2020 级

《大数据存储与管理》课程报告

多维数据的 Bloom Filter 设计与实现

姓　名　胡鹏飞

学　号　U202015514

班　号　计算机 2007 班

日　期　2023.05.22

目　　录

一、Bloom Filter

1．Bloom Filter 简介

Bloom filter 是一种二进制向量数据结构，由 Howard Bloom 在 1970 年提出，用于检测元素是否属于某个集合。它通过哈希函数将元素映射到位数组中，并判断相应的位是否被标记为 1，从而判断元素是否存在于集合中。Bloom filter 被视为一种利用空间换取时间的策略，用于优化算法中判断集合成员存在性的问题。同时，Bloom filter 也是 Bit-Map 方法的一个典型例子，可以极大地节省存储空间，但也可能存在误判的风险。因此，在对错误率能够容忍的应用场景下，Bloom filter 具有空间效率高、时间效率低的特点，是一种非常实用的数据结构。

2．Bloom Filter 原理

Bloom Filter 其实就是多个哈希函数的对应。在人们判断元素是否存在这个问题时，在随着数据持续性增大的条件下，总会出现两个元素的哈希值相等的情况，在这时，我们就无法通过观测这两个元素的哈希值对应的值是否为 1 来判断这两个元素其中一个是否存在（可能只有一个存在，或者两个都存在）。而 Bloom Filter 就是多个哈希函数对应哈希值的翻版。假设我们有着 S={x1, x2,…,xn}这 n 个元素，又有着 k 个相互独立的哈希函数。那么我们对 S 中每一个元素都进行相应的哈希求值，那么最多的情况下我们就需要 k*n = m 个元素的数组来进行哈希值的存储。因此 m <= k*n 的。

在进行 k 个哈希值进行对应计算后，我们的 BloomFilter 中的值就要进行相应的值的改变。这里假设 k=3，如下图所示。

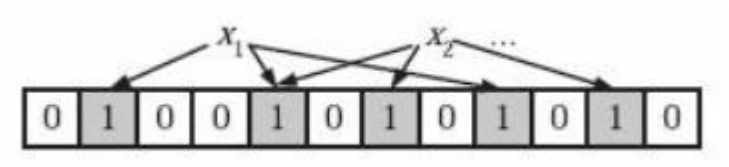

图 一-1 Bloom Filter set

在图中，x1 和 x2 中有着一处不同哈希计算可是最终得到的哈希值相等的情况，在 BloomFilter 中，位数组只对第一次出现的哈希值对应的位置进行置 1，之后出现这个哈希值我们都不需要其他操作，每有一处重复，我们的最大 kn 的大小都要进行减一。

但是尽管我们使用了多个哈希函数，还是有可能在元素数量巨大的情况下出现两个元素的 k 个哈希值一致的情况。所以，BloomFilter 并不能保证数据检测的完全正确。因此，我们有着这样的结论，如果我们在 BloomFilter 中找到一个元素对应的哈希值都存在，那我们只能认为他可能存在，即存在或者他是一个 false positive。但是只要其中一个元素的对应哈希值不存在，那么这个元素肯定就不存

在。

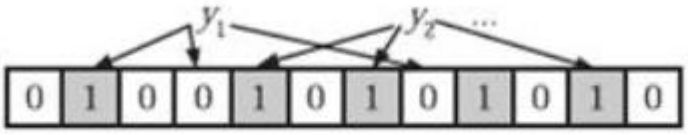

图 一-2 1．Bloom Filter test

图中，y1 由于有两处哈希值对应为 0，那么它就肯定不存在，而 y2 对应的三个哈希值都为 1，我们也只能说它可能存在。

在海量的数据之中，利用 BloomFilter 具有非常有效的判断元素是否存在的效率，在允许一定的误差的条件下，BloomFilter 无疑是一个好的判断元素是否存在的选择。比如网络爬虫，我们判断一个网站的 URL 是否被爬取过，利用 BloomFilter 就有着很高的效率，因为我们允许有一些网站不被爬取。

3．Bloom Filter 分析

前面我们已经提到了，Bloom Filter 在判断一个元素是否属于它表示的集合时会有一定的错误率（false positive rate），下面我们就来估计错误率的大小。

首先我们对于 m 位的位数组，有一个哈希函数选中其中一位的概率为 1/m。而当集合 S={x1, x2,…,xn}的所有元素都被 k 个哈希函数映射到 m 位的位数组中时，这个位数组中某一位还是 0 的概率是：

$$p' = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}.$$

公式 1

其中的不等式由相应的微积分的知识的计算得到，这里并不赘述。ρ 为位数组中 0 的比例，则 ρ 的数学期望 E(ρ)= p’。在 ρ 已知的情况下，要求的错误率（false positive rate）为：

$$(1-\rho)^k \approx (1-p')^k \approx (1-p)^k.$$

公式 2

二、多维数据的 Bloom Filter

1．基本原理

针对多维元素的表示和查询问题，目前存在一种多维布鲁姆过滤器(MDBF)解决方案。MDBF 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，如图所示。

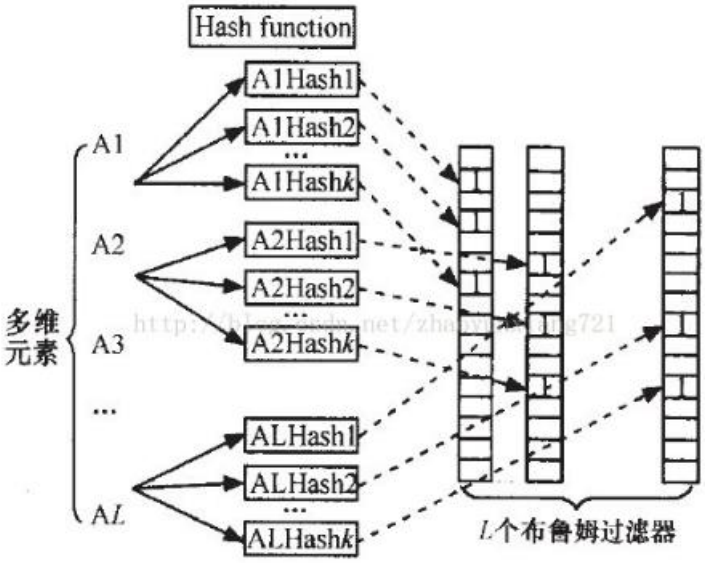

图 二-1 MDBF

2．缺点

除了一维元素哈希值可能出现相同导致假阳性外，MDBF 还会出现由于不同维度的元素值分别出现过而导致错误判断存在。

例如对于一个加入过(a,b),(c,d)元素的 MDBF，对(a,d),(b,c)就会产生误判。

三、多维数据的 Bloom Filter 具体设计与实现

1．设计

- **数据结构**

假设每个元组具有两个维度，使用两个 n 位 bitset 记录信息。

- **Hash 函数**

k 值设置为 3，分别使用三种不同的哈希算法进行计算来降低冲突，分别为 BKDR 哈希 、AP 哈希、DJB 哈希。

- **映射方法**

对 MDBF 中的映射方法进行改良，第一维元素 hash 值正常计算，而第二维元素的 hash 值额外乘以第一维元素的 hash 值，具体操作为：

A2Hash1’=A2Hash1*A1Hash2;

A2Hash2’=A2Hash2*A1Hash3;

A2Hash3’=A2Hash3*A1Hash1;

2．实现

- **数据结构**

```
template<size_t N, class K = string, class Hash1 = HashBKDR, class Hash2 = HashAP, class Hash3 = HashDJB>
class BloomFilter
{
public:
    void Set(const K& key1, const K& key2) { ... }
    bool Tests(const K& key1, const K& key2) { ... }
private:
    bitset<N> bitset1;
    bitset<N> bitset2;
};
```

图 三-1 Bloom Filter 数据结构

- **Hash 函数**

```
//BKDR哈希
struct HashBKDR
{
    size_t operator()(const string& s)
    {
        size_t value = 0;
        for (auto e : s)
        {
            value += e;
            value *= 131;
        }

        return value;
    }
};
```

图 三-2 hash1

```
//AP哈希
struct HashAP
{
    size_t operator()(const string& s)
    {
        register size_t hash = 0;
        size_t ch;
        for (long i = 0; i < s.size(); i++)
        {
            ch = s[i];
            if ((i & 1) == 0)
            {
                hash ^= ((hash << 7) ^ ch ^ (hash >> 3));
            }
            else
            {
                hash ^= (~(hash << 11) ^ ch ^ (hash >> 5));
            }
        }

        return hash;
    }
};
```

图 三-3 hash2

```
//DJB哈希
struct HashDJB
{
    size_t operator()(const string& s)
    {
        register size_t hash = 5381;
        for (auto e : s)
        {
            hash += (hash << 5) + e;
        }

        return hash;
    }
};
```

图 三-4 hash3

- **Set 方法**

```
void Set(const K& key1, const K& key2)
{
    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;
    /*cout << i1 << " " << i2 << " " << i3 << ';';
    cout << i4 << " " << i5 << " " << i6 << ';' << endl;*/
    bitset1.set(i1);
    bitset1.set(i2);
    bitset1.set(i3);
    bitset2.set(i4);
    bitset2.set(i5);
    bitset2.set(i6);
}
```

图 三-5 set

- **test 方法**

```
bool Tests(const K& key1, const K& key2)
{
    size_t i1 = Hash1()(key1) % N;
    if (bitset1.test(i1) == 0)
    {
        return false;
    }
    size_t i2 = Hash2()(key1) % N;
    if (bitset1.test(i2) == 0)
    {
        return false;
    }
    size_t i3 = Hash3()(key1) % N;
    if (bitset1.test(i3) == 0)
    {
        return false;
    }
    size_t i4 = (Hash1()(key2)*i2) % N;
    if (bitset2.test(i4) == 0)
    {
        return false;
    }
    size_t i5 = (Hash2()(key2)*i3) % N;
    if (bitset2.test(i5) == 0)
    {
        return false;
    }
    size_t i6 = (Hash3()(key2)*i1) % N;
    if (bitset2.test(i6) == 0)
    {
        return false;
    }
    return true;
}
```

图 三-6 test

四、测试

1. 针对性测试

针对二.2 中提到的缺点进行测试。

```
void TestBloom_test()
{
    BloomFilter<1000> bf;
    bf.Set("123", "aaa");
    bf.Set("456", "bbb");
    cout << bf.Tests("123", "bbb");
}
```

图 四-1 针对性测试

```
0
C:\Users\胡鹏飞\Desktop\布隆过滤器\x64\Debug\布隆过滤器.exe (进程 10036)已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口. . .
```

图 四-2 测试 1 结果

测试结果正确。

2. 性能测试

数据来源：All The Universities In The World | Kaggle

测试方法：每一组数据都是不同的，在插入每一组数据前，先检查是否存在，如果存在，说明出现错误。

测试数据（共 9361 行数据）：

Bloom filter 大小	10000	20000	30000	40000	50000
出错数	2262	528	145	57	33
出错率	24.16%	5.64%	1.55%	0.61%	0.35%
运行时间/秒	0.064	0.054	0.056	0.054	0.053

五、总结

本次实验完成了多维数据 Bloom Filter 的设计以及使用 c++将其实现。

通过本次实验，我深入了解了 Bloom Filter 的作用以及原理，并结合自己的理解进行了改进。同时也通过自己的探索得到了一些数据集的获取途径。

最后，感谢老师的悉心教导。

参考文献

•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.
• 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.
• S. Dharmapurikar, P. Krishnamurthy, and D.E. Taylor, “Longest Prefix Matching Using Bloom Filters,” Proc. ACM SIGCOMM, pp. 201-212, 2003.
• 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.
• 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.
• 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.
• D. Guo, J. Wu, H. Chen, and X. Luo, “Theory and Network Application of Dynamic Bloom Filters,” Proc. IEEE INFOCOM, 2006.