URI防重原理和性能

策略确定：

URI 防重器解决URI存储的重复问题，防止重复爬取，消耗多余的时间和空间。生成器得到的URI 存储在数据库中，当URI 条数达到几万条或者更多时，通过遍历数据库做匹配的方法不现实。通过数据库索引二叉查找树等方法同样时间复杂度较高。用Hashtable 去重占用空间比较大，且使用率在50%以下。

目前比较高效的方法是利用 布隆过滤器 （Bloom Filter）。可以节约大量空间，且去重查询复杂度为O(1)，时间和空间上都有优势。布隆过滤器有一定误报率（false positive rate），但可以严格防止漏报（false negative）。通过控制bit－map的大小和hash函数的个数，可以将误报率控制在0.01%以下。同时Bloom Filter 允许插入和查询，不允许删除（需要删除时要用改进的Counting Bloom Filter,同时用于bit－map需要原来4倍空间大小，可保证溢出率逼近0）

Bloom Filter基本原理：

1. 一个一维数组（bit-map）和 k 个映射函数（Hashtable），初始每位置 0：

0	0	0	0	0	0	0	0

对于每一个待插入存储的 URI，与 k 个 hash 函数做映射，得到 k 个映射值，将 k 个映射值对应数组坐标置 1，作为此 URI 的标识。

0	1	0	1	1	0	1	0

后续去重只用将每个 URI 进行映射，对比得到的 k 个值坐标位是否都被置 1。若全为 1 则说明已存在。

（URI 映射得到的 k 个位置正好被不同 URI 全置 1 的概率可以控制在 0.01%以下）

误判率和位数组的大小 m 以及映射函数的个数 n 有关。假设映射函数个数为 k，关系如下：

p=2-(m/n)*ln2　　可得　m=(-n*lnp)/(ln2)2=-2*n*lnp=2*n*ln(1/p)

k=(m/n)*ln2=0.7*(m/n)

误判概率及bloom filter 选择

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 操作中没有被置位为1的概率是：$1-\frac{1}{m}$；那么在所有 k 次 Hash 操作后该位都没有被置 "1" 的概率是：$(1-\frac{1}{m})^k$；如果我们插入了 n 个元素，那么某一位仍然为 "0" 的概率是：$(1-\frac{1}{m})^{km}$ 因而该位为 "1"的概率是：$1-(1-\frac{1}{m})^{km}$；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）。

上述结果是在假定由每个 Hash 计算出需要设置的位（bit） 的位置是相互独立为前提计算出来的，不难看出，随着 m （位数组大小）的增加，假正例（False Positives）的概率会下降，同时随着插入元素个数 n 的增加，False Positives的概率又会上升，对于给定的m，n，如何选择Hash函数个数 k 由以下公式确定：$\frac{m}{n}\ln 2 \approx 0.7\frac{m}{n}$；此时False Positives 的概率为：$2^{-k} \approx 0.6185^{m/n}$；而对于给定的False Positives概率 p，如何选择最优的位数组大小 m 呢，$m=-\frac{n\ln p}{(\ln 2)^2}$；该式表明，位数组的大小最好与插入元素的个数成线性关系，对于给定的 m，n，k，假正例概率最大为：$(1-e^{-k(n+0.5)/(m-1)})^k$

布隆过滤器配置

用户只需要决定 add 的元素数 n 和控制的误差率 P，之后的所有参数将由系统计算。

系统首先要计算需要的内存大小 m bits:

$$P=2^{-\ln 2\frac{m}{n}} \Rightarrow \ln p=\ln 2\bullet(-\ln 2)\bullet\frac{m}{n} \Rightarrow m=-\frac{n\cdot\ln p}{(\ln 2)^2}$$

再由 m，n 得到 hash 函数个数 k：

$$k=\ln 2\bullet\frac{m}{n}=0.7\bullet\frac{m}{n}$$

至此系统所需的参数已经备齐，接下来 add n 个元素至布隆过滤器中，再进行 query。

根据公式，当 k 最优时：

$$P(error) = 2^{-k} \Rightarrow \log_2 P = -k \Rightarrow \log_2 \frac{1}{P} \Rightarrow \ln 2 \frac{m}{n} = \log_2 \frac{1}{P} \Rightarrow \frac{m}{n} = \ln 2 \bullet \log_2 \frac{1}{P} = 1.44 \bullet \log_2 \frac{1}{P}$$

因此可验证当 P=1%时，存储每个元素需要 9.6 bits：

$$\frac{m}{n} = 1.44 \bullet \log_2 \frac{1}{0.01} = 9.6 \quad bits$$

而每当想将误判率降低为原来的 1/10，则存储每个元素需要增加 4.8 bits：

$$\frac{m}{n} = 1.44 \bullet (\log_2 10a - \log_2 a) = 1.44 \bullet \log_2 10 = 4.8 \quad bits$$

这里需要特别注意的是，9.6 bits/element 不仅包含了被置为 1 的 k 位，还把包含了没有被置为 1 的一些位数。此时的

$$k = 0.7 \bullet \frac{m}{n} = 0.7 * 9.6 = 6.72 \quad bits$$

才是每个元素对应的为 1 的 bit 位数。

$k = 0.7 \bullet \frac{m}{n}$ ，从而使得 P(error)最小时，我们注意到：

$P(error) = (1 - e^{-\frac{nk}{m}})^k$ 中的 $e^{-\frac{nk}{m}} = \frac{1}{2}$，即

$$(1 - \frac{1}{m})^{kn} = \frac{1}{2}$$

此概率为某 bit 位在插入 n 个元素后未被置位的概率。因此，想保持错误率低，布隆过滤器的空间使用率需为 50%。

性能

使用Bloom Filter实现防重器，时间复杂度O(1)，空间复杂度远小于数据库索引查找，Hashtable等。

用户只需要决定add的元素数n和控制的误差率P，之后的所有参数将由系统计算。系统首先要计算需要的内存大小m bits:

$$P = 2^{-\ln 2\frac{m}{n}} \Rightarrow \ln p = \ln 2 \bullet (-\ln 2) \bullet \frac{m}{n} \Rightarrow m = -\frac{n \cdot \ln p}{(\ln 2)^2}$$

再由 m，n 得到 hash 函数个数 k：

$$k = \ln 2 \bullet \frac{m}{n} = 0.7 \bullet \frac{m}{n}$$

m∝ n

Example:

设置 uri数量为n=1000000，控制差错率p=0.01% ，代入计算

m= 19 170 116.754734 bits = 2.28mb

m正比于n, 即1千万uri需要22.9mb 位图（bit-map），1亿uri需要229mb 位图（bit-map）

哈希函数个数不变，k=13.4=14个