值有可能一样，导致一个不存在的元素有可能对应的比特位为1，这就是所谓“假阳性”（false positive）。相对地，“假阴性”（false negative）在BF中是绝不会出现的。因此，Bloom Filter不适合那些 “零错误”的应用场合。而在能容忍低错误率的应用场合下，Bloom Filter通过极少的错误换取了存储空间的极大节省。

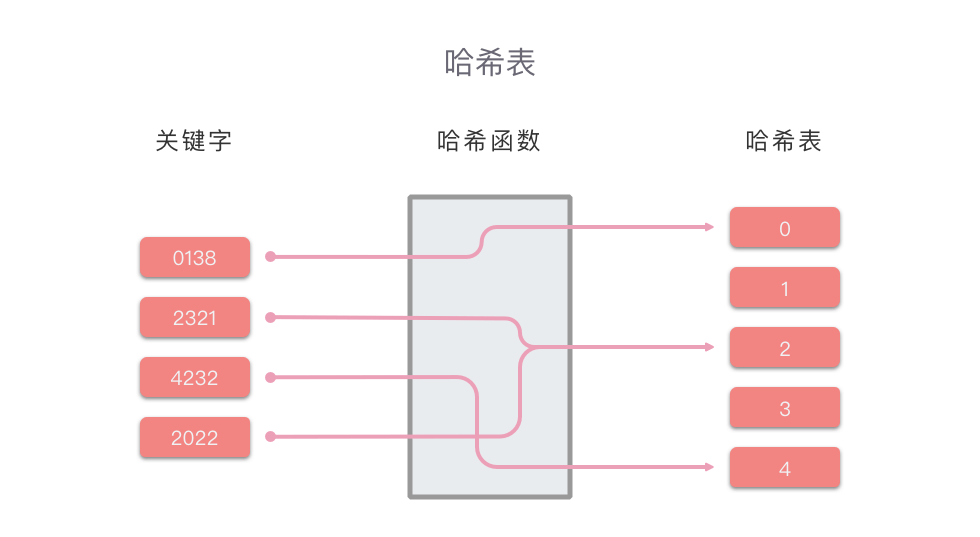
本次课程设计通过分析Bloom Filter的原理，最终提出对性能进行一定的优化的方法。

# 三、课设内容

## 3.1 Bloom Filter原理

如果想判断一个元素是不是在一个集合里，一般想到的是将所有元素保存起来，然后通过比较确定。链表、树等等数据结构都是这种思路。但是随着集合中元素的增加，我们需要的存储空间越来越大，检索速度也越来越慢(O(n),O(logn))。这时候我们可以利用哈希表这种数据结构，基于哈希函数的特性，它在理想情况下(不发生哈希冲突)，检索速度可以达到O(1)。

一张哈希表的示意图如下所示：



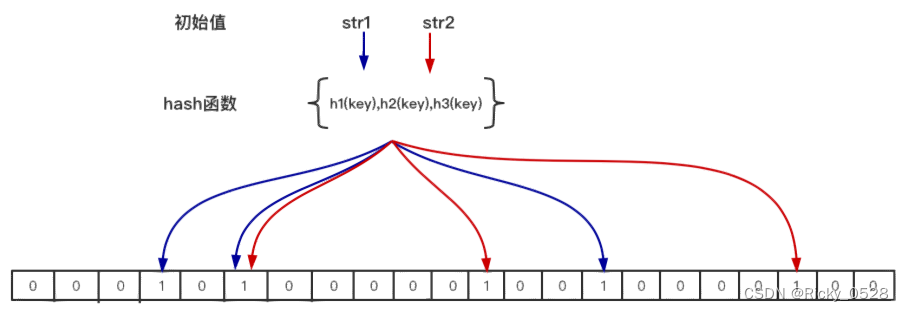
如果找到的哈希函数足够完美，那么理想状态下可以做到每个key对应一个唯一的hashcode，但实际上往往会出现哈希冲突，即两个不同的key对应同一个hashcode，即发生了碰撞。

布隆过滤器使用了上面的思路，即利用哈希表这个数据结构，通过一个Hash函数将一个元素映射成一个位阵列（Bit array）中的一个点（每个点只能表示0或者1），这样一来，我们只要看看这个点是不是1就知道在集合中有没有它了。

但是用 hash表存储大数据量时，空间效率还是很低，当只有一个 hash 函数时，还很容易发生哈希碰撞。在哈希冲突的情况下，我们无法使用一个哈希函数来判断一个元素是否存在于集合之中，解决方法就是使用多个哈希函数，如果其中有一个哈希函数判断该元素不在集合中（元素经过Hash之后映射在位阵列中的点为0），则不在。如果它们都判断存在，也有一定判断错误可能，不过这要比只用一个哈希函数来判断“一个元素存在于集合之中”的可靠性要高很多。这种多个Hash组成的数据结构就叫Bloom Filter。

Bloom Filter是基于一个m位的位阵列（b1,…bm），这些位阵列的初始值为0。另外，还有一系列的hash函数（h1,…hk），这些hash函数的值域属于1~m。

当有变量被加入集合时，通过 K 个映射函数将这个变量映射成位图中的 K 个点，把它们置为 1。其结构如下图所示。



查找时，如果K个hash值对应的位向量都为1，则判断可能在此数据集中。但是如果有任意hash值对应的位向量为0，因此判断必定不在此集合中。

而事实上，查询所得位向量都为1的查询变量也可能不在此集合中，即会出现误报。显然，插入数据越多，1的位数越多，错误率越大。

## 3.2 false positive分析

易知由于哈希函数存在碰撞，而哈希碰撞导致的巧合会将不同的元素存储在相同的比特位上，这样会导致无法判断究竟是哪个输入产生的1，因此会有一定的错误率。而这种错误率的大小是可以估计的。下面我们来估计这个数值。

为了简化计算，假设我们的哈希函数选择位数组中的比特时，都是等概率的。

在位数组长度m的bloom filter中插入一个元素，它的其中一个哈希函数会将某个特定的比特置为1。因此，在插入元素后，该比特仍然为0的概率是： 。

现有k个哈希函数，并插入n个元素，则该比特仍然为0的概率是：。

易得它已被置为1的概率是：。

即在插入n个元素后，如果查询变量不在集合中，那么被误认为在集合中的概率（也就是所有哈希函数对应的比特都为1的概率）为：。

当n比较大时，这个概率近似为：。

最终我们发现，最优的k值为，此时错误率为。

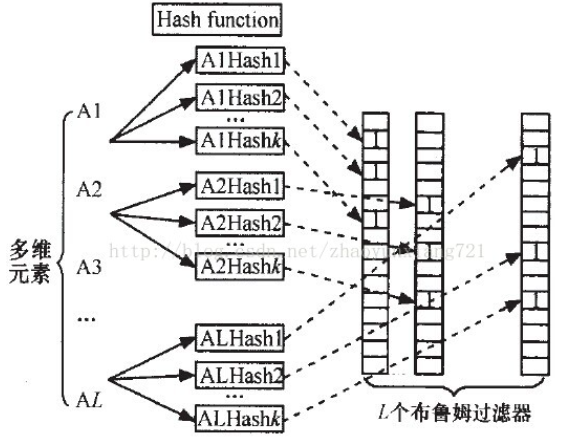
同时易得，在哈希函数的个数k一定的情况下：

（1）位数组长度m越大，错误率越低。

（2）已插入元素的个数n越大，错误率越高。

## 3.3 Bloom Filter的多维数据属性表示

对于多重属性的元素，当我们判断该元素是否存在时，需要对这些属性分别进行判断。即每一种属性都对应一个位数组，假设数据有n维，则采用n组hash函数对每一维进行处理。将结果取AND运算，只有每个属性都为1时才能判断该种元素有可能存在，否则一旦有一个属性的位数组为0，那么该元素必定不存在。同时易知，多维数据属性显然也存在false positive。



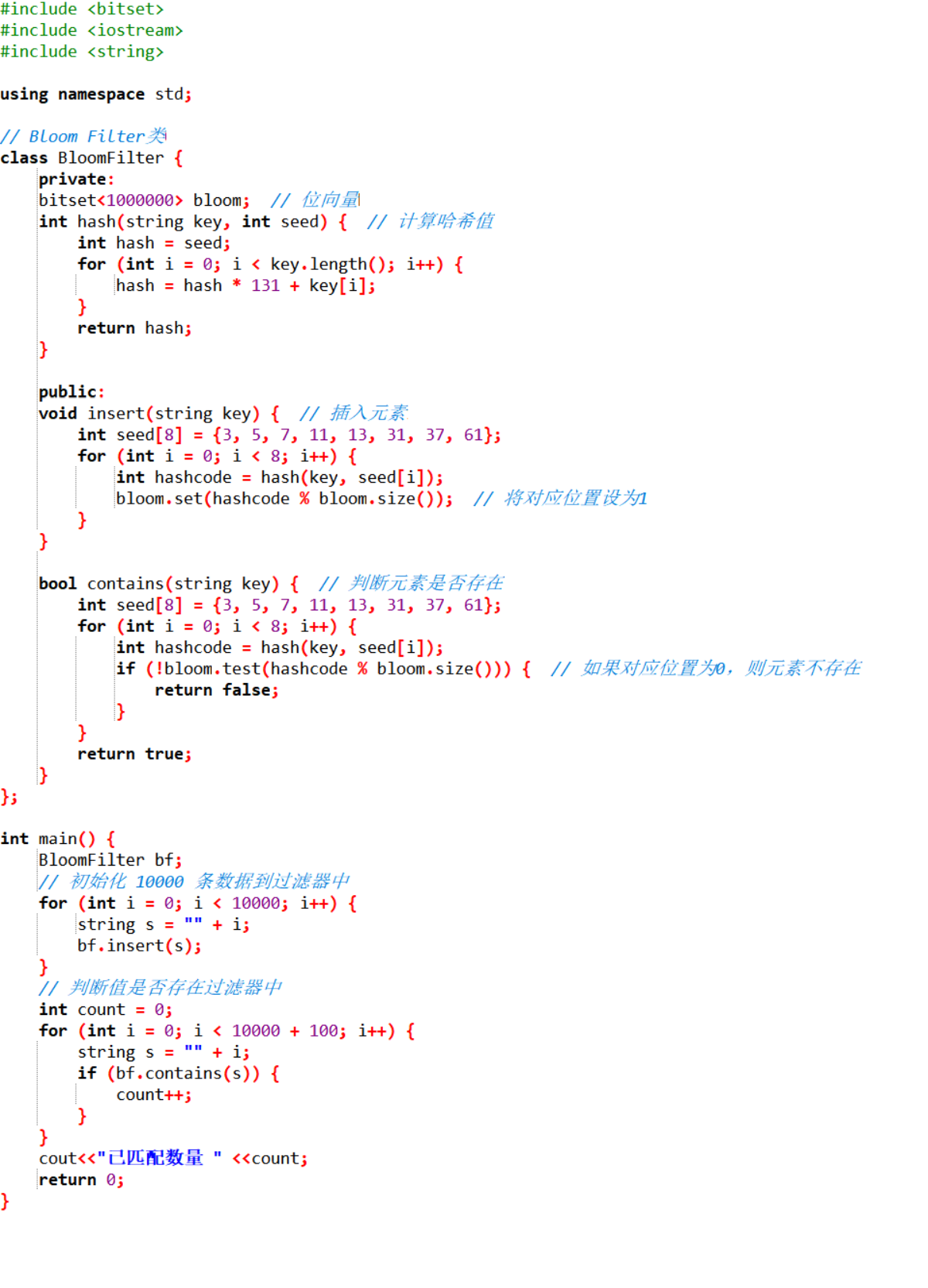
# 四、实验设计

本实验实现了一个简易Bloom Filter，并实现了插入元素、查询元素功能。

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简易的Bloom Filter，代码如下：



# 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BloomFilter类中的位向量bloom大小（即位数组的大小）。

位数组大小为1000000，因此其空间开销为1000000个比特位，即125000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

改变数据大小，通过对比发现，当哈希函数个数为8时：

数据量为1000时误判率约为0.278；

数据量为10000时误判率约为0.083；

数据量为100000时误判率约为0.028；

观察结果可知，当插入数据量不断增大的时候，误判率有所降低，易知Bloom Filter很适合大数据时代的使用。

# 六、课程总结

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。

同时，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] 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.