



2019 级

基于 Bloom Filter 的设计

实 验 报 告

姓 名 王越

学 号 U201910930

班 号 计算机校交 1902 班

日 期 2022.04.18

目 录

一、实验目的.....	2
二、实验背景.....	2
三、实验内容.....	3
3.1 Bloom Filter 结构简介.....	3
3.2 false positive 概率推导	3
3.3 Bloom Filter 的多维数据属性表示.....	4
四、实验设计	4
4.1 R 树和 Bloom Filter 相结合的索引结构 RBF.....	5
4.2 更新缓存结构	6
4.3 点查询	6
五、性能测试.....	7
5.1 误判率研究	7
5.2 空间开销研究	7
5.3 查询延迟研究	8
5.4 普通 bloom filter 测试	8
5.5 RBF 点查询延迟	8
六、实验总结	9
参考文献.....	10

一、实验目的

1. 分析 bloom filter 的设计结构和操作流程;
2. 理论分析 false positive;
3. 多维数据属性表示和索引 (系数 0.8)
4. 实验性能查询延迟, 空间开销, 错误率的分析。

二、实验背景

随着社会对信息存储需求的增长, 大规模存储系统的应用越来越广泛, 存储容量也从以前的 TB (Terabyte) 级上升到 PB (Petabyte) 级甚至 EB (Exabyte) 级。查找和处理文件变得越来越困难。现有的基于层次目录树结构的数据存储系统的扩展性和功能性不能有效地满足大规模文件系统中快速增长的数据量和复杂元数据查询的需求。为了有效地处理这些快速增长的数据, 迫切需要提供快速有效的数据管理系统来帮助用户更好的理解 and 处理文件。

元数据 (metadata) 是关于数据的数据, 是关于信息资源的形式、主要内容、数据的特征和属性、数据的使用者、使用和修改记录、存放位置等信息的集合。在文件系统中元数据用于描述和索引文件, 例如超级块信息, 记录全局文件信息如文件系统的大小, 可用空间等; 索引节点信息, 记录文件类型、文件的链接数目、用户 ID、组 ID、文件大小、访问时间、修改时间、文件使用的磁盘块数目等; 目录块信息, 记录文件名和文件索引节点号等。有效的管理这些元数据并提供各种查询接口, 能帮助用户更好的理解 and 处理数据。

一般来讲, 尽管存储系统中元数据的数据量远小于整个系统的存储容量, 文件系统中元数据占用的空间往往不到 10%, 但元数据操作是整个文件系统操作的 50%-80%, 所以元数据的高效管理十分必要。

Bloom Filter 是 1970 年由布隆提出的, 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节省。它由一个很长的

二进制向量数组和一系列随机映射函数组成，只需要哈希表的 $1/8$ 到 $1/4$ 的大小就能解决同样规模的集合的查询问题。

三、实验内容

3.1 Bloom Filter 结构简介

为了表示一个具有 n 个元素的集合 $S=\{s_1, s_2, \dots, s_n\}$ ，需要一个长度为 m 的二进制向量数组来记录 Bloom filter($m>n$)，初始化数组的每一个元素的值为 0；并使用 k 个相互独立的哈希函数 h_1, h_2, \dots, h_k ，它们的域值均为 $\{0, 1, \dots, m-1\}$ 。对于每一个元素 $s \in S$ ，将数组中对应于 $h_1(s), h_2(s), \dots, h_k(s)$ 的地址位置置成 1。

这样，描述一个元素 $s \in S$ 就可以用它的哈希值 $h_1(s), h_2(s), \dots, h_k(s)$ 在数组上对应位置的元素值是否全为 1 来表示，只要有一个对应的元素值为 0，则表示这个元素 s 不在集合 S 上。由上可见，Bloom filter 的本质是哈希计算，不同之处在于 Bloom filter 对同一数据使用多个哈希函数进行多次哈希，将结果保存在同一个向量数组中，所以 Bloom filter 在达到相同的功能的情况下比原始的哈希结构更节约存储空间。Bloom filter 算法的一个缺点在于查询一个元素是否在集合 S 上可能存在失误定位(False Positive)。失误定位是指可能存在元素 b 且 $b \notin S$ ，但是 Bloom filter 算法判断 $b \in S$ 。

3.2 false positive 概率推导

假设 Hash 函数以等概率条件选择并设置 Bit Array 中的某一位， m 是该位数组的大小， k 是 Hash 函数的个数，那么位数组中某一特定的位在进行元素插入时的 Hash 操作中没有被置位的概率是： $1 - \frac{1}{m}$ 。

那么在 k 次 Hash 操作后，插入 n 个元素后，该位都仍然为 0 的概率是：

$$\left(1 - \frac{1}{m}\right)^{kn}$$

现在检测某一元素是否在该集合中。标明某个元素是否在集合中所需的 k 个位置都按照如上的方法设置为 "1"，但是该方法可能会使算法错误的认为某一原本不在集合中的元素却被检测为在该集合中 (False Positives)，该概率由以下

公式确定：

$$[1 - (1 - \frac{1}{m})^{kn}]^k \approx (1 - e^{-kn/m})^k$$

根据推导可得，Hash 函数个数选取最优数目 $k = (\frac{m}{n}) \ln 2$

此时 false positives 的概率为 $f = (0.6185)^{m/n}$

3.3 Bloom Filter 的多维数据属性表示

多维数据属性表示正如老师上课所举的一个简单的例子，有具有两个属性的桌子，其中一个桌子是小的红色的，另一个桌子是大的蓝色的，现在将这些对应的属性对应的 hash 映射位置一，当再来一张大的红色的桌子的时候，由于对应的属性全为 1，所以会被误判在该集合中是存在的，这就导致了多维数据 false positive 的存在。

标准的 Bloom Filter 不支持多维元素成员的查询，由 Guo 提出的 MDBF 算法核心思想是针对多维数据集的每一个属性，构建独立的布隆过滤器，当查询元素时，通过多维元素的各个属性是否都存在于相应的布隆过滤器中，来判断元素是否属于集合，CMDBF 在 MDBF 的基础上新增了一个用于表示元素整体的联合布隆过滤器 CBF 将各属性的表示和查询作为第一步，第二步联合元素所有的属性域，利用 CBF 完成元素整体的表示和查询。BFM 通过在每个维度上保存一个位向量以构造布隆过滤器，并基于独立属性笛卡尔乘积构造位矩阵，用于全属性查询，从根本上消除了组合误差率。

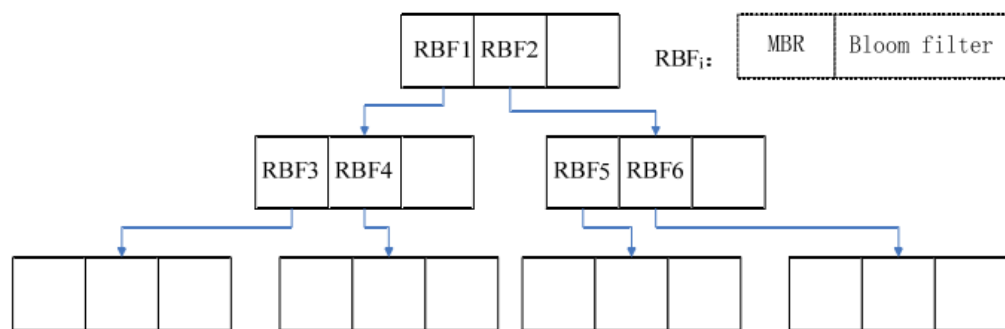
多维元素成员查询典型解决方案包括表索引、树索引及混合索引结构。

目前广泛应用的混合索引结构是将布鲁姆过滤器与树索引进行融合。Adina 等在 2015 年提出了一种混合索引结构 Bloofi，将 B+树和布鲁姆过滤器进行层次化融合，并在此基础上实现位级别的并行化算法 Flat-Bloofi，有效解决了多维数据集的元素查询问题。为了解决云存储环境下的非 KEY 字段查询问题，BF-Matrix 提出了一种层次化索引结构，结合了布鲁姆过滤器和 B+树，极大缩减了元素查询路径，并采用基于规则的索引更新机制，有效降低布鲁姆过滤器的假阴性概率。

四、实验设计

4.1 R 树和 Bloom Filter 相结合的索引结构 RBF

R 树能有效的支持多维范围查询，但不能有效的支持点查询，因为成员查询只能在叶子节点上进行，会导致效率低下，但是 Bloom Filter 是一种空间利用率高且能有效支持点查询的结构，所以扩展经典的 R 树结构，将 Bloom Filter 插入到每个 R 树节点上来支持点查询，维持多维范围并实现空间效率的方法被提出。结构图见下图。



R 树结构的每个分支上都添加了 bloom filter 结构，主要结构定义如下：

```
struct Rect
{
    Rect Real boundary[NUMSIDES]; /* xmin,ymin,...,xmax,ymax,...
    */
};

struct Branch
{
    struct Rect rect;
    struct Node *child;
    bloom_filter *filter;
    int count;
};

struct Node
{
    int count;
    int level; /* 0 is leaf, others positive */
    struct Branch branch[MAXCARD];
};
```

Rect 表示多维数据范围，即结点 MBR。NUMSIDES 为维度

Branch 表示一个节点的多个分支数据结构，每个分支有 MBR 和 Bloom Filter

以及孩子指针。Node 是结点的数据结构，记录层数和分支信息。

4.2 更新缓存结构

由分区索引结构和其他索引结构两部分组成，分区索引结构按照 baseline index 的某一层做 version 划分，每一个 version 是 baseline index 的一颗子树的更新数据索引。利用数据的局部更新特性，如果更新的数据没有超出子树的 MBR 范围，按照删除再插入的更新机制还是被插入到 baseline index 的同一子树下，则不需要这样冗余的操作，可直接采用自底而上更新法。

分区索引结构的主要的数据结构为：

```
struct PVersion
{
    struct Branch *b;
    struct Node *Ver Tree;
    struct PVersion *next;
    struct PVersion *former;
};
struct Parti Version
{
    struct PVersion *head;
    struct PVersion *tail;
    int count;
};
```

4.3 点查询

点查询决定某一特定对象是否是一个数据集的成员。例如，对多维元数据的一个点查询请求可能是确定一个“大小为 10G，文件为可读，文件名为 a.c”的对象是否是当前的存储系统的成员。

点查询的定义如下：对于数据库 DB、查询点 Q，点查询为：Point Query(DB,Q) = {P ∈ DB | P = Q} 在 RBF 索引结构中，支持两类查询：首先是给定多维点数据，查找索引确定是否存在；其次是给定文件名或标识符，通过层次 Bloom filter 确定文件是否存在，以及文件的多维属性信息。其中层次 Bloom filter 也用于快速定位更新数据存放在哪个 version 上以及点查询时快速判断 version 中是否存在某个数据的更新数据。

对于带 version 的点查询，只要找到最新的数据即可，由于 version 链不是按照时间顺序建立的，所以遍历所有 incremental indexes 的 BF，找到相应的点即返回数据；若 version 中都没有找到，则查找原索引 baseline index。如果 version 中存在待查询的数据，则系统的查询延迟比普通的 RBF 索引结构要快得多，这也从一个方面解决了热点数据的更新和查询问题。

上面介绍了 version 系统的点查询流程，对于单个 RBF 树形结构结构，点查询算法如下：

```
Bloom Filter Search()
{
    //输入查询 id, 索引根节点
    if(节点层数>0){
        对该节点的每一个分支,
        if(该分支的BF包含该id)
            对该分支的孩子节点调用 Bloom Filter Search()函数;
    }
    else{
        对叶子节点的每一个分支,
        if(该分支的 BF 包含该 id)
            输出该数据存在的信息;
    }
}
```

五、性能测试

5.1 误判率研究

对于全属性查询对比 MDBF、CMDBF、CBF 索引结构。

随着相关系数增大，误判率趋近于 1。

在相关系数 $a=0.8$ 时，MDBF 的误判率 $>CMDBF > CBF$ ，其中 CBF 最佳，在 10^{-3} 数量级，MDBF 在 10^{-1} 数量级。

5.2 空间开销研究

在相同内存条件下，BFM 在存在较高的空间需求导致其位冲突率较高而增大了误判率，CMDBF 仍然保持较低的误判率。

可得 CMDBF 的空间开销 $<MDBF < BFM$ 。

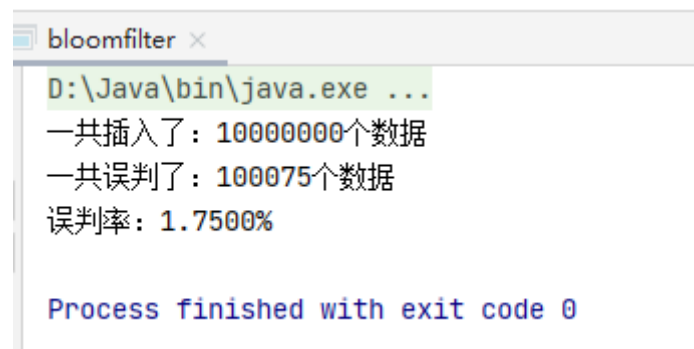
5.3 查询延迟研究

在相关系数 $a=0.8$ 时，MDBF、CBF、CMDBF 都相近，在 0.005ms 数量级。

CMF 查询延迟 $< \text{MDBF} < \text{CMDBF}$ 。

5.4 普通 bloom filter 测试

google 的 guava 提供了 bloom filter 接口，可传入参数进行测试。



```
bloomfilter x
D:\Java\bin\java.exe ...
一共插入了: 10000000个数据
一共误判了: 100075个数据
误判率: 1.7500%

Process finished with exit code 0
```

普通的 bloom filter 主要就是需要实现通过几个 hash 映射创建出布隆过滤器以及完成 hash 映射将对应的二进制位置为一，再实现一个 keymaymatch 的 bool 类型函数判断是否命中（包括 false positive）。

测试程序中计算 false positive 率

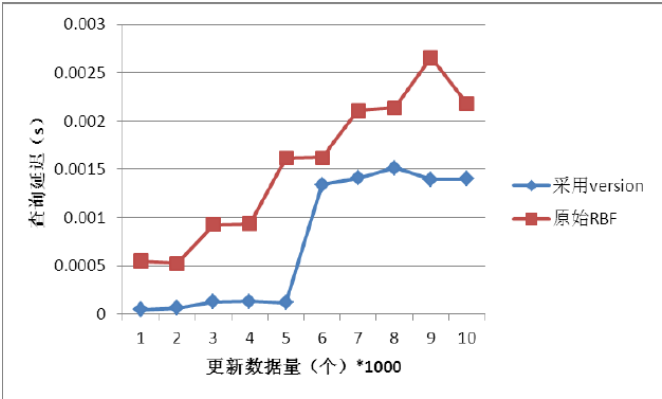
```
bool Matches(const std::string& s) {
    if (!keys_.empty()) {
        Build();
    }
    return policy_->KeyMayMatch(s, filter_);
}

double FalsePositiveRate(const std::vector<std::string>& data) {
    double result = 0;
    for (const auto& item : data) {
        if (Matches(item)) {
            result++;
        }
    }
    return result / 10000.0;
}
```

5.5 RBF 点查询延迟

对数据量进行更新，比较原始 RBF 和采用 version 缓存数据。在 version

中点查询只要找到最新的数据，如果 `version` 中存在待查询的数据，就不需要继续查找原始索引结构，则系统的查询延迟比普通的 `RBF` 索引结构要快得多。



六、实验总结

我在撰写本实验报告中主要是查找相关文献来了解 `Bloom Filter` 对于多维数据属性表示和索引的各种实现方式以及各自实现的效果和性能。其中参考了不少的论文在参考文献都已经注明，让我对于 `Bloom Filter` 有了更进一步的理解，同时也了解到了多种思路，比如通过矩阵笛卡尔积来表示多维情况，通过来结合 `R` 树发挥各自的优势结构。代码实现方面仅仅实现了一个最为简单的 `bloom filter`，采用 `C++` 撰写，通过分析 `k` 的参数后进行实现，并且能调用 `bloom filter` 的接口的地方也很多，`java` 可以调用 `maven` 导入的谷歌的包，也可以通过 `redis` 调用 `bloom filter` 进行操作处理。布隆过滤器已经经历了很长时间的发展和不断优化，提出了各种各样为了方便当下需求的改进模式，此次实验进行了了解仍然只是冰山一角，也为以后能够深入了解并自己构想并实现一种改善结构打下基础，令我受益匪浅。

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] 刘建坤. OBF-Index: 基于 Ordinal Bloom Filter 的分布式多维索引[D]. 云南大学, 2017.
- [9] 高梦颖. 存储系统中多维元数据索引的高效更新方法研究[D]. 华中科技大学, 2011.
- [10] Yong WANG, Xiao-chun YUN, ANGShu-peng WANG, Xi WANG. CBFM: cutted Bloom filter matrix for multi-dimensional membership query[J]. Journal on Communications, 2016, 37(3): 139-147.
- [11] GUO D, WU J, CHEN H, et al. Theory and network applications of dynamic bloom filters[C]//INFOCOM. c2006: 1-12.
- [12] 谢鲲, 秦拯, 文吉刚, 等. 联合多维布鲁姆过滤器查询算法[J]. 通信学报, 2008, 29(1): 56-64.
- XIE K, QIN Z, WEN J G, et al. Combine multi-dimension Bloom filter for membership queries[J]. Journal on Communications, 2008, 29(1): 56-64.
- [13] CHENG X, LI H, WANG Y, et al. BF-matrix: a secondary index for the cloud storage[M]. Springer International Publishing, 2014. 384-396.