2020 级

基于 Bloom Filter 的设计

姓　名　张学昊
学　号　U202015579
班　号　物联网 2001
日　期　2023.5.29

目　　录

一、实验目的 1
二、　实验背景 1
三、实验内容 1
四、实验总结 4
参考文献 4

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（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.

三、实验内容

1.数据结构

- bloom filter calculate
- n 是过滤器预期支持的元素个数
- m 是过滤器位数组的大小，即该过滤器总共占用多少 bit 的空间
- c 是每个元素平均占用的空间
- p 是假阳性概率，即 fpp (false positive probability)
- k 哈希函数的个数

计算公式

- $m = -nlog(p)\frac{1}{log(2)^2}$
- $c = \frac{m}{n}$
- $p = (1 - e^{\frac{-kn}{m}})^k = (1 - e^{\frac{-k}{\frac{m}{n}}})^k = (1 - e^{\frac{-k}{c}})^k$
- $k = \frac{m}{n} * ln2 = 0.7\frac{m}{n} = 0.7c$

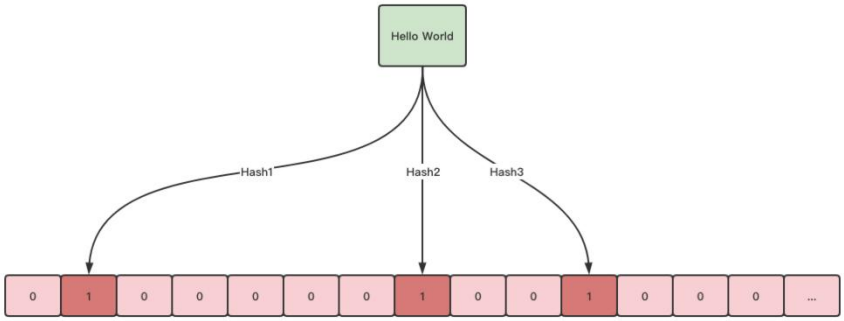
```
n = ceil(m / (-k / log(1 - exp(log(p) / k))))
p = pow(1 - exp(-k / (m / n)), k)
m = ceil((n * log(p)) / log(1 / pow(2, log(2))));
k = round((m / n) * log(2));
```

通常创建一个布隆过滤器，会需要用户提供想支持的 n 和预期的 p，然后通过公式得到其余最佳的参数

结构设计

布隆过滤器的数据结构其实也挺简单，如果之前有了解过位图的话，那么就更简单了。整个布隆过滤器的底层实现就是基于**“位数组 + 哈希函数”**实现的

假设某个布隆过滤器的位数组大小为 1000 bits, 即 1000 个位。由 3 个 hash functions 组成

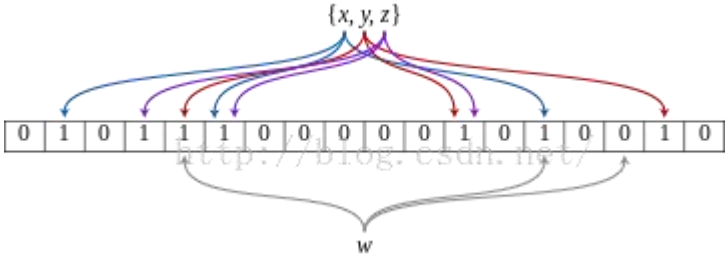


当我们往该布隆过滤器传入一个元素 (Hello World) ，那么该元素就会经过 3 个 hash functions 计算，得到 3 个 hash 值，最后哈希值经过模运算映射到位数组的具体第 x 位上，并标记为 1

当我们要判断某个元素是否在布隆过滤器时，同理，也是将该元素传入，经过哈希函数计算，并得到映射位数组的索引值，判断这个几个索引的位是否为 1， 如果都为 1 则代表该元素已经存在了，但只要有一个位置不为 1，我们就认为该元素并不存在

2.操作流程分析

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）。

3.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。

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）