2019 级

基于 Bloom Filter 的设计

姓　名 夏棋
学　号 U201914868
班　号 CE1901
日　期 2022.04.14

目　　录

一、实验目的

1. 了解 Bloom Filter 的发展历史。
2. 建立 Bloom Filter 模型，了解其工作原理。
3. 对 Bloom Filter 的性能，开销，错误率进行分析。

二、实验背景

本在大数据与云计算发展的时代，我们经常会碰到这样的问题。我们是否能高效的判断一个用户是否访问过某网站的主页（每天访问量上亿）或者需要统计网站的 pv、uv。最直接的想法是将所有的访问者存起来，然后每次用户访问的时候与之前集合进行比较。不管是将访问信息存在内存（或数据库）都会对服务器造成非常大的压力。而布隆过滤器（BloomFilter）就可以满足当前的使用场景，容忍一定的错误率，高效（计算复杂度、空间复杂度）的实现访问量信息的跟踪、统计。

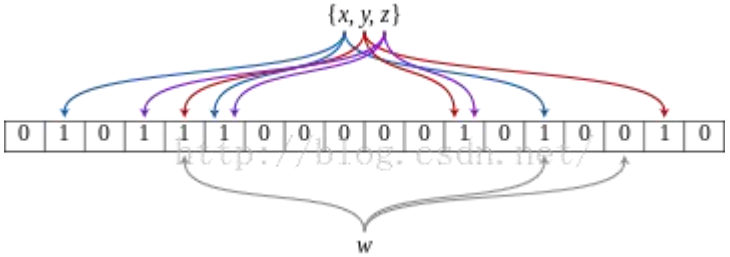
布隆过滤器（BloomFilter）是 1970 年由布隆提出的一种空间空间效率很高的随机数据结构，它利用位数组很简洁地表示一个集合，并判断一个元素是否属于这个集合。使用布隆过滤器，存在第一类出错（Falsepositive），但是不会存在第二类错误（Falsenegative），因此，布隆过滤器拥有 100%的召回率。也就是说，布隆过滤器能够准确判断一个元素不在集合内，但只能判断一个元素可能在集合内。因此，BloomFilter 不适合“零错误”的应用场合。在能够容忍低错误的应用场合下，BloomFilter 通过极少的错误换取了存储空间的极大节省。我们可以向布隆过滤器里添加元素，但是不能从中移除元素（普通布隆过滤器，增强的布隆过滤器是可以移除元素的）。随着布隆过滤器中元素的增加，犯第一类错误的可能性也随之增大。

三、实验内容

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。

算法描述：

一个空的布隆过滤器有长度为 M 比特的 bit 数组构成，且所有位都初始化 0。一个元素通过 K 个不同的 hash 函数随机散列到 bit 数组的 K 个位置上，K 必须远小于 M。K 和 M 的大小由错误率（falsepositiverate）决定。



Bloom Filter 的一个例子集合 S｛x，y，z｝。带有颜色的箭头表示元素经过

k（k＝3）hash 函数的到在 M（bit 数组）中的位置。元素 W 不在 S 集合中，因为元素 W 经过 k 个 hash 函数得到在 M（bit 数组）的 k 个位置中存在值为 0 的位置。

向集合 S 中添加元素 x：x 经过 k 个散列函数后，在 M 中得到 k 个位置，然后，将这 k 个位置的值设置为 1。

判断 x 元素是否在集合 S 中：x 经过 k 个散列函数后，的到 k 个位置的值，如果这 k 个值中间存在为 0 的，说明元素 x 不在集合中——元素 x 曾经插入到过集合 S，则 M 中的 k 个位置会全部置为 1；如果 M 中的 k 个位置全为 1，则有两种情形。情形一：这个元素在这个集合中；情形二：曾经有元素插入的时候将这 k 个位置的值置为 1 了（第一类错误产生的原因 FalsePositive）。简单的布隆过滤器无法区分这两种情况，在增强版中解决了这个问题。

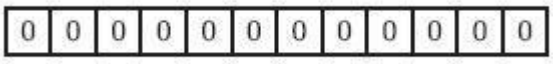
设计 k 个相互独立的 hash 函数可能工作量比较大，但是一个好的 hash 函数是降低误判率的关键。一个良好的 hash 函数应该有宽输出，他们之间的冲突应尽量低，这样 k 个 hash 函数能静可能的将值 hash 的更多的位置。hash 函数的设计是我们可以将 k 个不同的值（ 0, 1, ..., k - 1）作为参数传入，或者将它们加入主键中。对于大的 M 或者 k，hash 函数之间的独立性对误判率影响非常大（(Dillinger & Manolios (2004a),Kirsch & Mitzenmacher (2006)）),Dillinger 在 k 个散列函数中，多次使用同一个函数散列，分析对误判率的影响。

对于简单布隆过滤器来说，从集合 S 中移除元素 x 是不可能的，且 falsenegatives 不允许。元素散列到 k 个位置，尽管可以 将这 k 个位置的值置为 0 来移除这个元素，但是这同事也移除了那些散落后，有值落在这 k 位中的元素。因此，没有一种方法可以判断移除这个元素后是否影响其它已经加入集合中的元素，将 k 个位置置为 0 会引入二类误差（falsenegative）。

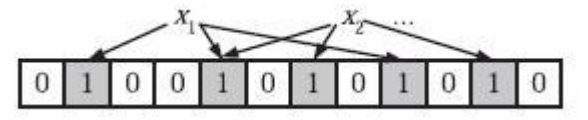
时间复杂度和空间复杂度：

在 falsepositives 的情况下，布隆过滤器相比其它的集合（平衡二叉树、树、hash 表、数组、链表）只需要少量的存储空间。布隆过滤器的添加和检查元素是否在集合内的复杂度为 O(K)。Hash 表的平均复杂度比布隆过滤器更低。Bloom 过滤器在误差最优的情况下，平均每个元素大概是 1.44bit。

下面我们具体来看 Bloom Filter 是如何用位数组表示集合的。初始状态时，Bloom Filter 是一个包含 m 位的位数组，每一位都置为 0。

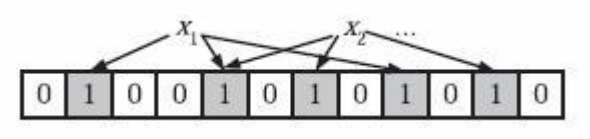


为了表达 S={x1, x2,…,xn}这样一个 n 个元素的集合，Bloom Filter 使用 k 个相互独立的哈希函数（Hash Function），它们分别将集合中的每个元素映射到{1,…,m}的范围中。对任意一个元素 x，第 i 个哈希函数映射的位置 hi(x)就会被置为 1（1≤i≤k）。注意，如果一个位置多次被置为 1，那么只有第一次会起作用，后面几次将没有任何效果。在下图中，k=3，且有两个哈希函数选中同一个位置（从左边数第五位）。



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：

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

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：

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

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：

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

(1-ρ)为位数组中 1 的比例，(1-ρ)k 就表示 k 次哈希都刚好选中 1 的区域，即 false positive rate。上式中第二步近似在前面已经提到了，现在来看第一步近似。p’只是 ρ 的数学期望，在实际中 ρ 的值有可能偏离它的数学期望值。M. Mitzenmacher 已经证明[2]，位数组中 0 的比例非常集中地分布在它的数学期望值的附近。因此，第一步的近似得以成立。分别将 p 和 p’代入上式中，得：

$$f' = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k = (1-p')^k$$

$$f = \left(1 - e^{-kn/m}\right)^k = (1-p)^k.$$

相比 p’和 f’，使用 p 和 f 通常在分析中更为方便。

最优的哈希函数个数

既然 Bloom Filter 要靠多个哈希函数将集合映射到位数组中，那么应该选择几个哈希函数才能使元素查询时的错误率降到最低呢？这里有两个互斥的理由：如果哈希函数的个数多，那么在对一个不属于集合的元素进行查询时得到 0 的概率就大；但另一方面，如果哈希函数的个数少，那么位数组中的 0 就多。为了得到最优的哈希函数个数，我们需要根据上一小节中的错误率公式进行计算。

先用 p 和 f 进行计算。注意到 f = exp(k ln(1 - e-kn/m))，我们令 g = k ln(1 - e-kn/m)，只要让 g 取到最小，f 自然也取到最小。由于 p = e-kn/m，我们可以将 g 写成

$$g = -\frac{m}{n}\ln(p)\ln(1-p),$$

根据对称性法则可以很容易看出当 p = 1/2，也就是 k = ln2 • (m/n)时，g 取得最小值。在这种情况下，最小错误率 f 等于(1/2)k ≈ (0.6185)m/n。另外，注意到 p 是位数组中某一位仍是 0 的概率，所以 p = 1/2 对应着位数组中 0 和 1 各一半。换句话说，要想保持错误率低，最好让位数组有一半还空着。

需要强调的一点是，p = 1/2 时错误率最小这个结果并不依赖于近似值 p 和 f。同样对于 f’ = exp(k ln(1 - (1 - 1/m)kn))，g’ = k ln(1 - (1 - 1/m)kn)，p’ = (1 - 1/m)kn，我们可以将 g’写成

$$g' = \frac{1}{n\ln(1-1/m)}\ln(p')\ln(1-p'),$$

同样根据对称性法则可以得到当 p’ = 1/2 时，g’取得最小值。

位数组的大小：

下面我们来看看，在不超过一定错误率的情况下，Bloom Filter 至少需要多少位才能表示全集中任意 n 个元素的集合。假设全集中共有 u 个元素，允许的最大错误率为ε，下面我们来求位数组的位数 m。

假设 X 为全集中任取 n 个元素的集合，F(X)是表示 X 的位数组。那么对于集合 X 中任意一个元素 x，在 s = F(X)中查询 x 都能得到肯定的结果，即 s 能够接受 x。显然，由于 Bloom Filter 引入了错误，s 能够接受的不仅仅是 X 中的元素，它还能够ε (u - n)个 false positive。因此，对于一个确定的位数组来说，它能够接受总共 n + ε (u - n)个元素。在 n + ε (u - n)个元素中，s 真正表示的只有其中 n 个，所以一个确定的位数组可以表示 $\binom{n+\epsilon(u-n)}{n}$ 个集合。m 位的位数组共有 2m 个不同的组合，进而可以推出，m 位的位数组可以表示 $2^m\binom{n+\epsilon(u-n)}{n}$ 个集合。全集中 n 个元素的集合总共有 $\binom{u}{n}$ 个，因此要让 m 位的位数组能够表示所

有 n 个元素的集合，必须有 $2^m\binom{n+\epsilon(u-n)}{n} \geq \binom{u}{n}$ 即：

$$m \geq \log_2 \frac{\binom{u}{n}}{\binom{n+\epsilon(u-n)}{n}} \approx \log_2 \frac{\binom{u}{n}}{\binom{\epsilon u}{n}} \geq \log_2 \epsilon^{-n} = n \log_2(1/\epsilon).$$

上式中的近似前提是 n 和ϵu 相比很小，这也是实际情况中常常发生的。根据上式，我们得出结论：在错误率不大于ϵ的情况下，m 至少要等于 n log2(1/ϵ)才能表示任意 n 个元素的集合。

上一小节中我们曾算出当 k = ln2 • (m/n)时错误率 f 最小，这时 f = (1/2)k = (1/2)mln2 / n。现在令 f≤ϵ，可以推出

$$m \geq n\frac{\log_2(1/\epsilon)}{\ln 2} = n \log_2 \mathrm{e} \cdot \log_2(1/\epsilon).$$

这个结果比前面我们算得的下界 n log2(1/ϵ)大了 log2 e ≈ 1.44 倍。这说明在哈希函数的个数取到最优时，要让错误率不超过ϵ，m 至少需要取到最小值的 1.44 倍。

四、实验总结

本次大数据存储与管理课程的结课报告，我选择选题一，了解了 boomfilter 的发展历程，同时对其性能进行了简单分析，由于动手能力较差，并未进行实验验证。

通过这次课程的学习，我对于存储有了进一步的了解，同时对于论文的规范性有了更深刻的认识。

参考文献

[1] ZHENG Q, CHEN H, WANG Y 等. COSBench: A Benchmark Tool for Cloud Object Storage Services[C]//2012 IEEE Fifth International Conference on Cloud Computing. 2012: 998－999.

[2] ARNOLD J. OpenStack Swift[M]. O'Reilly Media, 2014.

[3] WEIL S A, BRANDT S A, MILLER E L 等. Ceph: A Scalable, High-performance Distributed File System[C]//Proceedings of the 7th Symposium on Operating Systems Design and Implementation. Berkeley, CA, USA: USENIX Association, 2006: 307－320.

[4] Dean J, Barroso L A. Association for Computing Machinery, 2013. The Tail at Scale[J]. Commun. ACM, 2013, 56(2): 74－80.

[5] Delimitrou C, Kozyrakis C. Association for Computing Machinery, 2018. Amdahl's Law for Tail Latency[J]. Commun. ACM, 2018, 61(8): 65－72.

（可以根据实际需要更新调整）