2019 级

《大数据存储与管理》课程

课 程 报 告

姓　名 陈昱夫

学　号 U201990056

班　号 CS1903 班

日　期 2022.04.18

目　　录

一、摘要

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：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，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。

本文分析了现有多维布鲁姆过滤器查询算法(MDBF)工作原理，提出了一种改进的两步表示和查询的联合多 维布鲁姆过滤器(CMDBF)查询算法。CMDBF 新增一个用于表示元素整体的联合布鲁姆过滤器 CBF，CMDBF 中元素表示和查找分两步进行。将 MDBF 的各属性的表示和查询作为第一步，第二步联合元素所有属性域，利用 CBF 完成元素整体的表示和查询确认。理论分析和仿真实验结果表明，CMDBF 能够支持多维集合元素的简洁表示和查询，相比 MDBF 查询误判率降低明显。

二、选题背景与意义

Bloom filter（布隆过滤器）是 Howard Bloom 在 1970 年提出的二进制向量 数据结构，具有良好的空间和时间效率，用于检测某元素是否为集合的成员。

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。它是一个判断元素是否存在集合的快速的概率算法。Bloom Filter 有可能会出现错误判断，但不会漏掉判断。也就是 Bloom Filter 判断元素不在集合，那肯定不在。如果判断元素存在集合中，有一定的概率判断错误。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 比其他常见的算法（如 hash 折半查找）极大节省了空间。

它的优点是空间效率和查询时间都远远超过一般的算法，通过极少的错误换取了存储空间的极大节省，而缺点是有一定的误识别率和删除困难。尤其当要存储的信息为多维时，靠简单的多维 bloom filter 的叠加会让 false positive 的概率大大提升。

通过本课题的研究，分析一个可以减少 false positive 的概率的结构设计。

三、总体设计

3.1 Bloom Filter 基本思想

3.1.1 BloomFilter 原理：

当一个元素被加入集合时，通过 k 个散列函数将这个元素映像成一个位数组中的 k 个点，把它们置为 1。检索时，我们只要看看这些点是不是都是 1 就（大约）知道集合中有没有它了，如果这些点有任何一个 0，则被检元素一定不在；如果都是 1，则被检元素很可能在。

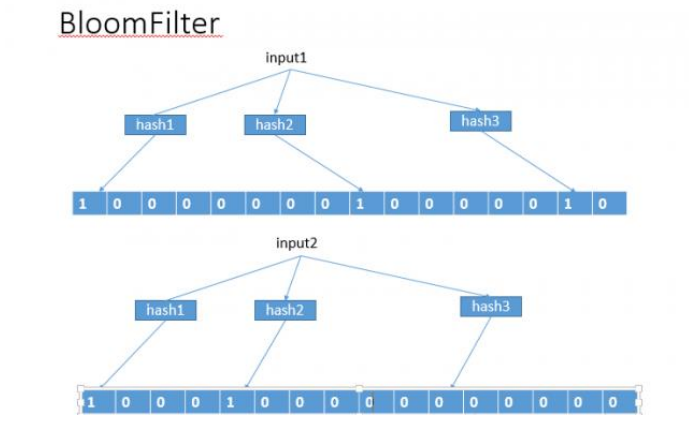

图 3-1 BloomFliter 原理图

如上图所示，我们定义了一个 16 位的二进制向量，3 个 hash 函数，这个 3 个函数 hash 的结果为 0 或者 1，该结果存放的位置为 0~15 之间，将 hash 的结果的位置映像到二进制向量的 index，并保存结果.

对于输入数据 input1，得到的的结果存在于 0，8，14，结果全部为 1，那么说明 input1 可能存在于指定的集合。

对于 input2，得到的结果存在于 0，4，10，有一个是 0，那么说明 input2 一定不存在于指定的集合。

3.1.2 Bloom Filter 的性能分析

分析 Bloom Filter 的性能问题即 False Positive 的比率 f 的问题，探讨何时 f 才能最小。

初始状态时，如图 2，二进制数组的 m 位均为 0，此时进行一次 Hash，则某一位为 0 的概率是$\frac{m-1}{m}$（只有 1 位为 1，且假设 Hash 函数计算结果在每一位的概率均等），因此，对 n 个元素进行 k 次 Hash，则某一位为 0 的概率 p 为:

$$\mathrm{p}=(1-\frac{1}{m})^{nk}$$

对上式做变换，并利用利用自然指数极限的代换，有：

$$\mathrm{p}=(1-\frac{1}{m})^{nk}=(1-\frac{1}{m})^{m\cdot\frac{nk}{m}}\approx e^{-\frac{nk}{m}}$$

一个 False Positive 发生，即一个不在集合的数却被判定在集合中的概率，是在集合中任选 k 个数，其结果均为 1 的概率，该概率即为 False Positive 的比率 f，计算如下：

$$\mathrm{f}=(1-p)^k\approx\left(1-e^{-\frac{nk}{n}}\right)^k=e^{kln\left(1-e^{-\frac{nk}{m}}\right)}$$

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

上式中，e^{x}为单调递增函数，因此当其指数最小时，取最小值，也即$\ln(p)\ln(1-\mathrm{p})$取最大值，此时有 $\mathrm{p}=\frac{1}{2}$，即$e^{-\frac{nk}{m}}=\frac{1}{2}$，$\mathrm{k}=\frac{m}{n}ln2\approx0.7\frac{m}{n}$，此时给出的 f 为：

$$\mathrm{f}=(\frac{1}{2})^k\approx0.6185^{\frac{m}{n}}$$

3.1.3 Bloom Filter 的不足

通过上面的分析，我们可以看到 Bloom Filter 高效的查找与优越的性能，但是也能看到几个明显的问题。

（1）无法删除集合中的元素

Bloom Filter 在插入元素时，对于 Hash 计算结果处已经被标记为 1 的位是不做操作的，所以在删除时就会出现问题，如果直接将该元素经过 Hash 计算后的位置置为 0，则会牵动到其他元素。

（2）Hash 函数的选择会影响到算法的结果

根据前面的错误率计算，当哈希函数个数$\mathrm{k}=\ln2\times\frac{m}{n}$时错误率最小，在实际的应用中，只有 n 是固定的，我们要综合考虑 m 和 k 的选择问题以及哈希函数的设计问题。

3.2 数据结构设计

3.2.1 多维布鲁姆过滤器 MDBF

针对多维元素的表示和查询问题，目前存在一种 MDBF 的方案。该方案采用和元素维数相同的多个标准布鲁姆过滤器组成，直接将多维元素的表示和查询分解为单属性值子集合的表示查询，元素的维数有多 少，就采用多少个标准的布鲁姆过滤器分别表示 各自对应的属性。进行元素查询时，通过判断多 维元素的各个属性值是否都在相应的标准布鲁 姆过滤器中来判断元素是否属于集合。如下图 3 所示。

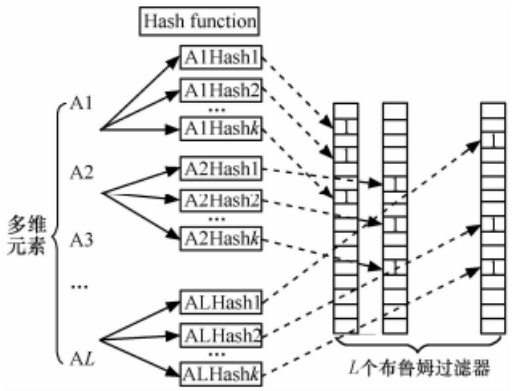

图 3 MDBF 结构

对于{n,m,k,L}的 MDBF，判断元素是否从属集 合，需要判断所有的属性值是否在对应的属性字集 合，多维布鲁姆过滤器误判率为

$$f^{\mathrm{MDBF}}(m,k,n,L)=\prod_{i=1}^{L} f^{\mathrm{BF}}(m,k,n)=\left(f^{\mathrm{BF}}(m,k,n)\right)^{L}$$

多维布鲁姆过滤器查询时间为 O(k×L)，所需空 间为 O(m×L)。

通过一个二维元素表示和查询的例字来分析 现有 MDBF 误判率情况。经过分析指出 MDBF 算 法在元素表示和查询时的缺陷，进而给出改进的 CMDBF 查询算法。

例 2 多维布鲁姆过滤器查询算法实例。设二 维元素为 x={A1,A2}，其中 A1 和 A2 是 2 个不同的属 性。使用 MDBF 需要 2 个标准布鲁姆过滤器 BF1、 BF2 用来分别表示属性 A1 和属性 A2。单属性域的布 鲁姆过滤器向量都取 m=8 bit。每次映射和查找的散 列函数的个数为 2 个，2 个属性值映射的散列函数取 一致，简单定义这 2 个散列函数为：h1(x)=x mod 8 和 h2(x)=(2x+3) mod 8。其中数据集合是{(9,7), (11,9)}，需要查询的元素是(11,15)。 表示集合之前，2 个向量都需要初始化。元素(9,7) 插入后，第一维布鲁姆过滤器向量 BF1[1]和 BF1[5] 置位，第二维布鲁姆过滤器向量 BF2[7]和 BF2[1]置 位，两向量状态分别如图 4 第 2 行所示。那么元素 (11,9)插入后的向量状态如图 4 第 3 行所示。

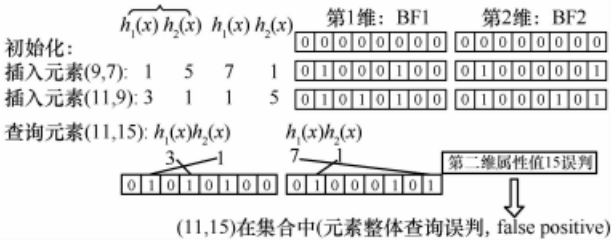

图 4 MDBF 算法实例

查询(11,15)是否在集合中。首先检查 11 是否在 第一个属性集合中，11 对应的 2 个散列地址 3、1， 发现 BF1[3]，BF1[1]都已置位，说明 11 在属性 A1 集合中。然后检查 15 的 2 个散列地址，发现 BF2[7]， BF2[1]也已经置位，说明 15 也在属性 A2 的集合中， 得出(11,15)在集合中的错误结论。

文献[17]提出了 MDBF 算法，并认为“MDBF 出 现误判的情况是每维都出现误判断时，元素才会被误 判断”。通过上面的实例分析发现：实际上，MDBF 算法在任何一维误判断时都会导致元素的误判断。如 例 2 所示，两维元素(11,15)进行查询判断时，第二维 15 却不在第二维字集中，但是使用 BF2 误判断 15 在该字集中，此时就出现了两维元素由于一维的误判而 导致元素整体的误判。成为 MDBF 的缺陷。

上述结果的产生，是因为 MDBF 使用每个单独 的 BF 来表示元素的每个单独的属性值，没有相应 的结构将各属性值合成的元素表达出来。MDBF 分 割了各个属性值属于元素一体的特点，仅通过单独 判断元素的各个属性值是否在对应的字集合来进 行元素的从属判断，不可避免发生例 2 的情况，将 不属于集合的元素(11,15)误判为属于集合，因此有 必要对 MDBF 进行改进。

3.2.2 联合多维布鲁姆过滤器结构(CMDBF)

考虑到如果能将各个属性合为一体的特性在布鲁 姆过滤器中表示出来，得到元素的整体信息必然可以 减少由于单维属性误判而导致的元素整体误判的可能性。由此提出改进的联合多维布鲁姆过滤器(CMDBF, combine multi-dimensional Bloom filters)。CMDBF 由 两部分过滤器组成，第一部分是用标准布鲁姆过滤器 表示的各属性字集，采用和 MDBF 一样的机制；第二 部分是一个联合布鲁姆过滤器 CBF(combine blooms filter)，用来表示各个属性值的联合。这里存在如何将 元素表示到 CBF 的问题，如果 MDBF 每次映射散列地址范围一致，那么可以直接用不同属性域的对应散 列映射地址进行异或运算，通过 2 次散列运算获得元 素在 CBF 中的位置，将 CBF 对应位置置位，完成元 素到 CBF 的表示。下面用一个例字来解释算法思路， 如图 5 所示，其中，⊕为异或运算。

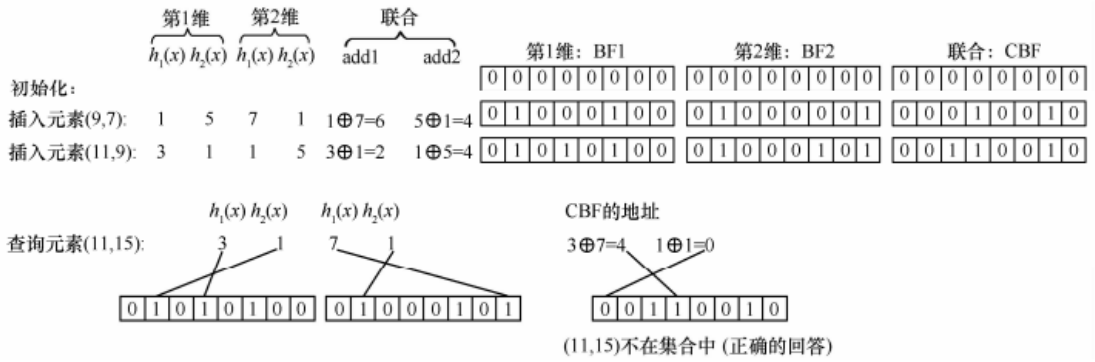

图 5 CMDBF 算法实例

图 5

例 3 用 CMDBF 表示的二维集合{(9,7), (11,9)}，查询(11,15)是否在集合中。 表示集合之前，3 个向量都需要初始化。2 个 元素插入后，BF1 和 BF2 状态和例 2 中的 MDBF 完 全相同。所不同的是，改进的多维布鲁姆过滤器中， 增加了一个联合过滤器 CBF，元素(9,7)在 CBF 的 映射地址可由单属性 9,7 在 BF1 和 BF2 中的地址直 接计算，如 addr1=1⊕7=6(这里的 1 是属性“9”在 BF1 中的地址，7 是属性“7”在 BF1 中的地址，addr1 是由 2 个属性“9”，“7”共同决定)，addr2=5⊕1=4。 因此将 CBF[6]和 CBF[4]置位，完成元素(9,7)的插入。同理完成元素(11,9)到 CMDBF 的表示，状态如 图 5 第 3 行所示。

查询(11,15)是否是集合的元素。在 MDBF 的基础 上，还需要进行元素整体是否在 CBF 的检查，元素 (11,15)在 CBF 中的 2 个映射地址分为 addr1=3⊕7=4，addr2=1⊕1=0。虽然 CBF[4]=1，但是 CBF[0]=0，得 出元素(11,15)不在集合中的正确结论。 虽然 CMDBF 算法只在 MDBF 算法上增加了一 个用于表示各属性联合的 CBF 过滤器，但元素在 CBF 的映射位置由所有的属性值共同决定，有效解决由于 单属性的误判断而造成元素整体误判断的可能。

设多维元素 x={A1,A2,…,AL}，使用布鲁姆过滤 器，虽然对于每维属性值，可能需要选择不同的散 列函数，但是仍然可以设计每维属性使用相同的散 列函数个数 ki=k(1≤i≤L)，每个属性维数使用相同 的长度的过滤器向量 mi=m(1≤i≤L)。那么每个属 性的散列映射地址为

$$\text{attr}j\_\text{addr}i = \text{Hash}[i](x.A_j) \in \{0,1,\cdots,m-1\}$$
$$(1 \leqslant j \leqslant L, 1 \leqslant i \leqslant k)$$

和 MDBF 一样，attrj_addri{0,1,…,m−1}就是属性 j 的第 i 个散列地址。这就为每维属性值的联合提供 了方便，可以通过这些映射地址的运算完成元素的多属性联合表示，可定义联合布鲁姆过滤器 CBF 的散列函数为

$$\text{Hash}i(x) = \text{attr}1\_\text{addr}i \oplus \text{attr}2\_\text{addr}i \oplus \cdots \oplus$$
$$\text{attr}L\_\text{addr}i = \text{addr}i \quad (1 \leqslant i \leqslant k)$$

addri(1≤i≤k)是元素在 CBF 中的第 i 个映射地 址，由各个属性值的映射地址通过异或运算得来， 由多个属性联合确定。可以表现元素各属性值一体 的特性，而且异或函数也是常用的散列处理函数。 由 attrj_addri 易得 addri，这样定义的 CBF 向量长度 仍然是 m。

CMDBF 在 MDBF 基础上增加一个联合的布鲁 姆过滤器 CBF，而新增的 CBF 又由元素的所有属 性值确定，这样元素整体就可以通过 2 次散列运算 表示出来，完成元素的表示和查找，如图 6 所示。

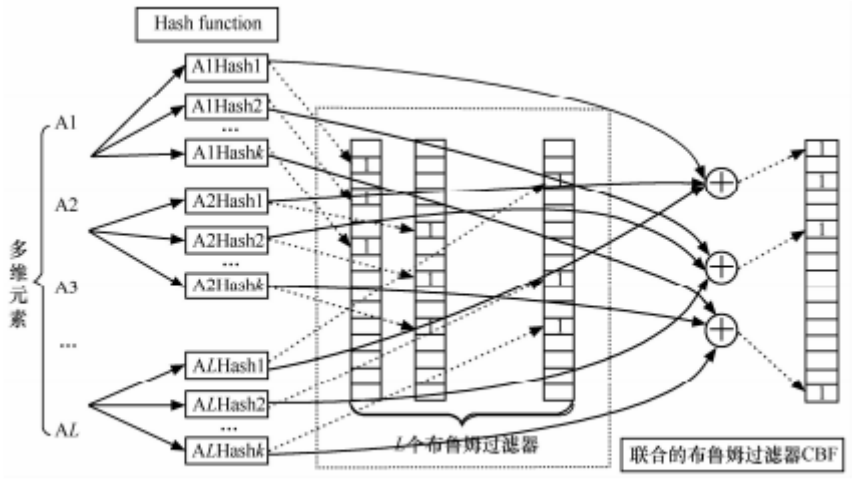

图 6　CMDBF 算法原理

3.3 操作流程

```
Algorithml AddElement (Object Element)
Input: An object element with multi-attribute
Output: void
int Addr[k], CAddr[k]
For(i=0;i<Element.adttr_Length;i++)
    BF= GetBF(i)                          //获得该维对应的过滤器
    Hash= GetHash(i)                      //获得第i维属性对应的k个Hash函数
    For( j=0; j<k; j++)
        Addr[j]=Hashj(Element attr[i])//得到单维布鲁姆过滤器地址
        BF[Addr[j]]=1                     //将对应过滤器置位
        CAddr[j]=CAddr[j]^Addr[j]//计算CBF的散列地址
    End For
End For
For( j=1, j<k+1; j++)
    CBF[CAddr[j]]=1                       //将CBF布鲁姆过滤器置位
End For
```

元素插入

```
Algorithm2 MembershipQuery(Object Element)
Input: An object element with multi-attribute
Output: If element is a member of Set A, response true, else response false
int Addr[k], CAddr[k]
For(i=0; i<Element.attr_Length; i++)
    BF= GetBF(i)                          //获得该维对应的过滤器
    Hash=GetHash(i)                       //获得第i维属性对应的k个hash函数
    For(j=0;j<k;j++)
        Addr[j]=hashj(Element.attr[i])//得到单维布鲁姆过滤器地址
        If(BF[Addr[j]]==0)                //判断是否置位
            Return false
        End if
        CAddr[j]=CAddr[j]^Addr[j]//计算CBF的哈希地址
    End For
End For
For(j=1; j<k+1; j++)
    If( CBF[CAddr[j]]==0)                 //判断是否置位
        Return false
    End If
End For
Return true
```

元素查询

四、理论分析

定理 1 CMDBF 算法的查询误判率小于 MDBF 算法。 证明 对于{n,m,k,L}的CMDBF，判断元素是 否从属集合，需要第一轮 MDBF 的 L 个布鲁姆过滤 器和第二轮联合布鲁姆过滤器对应位置都已置位。 第二轮联合布鲁姆过滤器 CBF 的误判率可以 直接计算为

$$f^{\mathrm{CBF}}(m,k,n)=(1-p)^{k}=\exp\left(k\ln\left(1-e^{-kn/m}\right)\right)$$
$$=f^{\mathrm{BF}}(m,k,n)$$

则 CMDBF 的误判率为

$$f^{\mathrm{CMDBF}}(m,k,n,L)=f^{\mathrm{MDBF}}(m,k,n,L)\,f^{\mathrm{CBF}}(m,k,n)$$
$$=\left(f^{\mathrm{BF}}(m,k,n)\right)^{L+1}$$

由于 0＜f BF (m,k,n)＜1，由式(3)和式(7)显然有 f CMDB ＜f MDBF 。 推广到更一般的情况，当进行元素到 CBF 的 表示时，如采用其他 attrj_addri 到 addri 的转换 方法，此时 CBF 的长度可能并不是 m，设为 m1， 由式(2)知，0＜f CBF (m1,k,n)＜1，式(7)仍然可以得 出 f CMDB ＜ f MDBF 的结论。 虽然 CMDBF 采用多于 MDBF 一个 m bit 长度 的向量空间为代价，但是能够将元素属性值表达

成 一个整体，可以得到比 MDBF 更低的查询误判率， 在第 4 节的实验分析中，将得出这种空间消耗对整个算法的性能影响并不大。

五、实验测试

由于能力有限，只是根据上述的算法流程写了个能满足上述 3.2.2 中例二的基本 CMDBF 结构。附录附上源代码。

六、结语

本文首先通过对普通的多维 Bloom Filter(MDBF) 的流程分析，指出其存在的问题。

接着针对 Bloom Filter 存在的问题进行改进，提出了通过增加一个 CBF 来记录多维信息间的联系，来解决问题。

综上所述，本文中改进的 CMDBF 尽管增加了一点存储信息的空间开销，但却减少了 false positive 的概率。

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.

[8] 联合多维布鲁姆过滤器查询算法 谢鲲[1,4],秦拯[2],文吉刚[1],张大方[2],谢高岗[3]

附录

CMDBF 程序：

```
//BF.h

#pragma once
#include <vector>
using namespace std;

const int k = 8;        //filter length

vector<int> Addr(k);
vector<int> CAddr(k);
class BloomFilter {

private:
	vector<int> BF;
public:
	BloomFilter();

	void setValue(int pos);
	int getValue(int pos);
};

BloomFilter::BloomFilter(){}

void BloomFilter::setValue(int pos) {
	BF[pos] = 1;
}

int BloomFilter::getValue(int pos) {
	return BF[pos];
}


//CMDBF.h
#pragma once
#include "BF.h"
#include <iostream>
class Mutiple_Dimensions_BloomFilter {
public:
	BloomFilter MDBF[2];
	Mutiple_Dimensions_BloomFilter(int dimensions);
	BloomFilter GetBF(int i);
};

Mutiple_Dimensions_BloomFilter::Mutiple_Dimensions_BloomFilter(int dimensions) {
	for (int i = 0; i < dimensions; i++) {
		MDBF[i] = BloomFilter();
	}
}



BloomFilter Mutiple_Dimensions_BloomFilter::GetBF(int i) {
	return MDBF[i];
}
```

```
//query.h

#pragma once
#include "BF.h"
#include "CMDBF.h"
#include "hashkernel.h"
#include <iostream>
bool query_index(vector<int> element, Mutiple_Dimensions_BloomFilter* CMDBF, BloomFilter* CBF);
bool query_index(vector<int> element, Mutiple_Dimensions_BloomFilter* CMDBF, BloomFilter* CBF) {
	for (int i = 0; i < 2; i++) {
		BloomFilter BF = CMDBF->GetBF(i);
		for (int j = 0; j < x; j++) {	//简易测试为 2 个哈希函数
			switch (j) {
			case 0:
				Addr[j] = hash_1(element[i]);
				break;
			case 1:
				Addr[j] = hash_1(element[i]);
				break;
			}

			if (BF.getValue(Addr[j]) == 0)
				return false;
			CAddr[j] = CAddr[j] ^ Addr[j];
		}
	}

	for (int j = 0; j < x; j++) {	//k?
		if (CBF->getValue(CAddr[j]) == 0)
			return false;
	}

	return true;
}



//insert.h
#pragma once
#include "BF.h"
#include "CMDBF.h"
#include "hashkernel.h"
#include <iostream>
int insert(vector<int> element, Mutiple_Dimensions_BloomFilter* CMDBF, BloomFilter* CBF);
int insert(vector<int> element, Mutiple_Dimensions_BloomFilter* CMDBF, BloomFilter* CBF) {


	for (int i = 0; i < 2; i++) {
		for (int j = 0; j < x; j++) {	//简易测试为 2 个哈希函数
			switch (j) {
			case 0:
				Addr[j] = hash_1(element[i]);
				break;
			case 1:
```

```
                    Addr[j] = hash_1(element[i]);
                    break;
                }

                (CMDBF->GetBF(i)).setValue(Addr[j]);
                CAddr[j] = CAddr[j] ^ Addr[j];
            }
        }


        for (int j = 0; j < x; j++) {
            CBF->setValue(CAddr[j]);
        }

        return 0;
}


//hashkernel.h
#pragma once
const int x = 2;        //num of hash function
//如何根据循环计数器 i，来对应调用 hashi
int hash_1(int val) {
        return val % 8;
}

int hash_2(int val) {
        return (val * 2 + 3) % 8;
}
```

```
//main.cpp
#pragma once
#include "BF.h"
#include "CMDBF.h"
#include "hashkernel.h"
#include "insert.h"
#include "query.h"
#include <iostream>

int main() {
        bool test = true;
        cout << test << endl;
        vector<int> element;      //维数

        Mutiple_Dimensions_BloomFilter* CMDBF = new Mutiple_Dimensions_BloomFilter(2);

        BloomFilter* CBF = new BloomFilter();      //异或过滤器

        for (int j = 0; j < 2; j++) {
            int i = 0;
            while (i<2) {
                int temp = 0;
                cin >> temp;
                element.push_back(temp);
                i++;
            }
            element.clear();
```

```
            insert(element, CMDBF, CBF);
        }

        int i = 0;
        bool result;
        while (i<2) {
            //cin >> element[i];
            int temp = 0;
            cin >> temp;
            element.push_back(temp);
            i++;
        }
        result = query_index(element, CMDBF, CBF);
        cout << result;
}
```