



基于 Bloom Filter 的设计

课 程 报 告

姓 名 武传航

学 号 U201915020

班 号 CS2001

日 期 2023.05.29

目 录

一、实验目的.....	2
二、实验背景.....	3
三、实验内容.....	4
3.1 Bloom Filter 原理	4
3.2 Bloom Filter 错误率估计.....	4
3.3 Bloom Filter 的优缺点.....	5
3.4 数据结构设计	5
四、理论分析.....	8
五、实验测试.....	9
5.1 实验说明	9
5.2 实验结果与分析	9
六、课程总结.....	11
参考文献.....	12

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

Bloom filter 是由 Howard Bloom 在 1970 年提出的二进制向量数据结构，它具有空间和时间效率，被用来检测一个元素是不是集合中的一个成员。

如果检测结果为是，该元素不一定在集合中；但如果检测结果为否，该元素一定不在集合中。因此 Bloom filter 具有 100% 的召回率。这样每个检测请求返回有“在集合内（可能错误）”和“不在集合内（绝对不在集合内）”两种情况，可见 Bloom filter 是牺牲了正确率和时间以节省空间。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。

Bloom filter 优点就是它的插入和查询时间都是常数，另外它查询元素却不保存元素本身，具有良好的安全性。它的缺点也是显而易见的，当插入的元素越多，错判“在集合内”的概率就越大了，另外 Bloom filter 也不能删除一个元素，因为多个元素哈希的结果可能在 Bloom filter 结构中占用的是同一个位，如果删除了一个比特位，可能会影响多个元素的检测。

通过本课题的研究，可以优化 Bloom Filter 的性能，以及在其缺点上寻求解决方案并给出一个具体的实验范例。

三、实验内容

3.1 Bloom Filter 原理

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节省。

初始状态时，Bloom Filter 是一个包含 m 位的位数组，每一位都置 0。

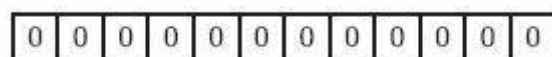


图 3-1 BloomFilter 初始状态

为了表达 $S=\{x_1, x_2, \dots, x_n\}$ 这样一个 n 个元素的集合，Bloom Filter 使用 k 个相互独立的哈希函数（Hash Function），它们分别将集合中的每个元素映射到 $\{1, \dots, m\}$ 的范围中。对任意一个元素 x ，第 i 个哈希函数映射的位置 $h_i(x)$ 就会被置为 1（ $1 \leq i \leq k$ ）。注意，如果一个位置多次被置为 1，那么只有第一次会起作用，后面几次将没有任何效果。在下图中， $k=3$ ，且有两个哈希函数选中同一个位置（从左边数第五位）。



图 3-2 BloomFilter 映射关系

在判断 y 是否属于这个集合时，我们对 y 应用 k 次哈希函数，如果所有 $h_i(y)$ 的位置都是 1（ $1 \leq i \leq k$ ），那么我们就认为 y 是集合中的元素，否则就认为 y 不是集合中的元素。下图中 y_1 就不是集合中的元素。 y_2 或者属于这个集合，或者刚好是一个 false positive

3.2 Bloom Filter 错误率估计

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：

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

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

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

3.3 Bloom Filter 的优缺点

布隆过滤器的优点是相比于其它的数据结构，其在空间和时间方面都有巨大的优势。布隆过滤器存储空间和插入/查询时间都是常数 $O(K)$ ，另外，散列函数相互之间没有关系，方便由硬件并行实现。布隆过滤器不需要存储元素本身，在某些对保密要求非常严格的场合有优势。

布隆过滤器可以表示全集，其它任何数据结构都不能；

但是布隆过滤器的缺点和优点一样明显。误算率是其中之一。随着存入的元素数量增加，误算率随之增加。但是如果元素数量太少，则使用散列表足矣。

另外，一般情况下不能从布隆过滤器中删除元素。我们很容易想到把位数组变成整数数组，每插入一个元素相应的计数器加 1，这样删除元素时将计数器减掉就可以了。然而要保证安全地删除元素并非如此简单。首先我们必须保证删除的元素的确在布隆过滤器里面。这一点单凭这个过滤器是无法保证的。另外计数器回绕也会造成问题。

在降低误算率方面，有不少工作，使得出现了很多布隆过滤器的变种。

3.4 数据结构设计

如下图，题目中给出的多维 Bloom Filter 的结构是在 Bloom Filter 的基础上改进的 Counter Bloom Filter。这种结构中，不再使用二进制数组来管理数据，而是为每一个位置维护了一个 Counter，每次 Hash 函数计算之后，将结果的位

置计数加一。这样的改进可以支持元素的删除，同时又具有 Bloom Filter 查询的高效性能，唯一的缺陷是增加了存储空间，当然这也是必须的，因为二进制数组只能存储一位的数据。

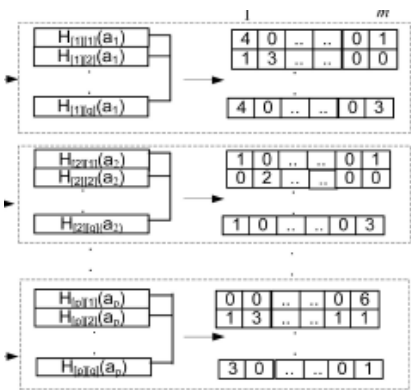


图 3-2 多维 Bloom Filter 结构

其性能上并没有变化，仍然是在 $e^{-\frac{nk}{m}} = \frac{1}{2}$ 时可以取到 f 的最小值。

CBF 和 BF 的一个主要的不同就是 CBF 用一个 Counter 取代了 BF 中的一位，那么 Counter 到底取多大比较合适呢？这里就要考虑到空间利用率的问题了，从使用的角度来看，当然是越大越好，因为 Counter 越大就能表示越多的信息。但是越大的 Counter 就意味着更多的资源占用，而且在很多时候会造成极大的空间浪费。

因此，我们在选择 Counter 的时候，可以看 Counter 取值的范围多小就可以满足需求。

根据论文中描述，某一个 Counter 的值大于或等于 i 的概率可以通过如下公式描述，其中 n 为集合的大小，m 为 Counter 的数量，k 为 哈希函数的个数。

$$\Pr(\max(c) \geq i) \leq m \binom{nk}{i} \frac{1}{m^i} \leq m \left(\frac{enk}{im} \right)^i.$$

$$\Pr(\max(c) \geq i) \leq m \left(\frac{e \ln 2}{i} \right)^i.$$

如果每个 Counter 分配 4 位，那么当 Counter 的值达到 16 时就会溢出。这个概率如下，这个值足够小，因此对于大多数应用程序来说，4 位就足够了。

$$\Pr(\max(c) \geq 16) \leq 1.37 \times 10^{-15} \times m$$

为了简化问题，我们接下来考虑一维单个 Hash 数组。对于更复杂的问题，可以直接迁移考虑。

一次添加请求包括要添加的数据集合 $S' = \{a_1, a_2 \dots a_i\}$ ，当请求到来时，根据 Hash 函数的个数 k 创建 k 个线程，每个线程针对集合中的所有元素进行一次添加操作。这个添加操作相比原本的 Bloom Filter 也有简化，只计算一个 Hash 函数，并更新 Hash 数组在这个 Hash 函数结果处的值。

一次查询请求包括一个数据 a_1 ，当请求到来时，循环执行 k 个 Hash 函数来判断元素是否在集合中。这里不能使用并发操作，因为每一个元素是否在集合中是要根据五个 Hash 函数的结果来综合判断，如果根据 Hash 函数来开启多线程的话，会出现大量的同步问题，为解决这些同步问题所引入的互斥锁的开销非常大，反而会造成性能损失。

一次删除操作包括要删除的元素集合 $S'' = \{b_1, b_2 \dots b_j\}$ ，其中 b_j ($1 \leq i \leq j$) 为集合 S 中的元素，这一保证由其他函数来实现，不作为删除操作的核心步骤。和添加请求一样，根据 Hash 函数的个数 k 创建 k 个线程，每个线程针对集合中的所有元素进行一次删除操作。同样的，这里的删除操作也简化到只计算一个 Hash 函数即可。

四、理论分析

我们考虑这种情况下的 False Positive 的比率 g 。此时，Hash 数组中的任何一位均只能被一个 Hash 函数选中，并且概率为 $\frac{k}{m}$ ，所以这一位不被选中的概率为 $1 - \frac{k}{m}$ ，对 n 个元素而言，有 $p' = (1 - \frac{k}{m})^n$

很显然，有：

$$p' = (1 - \frac{k}{m})^n \geq \left(1 - \frac{1}{m}\right)^{nk} = p$$

对于 Counter Bloom Filter，对集合中的每一个元素，都要循环调用五个 Hash 函数来计算，并对 Hash 数组五个位置都进行操作后，再对下一个元素进行操作。而现在 k 个 Hash 函数可以安全的并行访问 Hash 数组，在大量的插入删除请求的情形下，有着极大的性能提升。具体的加速比因环境的不同而不同，在接下来的实验测试中将会给出两个 Bloom Filter 的速度对比。

另一方面，并发哈希的结构使得 Hash 数组中的元素分布变得均匀，高效利用了 Hash 数组。而 Counter Bloom Filter 的结构必须要精心选择足够强大的 Hash 函数才能做到这一效果。

五、实验测试

5.1 实验说明

实验中，数据集合 n 为收集到的 <https://openphish.com/feed.txt> 网站的 500 个不重复的 url 网址，并将最后 50 个 url 作为测试集合，实验将使用一般的 Counter Bloom Filter 来对这些 url 进行插入删除实验，并计算插入和删除操作所用的时间。

如果在 Bloom Filter 中查询最后 50 个 url，应该全部返回 false。据此我们就可以根据查询这些 url 返回的 true 的个数来计算 False Positive 的比率。

Hash 函数使用了 mmh3 库的 hash 函数

5.2 实验结果与分析

可以看到在不同 size 下 cbf 有不同的 False Positive 的比率

```
size = 1000
Add Time: 184.75000000000002
Search Time 33.759999999999344
Remove Time 177.6899999999998
False Rate: 0.14
```

图 5-1 size=1000

```
size = 2000
Add Time: 175.19000000000005
Search Time 27.19999999999945
Remove Time 201.45999999999913
False Rate: 0.84
```

图 5-2 size=2000

```
size = 3000
Add Time: 177.54999999999993
Search Time 32.340000000000146
Remove Time 180.28999999999962
False Rate: 0.94
```

图 5-3 size=3000

```
size = 4000  
Add Time: 172.31999999999942  
Search Time 24.58000000000043  
Remove Time 169.5699999999994  
False Rate: 1.0
```

图 5-4 size=4000

```
size = 5000  
Add Time: 203.2199999999998  
Search Time 23.92999999999787  
Remove Time 171.35000000000068  
False Rate: 1.0
```

图 5-5 size=5000

六、课程总结

通过本次课程，我对于 Bloomfilter 的结构和性能有了更加深入的了解和认识，它在应对大数据处理时具有良好的性能，在本次实验当中，我在网上查找了很多的资料来了解 BloomFilter。在这个过程中，学习到了很多关于数据存储的知识，更认识到了 BloomFilter 与传统的 hash 表相比具有的优势

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] 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.
- [9] CHENG X, LI H, WANG Y, et al. BF-matrix: a secondary index for the cloud storage[M]. Springer International Publishing, 2014. 384–396.