Simhash的生成及存储

发表于 2018-03-06 | 分类于 每日进步一点点， 算法

背景介绍

根据 *Detecting Near-Duplicates for Web Crawling* 论文中的介绍，在互联网中有很多的网页的内容是一样的，但是他们的网页元素却不是完全相同的，每个域名下的网页总会有一些自己的东西，比如广告、导航栏、网站版权之类的东西，但是对于搜索引擎来讲，只有内容部分才是有意义的，而后面的那些虽然不同，但是对搜索结果没有任何影响，所以在判定内容是否重复的时候，应该忽视后面的部分，当新爬取的内容和数据库中的某个网页的内容一样的时候，就称其为Near-Duplicates（重复文章）。对于重复文章，不应在执行入库操作，这种操作的优点是(A)节省带宽、(B)节省磁盘、(C)减轻服务器负荷以及(D)去除相似文章噪点干扰，提升索引的质量。

在现实中，一模一样的网页的概率是很小的，大部分的相似网页都会存在一些细节的变化，而如何进行这种判定就是一个本文要解决的一个问题。除了近似文章判定算法的难题，还有以下待解决的难点（按照80亿篇文章来考虑）：

- 数据规模巨大，对于海量数据如何存储
- 查找速度，如何做到在毫秒级别返回检索结果

simhash介绍

simhash是由 Charikar 在2002年提出来的,它是一种能计算文档相似度的hash算法，google用它来出来海量的文本去重工作。simhash属于局部敏感型（locality sensitive hash）的一种，其主要思想是降维，将高维的特征向量转化成一个f位的指纹（fingerprint），通过两个得出指纹的海明距离（hamming distince）来确定两篇文章的相似度，海明距离越小，相似度越低，根据 *Detecting Near-Duplicates for Web Crawling* 论文中所说，一般海明距离为3就代表两篇文章相同。
simhash也有其局限性，在处理小于500字的短文本时simhash的表现并不是很好，所以在使用simhash前一定要注意这个细节。

simhash与hash算法的区别

传统的Hash算法只负责将原始内容尽量均匀随机地映射为一个签名值，原理上仅相当于伪随机数产生算法。传统的hash算法产生的两个签名，如果原始内容在一定概率下是相等的；如果不相等，除了说明原始内容不相等外，不再提供任何信息，因为即使原始内容只相差一个字节，所产生的签名也很可能差别很大。所以传统的Hash是无法在签名的维度上来衡量原内容的相似度，而SimHash本身属于一种局部敏感哈希算法，它产生的hash签名在一定程度上可以表征原内容的相似度。

我们主要解决的是文本相似度计算，要比较的是两个文章是否相识，当然我们降维生成了hash签名也是用于这个目的。看到这里估计大家就明白了，我们使用的simhash就算把文章中的字符串变成 01 串也还是可以用于计算相似度的，而传统的hash却不行。我们可以来做个测试，两个相差只有一个字符的文本串，“你妈妈喊你回家吃饭哦，回家罗回家罗”和“你妈妈叫你回家吃饭啦，回家罗回家罗”。

通过simhash计算结果为：

1000010010101101111111100000101011010001001111100001001011001011

1000010010101101011111100000101011010001001111100001101010001011

通过传统hash计算为：

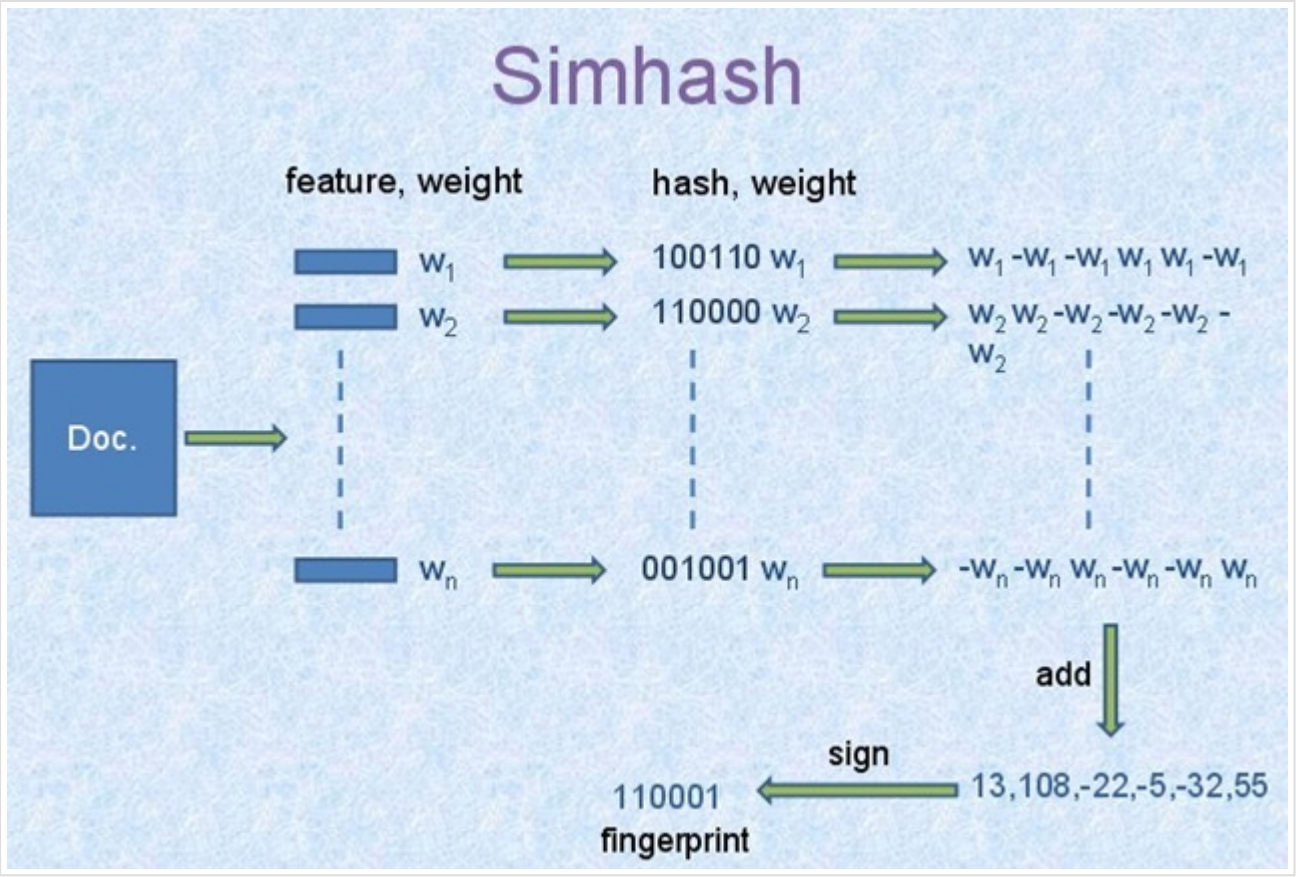
00010000011001101001110110111110

10100100011111111100101100111101

大家可以看得出来，相似的文本只有部分 01 串变化了，而普通的hash却不能做到，这个就是局部敏感哈希的魅力。

simhash的生成

simhash的生成图解如下图：



为了更加通俗易懂，采用例子来详解simhash的生成规则。simhash的生成划分为五个步骤：**分词->hash->加权->合并->降维**

- 1：分词，把需要判断文本分词形成这个文章的特征单词。最后形成去掉噪音词的单词序列并为每个词加上权重，我们假设权重分为5个级别（1~5）。比如：“美国“51区”雇员称内部有9架飞碟，曾看见灰色外星人” ==> 分词后为“美国（4） 51区（5） 雇员（3） 称（1） 内部（2） 有（1） 9架（3） 飞碟（5） 曾（1） 看见（3） 灰色（4） 外星人（5）”，括号里是代表单词在整个句子里重要程度，数字越大越重要。

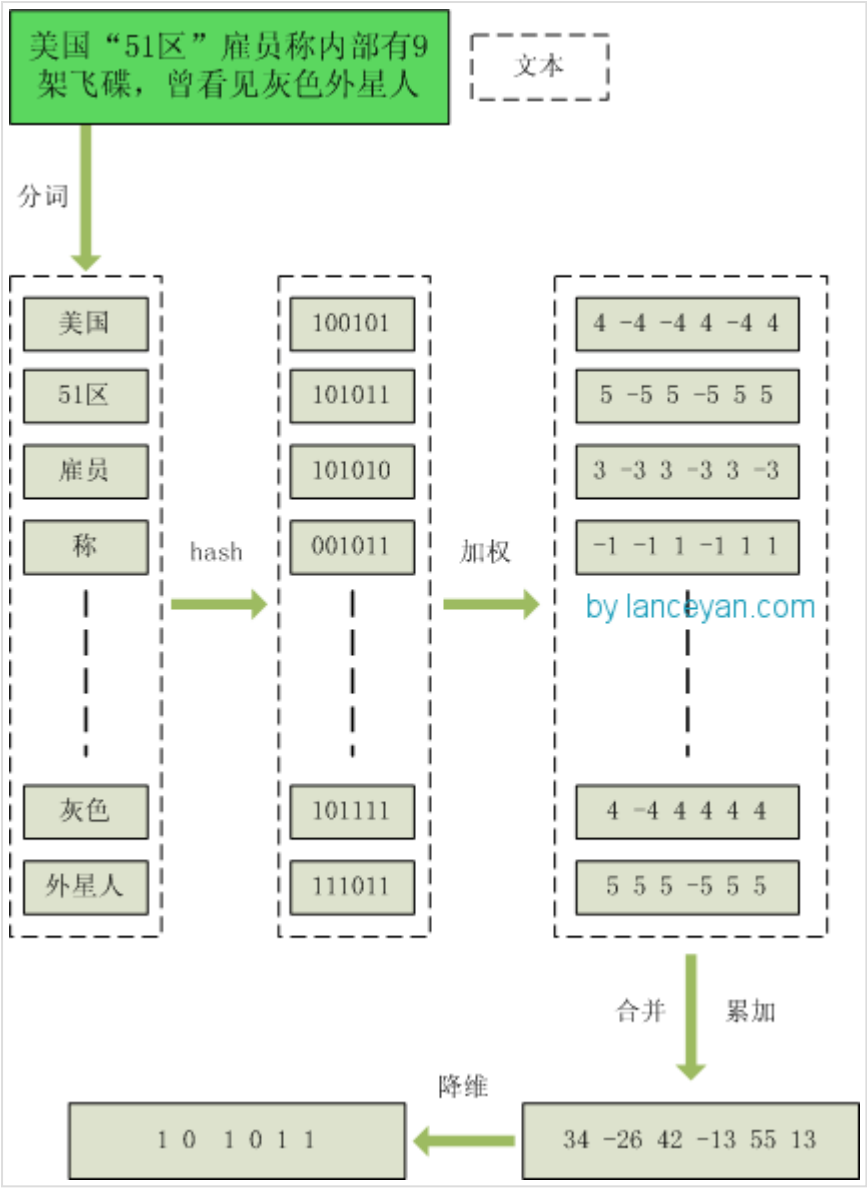
- 2：hash，通过hash算法把每个词变成hash值，比如“美国”通过hash算法计算为 100101,“51区”通过hash算法计算为 101011。这样我们的字符串就变成了一串串数字，还记得文章开头说过的吗，要把文章变为数字计算才能提高相似度计算性能，现在是降维过程进行时。

- 3：加权，通过 2步骤的hash生成结果，需要按照单词的权重形成加权数字串，比如“美国”的hash值为“100101”，通过加权计算为“4 -4 -4 4 -4 4”；“51区”的hash值为“101011”，通过加权计算为 “ 5 -5 5 -5 5 5”。

- 4：合并，把上面各个单词算出来的序列值累加，变成只有一个序列串。比如 “美国”的 “4 -4 -4 4 -4 4”，“51区”的 “ 5 -5 5 -5 5 5”， 把每一位进行累加， “4+5 -4+-5 -4+5 4+-5 -4+5 4+5” ==》 “9 -9 1 -1 1 9”。这里作为示例只算了两个单词的，真实计算需要把所有单词的序列串累加。

- 5：降维，把4步算出来的 “9 -9 1 -1 1 9” 变成 0 1 串，形成我们最终的simhash签名。 如果每一位大于0 记为1，小于0 记为 0。最后算出结果为：“1 0 1 0 1 1”。

整个过程的流程图为：



simhash分表存储策略

在线上查询算法中，首先建立多个指纹表:T1，T2，√...，Tt。每个指纹表 Ti 关联两个未知数：一个整型pi和一个在f bit-positions上的排列πi,Ti就是对已经存在的所有指纹进行排列πi得到的有序集合。对于一个指纹f和一个整数k，算法分两个步骤：

- 1 找到Ti中所有的前pi个bit-positions和πi（F）的前pi个bit-positions相同的指纹，假设为指纹集合F
- 2 在F中的每一个指纹，比较其是否和πi（F）有的差异不超过k个

分表存储原理

借鉴hashmap算法找出可以hash的key值，因为我们使用的simhash是局部敏感哈希，这个算法的特点是只要相似的字符串只有个别的位数是有差别变化。那这样我们可以推断两个相似的文本，至少有16位的simhash是一样的。

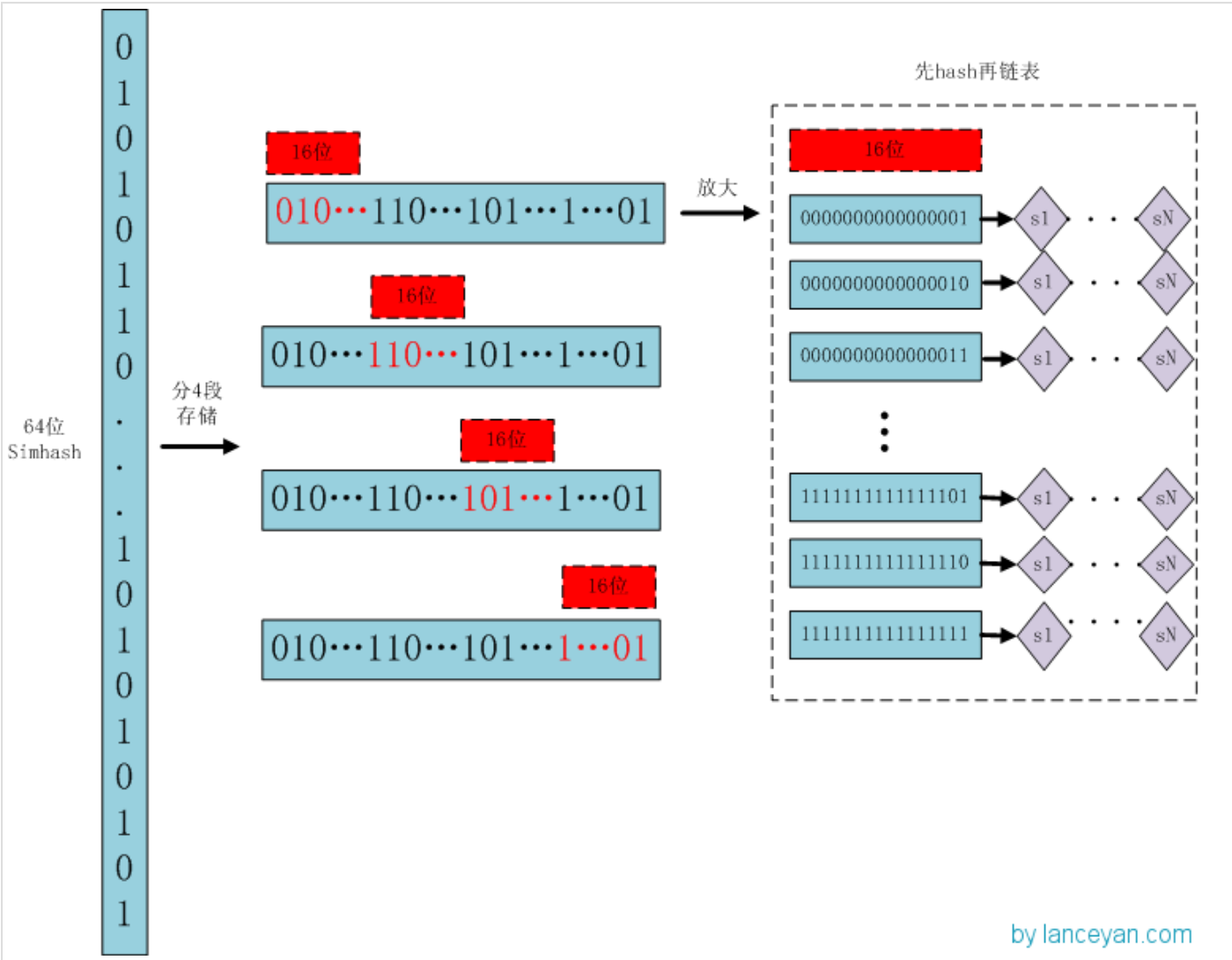
分表存储设计

假设f = 64，k=3，并且我们有80亿 = 2^34个数的网页指纹，d=34，可以有下面四种设计方法 （f：指纹位数，k：海明距离，d：将文章数量转化成2的幂次方，d就是幂值）

1.20个表：将64bit分为11,11,11,11,10,10六个bit块.根据排列组合，如果想从这6个块中找3个作为leading bits的话（这样才能满足|pi-d|是个小整数），一共有C(6,3)=20种找法，所以需要20个表，每个表的前三块来自不同的三个块，那么pi就有11+11+11、11+ 11+10和11+10+10三种可能了。
一次嗅探平均需要检索2^(34-31)=8个指纹
2.16个表：先将64bit均分成4份，然后对每份，将剩馀了48bit，再均分成四份，也就是16,12,12,12,12,12五部分，很明显这种组合的可能是4*4，而pi = 28。一次嗅探平均需要检索2^(34-28)=64个指纹
3.10个表：将65bit分成 13，13，13，13，12 五个bit快。根据排列组合，需要从5块中找到2个作为leading bits，共有C(5,2)=10种找法，需要10张表，而pi=25或26。一次嗅探平均需要检索2^(34-25)=512个指纹

4.4个表：同理 64 等分为4份，每份16bit，从四份中找出1个leading bits，共有C(4,1)=10种找法，pi=16,一次嗅探平均需要检索2^(34-16)=256K个指纹

分表存储实现



存储：

1、将一个64位的simhash签名拆分成4个16位的二进制码。（图上红色的16位）

2、分别拿着4个16位二进制码查找当前对应位置上是否有元素。（放大后的16位）

3、对应位置没有元素，直接追加到链表上；对应位置有则直接追加到链表尾端。（图上的 S1 — SN）

查找：

1、将需要比较的simhash签名拆分成4个16位的二进制码。

2、分别拿着4个16位二进制码每一个去查找simhash集合对应位置上是否有元素。

3、如果有元素，则把链表拿出来顺序查找比较，直到simhash小于一定大小的值，整个过程完成。

原理：

借鉴hashmap算法找出可以hash的key值，因为我们使用的simhash是局部敏感哈希，这个算法的特点是只要相似的字符串只有个别的位数是有差别变化。那这样我们可以推断两个相似的文本，至少有16位的simhash是一样的。具体选择16位、8位、4位，大家根据自己的数据测试选择，虽然比较的位数越小越精准，但是空间会变大。分为4个16位段的存储空间是单独simhash存储空间的4倍。之前算出5000w数据是 382 Mb，扩大4倍1.5G左右，还可以接受

最佳分表策略

根据 4.2节分表存储设计，给定 f,k 我们可以有很多种分表的方式，增加表的个数会减少检索时间，但是会增加内存的消耗，相反的，减少表的个数，会减少内存的压力，但是会增加检索时间。

根据google大量的实验，存在一个分表策略满足时间和空间的平衡点

$$X(f,k,d) = \begin{cases} 1 & \text{if } d < \tau \\ \min_{r>k} \binom{r}{k} \cdot X(\frac{fk}{r}, k, d - \frac{(r-k)f}{r}) & \text{otherwise} \end{cases}$$

τ=d-pi （**pi计算看4.2章节，取最小pi**）

simhash存储实现(Go)

国外有一大神用go实现了d=3和6的实现，在他的基础上我实现了d到8的扩展，源码请看 https://github.com/kricen/shstorage

参考文章

> 论文 Detecting Near-Duplicates for Web Crawling
> http://www.cnblogs.com/maybe2030/p/5203186.html

Simhash

Restful架构的理解及使用场景分析

Powered by

撰写评论

发布

还没有评论，快来抢沙发吧！

文章目录 站点概览