大数据存储与管理

基于 **Bloom Filter** 的设计

布隆过滤器（Bloom Filter）是 1970 年由布隆提出的。它实际上是一个很长的二进制向量和一系列随机映射函数。布隆过滤器可以用于检索一个元素是否在一个集合中。它的优点是空间效率和查询时间都远远超过一般的算法，缺点是有一定的误识别率和删除困难.

布隆过滤器的主要作用就是 **"以更低的空间效率检索一个元素是否存在其中"**

基本思想

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。

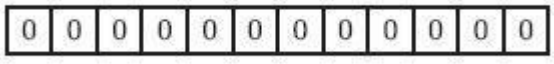
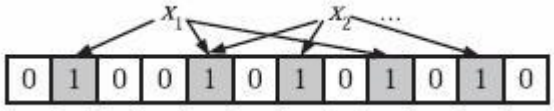


图 3-1 BloomFliter 初始状态

为了表达 S={x1, x2,…,xn}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,…,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 hi(x)就会被置为 1（1≤i≤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 BloomFliter 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 hi(y)的位置都是 1（1≤i≤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 y1 就不是集合中的元素。y2 或者属于这个集合，或者刚好是一个 false positive。

错误率估计

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。

在估计之前为了简化模型，我们假设 kn<m 且各个哈希函数是完全随机的。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：

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

其中 1/m 表示任意一个哈希函数选中这一位的概率（前提是哈希函数是完全随机的），(1-1/m)表示哈希一次没有选中这一位的概率。要把 S 完全映射到位数组中，需要做 kn 次哈希。某一位还是 0 意味着 kn 次哈希都没有选中它，因此这个概率就是（1-1/m）的 kn 次方。令 p = e-kn/m 是为了简化运算，这里用到了计算 e 时常用的近似：

$$\lim_{x\to\infty}\left(1 - \frac{1}{x}\right)^{-x} = \mathrm{e}$$

令 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：

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

优缺点

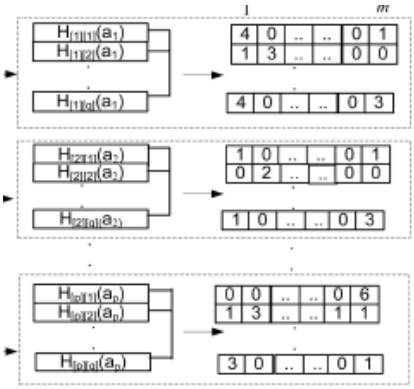
Bloom filter 优点就是它的插入和查询时间都是常数，另外它查询元素却不保存元素本身，具有良好的安全性。它的缺点也是显而易见的，当插入的元素越多，错判“在集合内”的概率就越大了，另外 Bloom filter 也不能删除一个元素，因为多个元素哈希的结果可能在 Bloom filter 结构中占用的是同一个位，如果删除了一个比特位，可能会影响多个元素的检测。

在计算机科学中，我们常常会碰到时间换空间或者空间换时间的情况，即为了达到某一个方面的最优而牺牲另一个方面。Bloom Filter 在时间空间这两个因素之外又引入了另一个因素：错误率。在使用 Bloom Filter 判断一个元素是否属于某个集合时，会有一定的错误率。也就是说，有可能把不属于这个集合的元素误认为属于这个集合（False Positive），但不会把属于这个集合的元素误认为不属于这个集合（False Negative）。在增加了错误率这个因素之后，Bloom Filter 通过允许少量的错误来节省大量的存储空间。

数据结构设计

Counter Bloom Filter 结构

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。



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。

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

$$\Pr(\max(c) \geq i) \leq m\left(\frac{\mathrm{eln}\,2}{i}\right)^{i}$$

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。

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

操作流程

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$S'=\{\ a_1, a_2 \ldots a_i\}$，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$S''=\{b_1, b_2 \ldots b_j\ \}$，其中$b_j$ (1≤i≤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。

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：

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

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。

实验测试

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

结果如下：

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

```
size = 1000
Add Time: 184.7500000000002
Search Time 33.759999999999344
Remove Time 177.6899999999998
False Rate: 0.14
```

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

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

```
size = 4000
Add Time: 172.31999999999942
Search Time 24.580000000000
Remove Time 169.569999999994
False Rate: 1.0
```

```
size = 5000
Add Time: 203.2199999999998
Search Time 23.929999999999787
Remove Time 171.35000000000068
False Rate: 1.0
```

总结

本文首先通过对普通 Bloom Filter 的流程分析，计算了 Bloom Filter 的理论

性能参数。接着针对 Bloom Filter 存在的问题进行改进，提出了一种 Hash 函数分区，并行计算的改进结构。并通过理论分析与编写代码测试两方面，论证了理论性能分析的正确性和改进的优越性。

综上所述，本文中改进的多路并行的 Bloom Filter 尽管降低了一点 False Positive 的比率，但却在性能上获得了极大的提升，总体是优于题目中原先的 Counter Bloom Filter 的。并且本文中的改进模型并没有影响 Bloom Filter 本身具有的功能，插入、删除和查询的执行也都正常，是一种可行的 Bloom Filter 改进设计方案。

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.