**Simhash**

**关键词：比较文本的相似度，用于查重，步骤：文本分词，对词语进行hash，加权，合并，降维。**

**64位存储，汉明距离为3，将64位分成4部分后以4个16位进行倒排索引**

通过 [采集系统](http://www.lanceyan.com/tech/arch/tensofmillioncrawler.html" \o "实战低成本服务器搭建千万级数据采集系统) 我们采集了大量文本数据，但是文本中有很多重复数据影响我们对于结果的分析。分析前我们需要对这些数据去除重复，如何选择和设计文本的去重算法？常见的有余弦夹角算法、欧式距离、Jaccard相似度、最长公共子串、编辑距离等。这些算法对于待比较的文本数据不多时还比较好用，如果我们的爬虫每天采集的数据以千万计算，我们如何对于这些海量千万级的数据进行高效的合并去重。最简单的做法是拿着待比较的文本和数据库中所有的文本比较一遍如果是重复的数据就标示为重复。看起来很简单，我们来做个测试，就拿最简单的两个数据使用Apache提供的 Levenshtein for 循环100w次计算这两个数据的相似度。代码结果如下：

[String](http://www.google.com/search?hl=en&q=allinurl:string+java.sun.com&btnI=I'm Feeling Lucky) s1 = "你妈妈喊你回家吃饭哦，回家罗回家罗" ;  
            [String](http://www.google.com/search?hl=en&q=allinurl:string+java.sun.com&btnI=I'm Feeling Lucky) s2 = "你妈妈叫你回家吃饭啦，回家罗回家罗" ;  
  
            long t1 = [System](http://www.google.com/search?hl=en&q=allinurl:system+java.sun.com&btnI=I'm Feeling Lucky).currentTimeMillis();  
  
            for (int i = 0; i < 1000000; i++) {  
                   int dis = StringUtils .getLevenshteinDistance(s1, s2);  
            }  
  
            long t2 = [System](http://www.google.com/search?hl=en&q=allinurl:system+java.sun.com&btnI=I'm Feeling Lucky).currentTimeMillis();  
  
            [System](http://www.google.com/search?hl=en&q=allinurl:system+java.sun.com&btnI=I'm Feeling Lucky). out .println(" 耗费时间： " + (t2 - t1) + "  ms ");

耗费时间： 4266 ms

大跌眼镜，居然计算耗费4秒。假设我们一天需要比较100w次，光是比较100w次的数据是否重复就需要4s，就算4s一个文档，单线程一分钟才处理15个文档，一个小时才900个，一天也才21600个文档，这个数字和一天100w相差甚远，需要多少机器和资源才能解决。

为此我们需要一种应对于海量数据场景的去重方案，经过研究发现有种叫 local sensitive hash 局部敏感哈希 的东西，据说这玩意可以把文档降维到hash数字，数字两两计算运算量要小很多。查找很多文档后看到google对于网页去重使用的是simhash，他们每天需要处理的文档在亿级别，大大超过了我们现在文档的水平。既然老大哥也有类似的应用，我们也赶紧尝试下。simhash是由 Charikar 在2002年提出来的，参考 [《Similarity estimation techniques from rounding algorithms》](http://dl.acm.org/citation.cfm?id=509965" \o "Similarity estimation techniques from rounding algorithms" \t "http://www.lanceyan.com/tech/arch/_blank) 。 介绍下这个算法主要原理，为了便于理解尽量不使用数学公式，分为这几步：

****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。

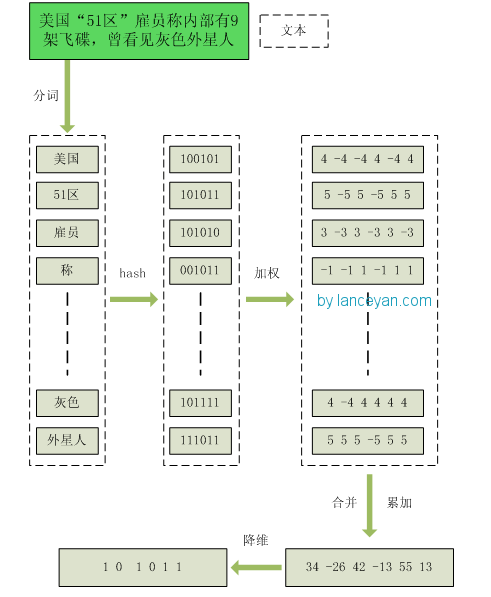
****2、hash****，通过hash算法把每个词变成hash值，比如“美国”通过hash算法计算为 100101,“51区”通过hash算法计算为 101011。这样我们的字符串就变成了一串串数字，还记得文章开头说过的吗，要把文章变为数字计算才能提高相似度计算性能，现在是降维过程进行时。

****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”。

整个过程图为：



大家可能会有疑问，经过这么多步骤搞这么麻烦，不就是为了得到个 0 1 字符串吗？我直接把这个文本作为字符串输入，用hash函数生成 0 1 值更简单。其实不是这样的，传统hash函数解决的是生成唯一值，比如 md5、hashmap等。md5是用于生成唯一签名串，只要稍微多加一个字符md5的两个数字看起来相差甚远；hashmap也是用于键值对查找，便于快速插入和查找的数据结构。不过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hashcode

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hashcode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：

1000010010101101****1****11111100000101011010001001111100001****0****0101****1****001011

1000010010101101****0****11111100000101011010001001111100001****1****0101****0****001011

通过 hashcode计算为：

1111111111111111111111111111111110001000001100110100111011011110

1010010001111111110010110011101

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hashcode却不能做到，这个就是局部敏感哈希的魅力。目前Broder提出的shingling算法和Charikar的simhash算法应该算是业界公认比较好的算法。在simhash的发明人Charikar的论文中并没有给出具体的simhash算法和证明，[“量子图灵”得出的证明simhash是由随机超平面hash算法演变而来的](http://www.cnblogs.com/linecong/archive/2010/08/28/simhash.html" \o "simhash是由随机超平面hash算法演变而来的" \t "http://www.lanceyan.com/tech/arch/_blank)。

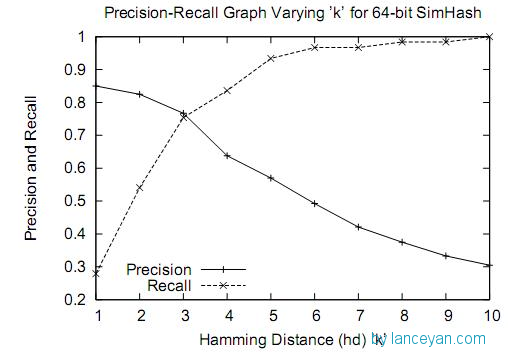
现在通过这样的转换，我们把库里的文本都转换为simhash 代码，并转换为long类型存储，空间大大减少。现在我们虽然解决了空间，但是如何计算两个simhash的相似度呢？难道是比较两个simhash的01有多少个不同吗？对的，其实也就是这样，我们通过海明距离（Hamming distance）就可以计算出两个simhash到底相似不相似。两个simhash对应二进制（01串）取值不同的数量称为这两个simhash的海明距离。举例如下： ****1****01****01**** 和 ****0****01****10**** 从第一位开始依次有第一位、第四、第五位不同，则海明距离为3。对于二进制字符串的a和b，海明距离为等于在a XOR b运算结果中1的个数（普遍算法）。

为了高效比较，我们预先加载了库里存在文本并转换为simhash code 存储在内存空间。来一条文本先转换为 simhash code，然后和内存里的simhash code 进行比较，测试100w次计算在100ms。速度大大提升。

未完待续：

1、目前速度提升了但是数据是不断增量的，如果未来数据发展到一个小时100w，按现在一次100ms，一个线程处理一秒钟 10次，一分钟 60 \* 10 次，一个小时 60\*10 \*60 次 = 36000次，一天 60\*10\*60\*24 = 864000次。 我们目标是一天100w次，通过增加两个线程就可以完成。但是如果要一个小时100w次呢？则需要增加30个线程和相应的硬件资源保证速度能够达到，这样成本也上去了。能否有更好的办法，提高我们比较的效率？

2、通过大量测试，simhash用于比较大文本，比如500字以上效果都还蛮好，距离小于3的基本都是相似，误判率也比较低。但是如果我们处理的是微博信息，最多也就140个字，使用simhash的效果并不那么理想。看如下图，在距离为3时是一个比较折中的点，在距离为10时效果已经很差了，不过我们测试短文本很多看起来相似的距离确实为10。如果使用距离为3，短文本大量重复信息不会被过滤，如果使用距离为10，长文本的错误率也非常高，如何解决？



介绍了simhash的原理，大家应该感觉到了算法的魅力。但是随着业务的增长 simhash的数据也会暴增，如果一天100w，10天就1000w了。我们如果插入一条数据就要去比较1000w次的simhash，计算量还是蛮大，普通PC 比较1000w次海明距离需要 300ms ，和5000w数据比较需要1.8 s。看起来相似度计算不是很慢，还在秒级别。给大家算一笔账就知道了：

随着业务增长需要一个小时处理100w次，一个小时为3600 \*1000 = 360w毫秒，计算一下一次相似度比较最多只能消耗 360w / 100w = 3.6毫秒。300ms慢吗，慢！1.8S慢吗，太慢了！很多情况大家想的就是升级、增加机器，但有些时候光是增加机器已经解决不了问题了，就算增加机器也不是短时间能够解决的，需要考虑分布式、客户预算、问题解决的容忍时间？头大时候要相信人类的智慧是无穷的，泡杯茶，听下轻音乐：）畅想下宇宙有多大，宇宙外面还有什么东西，程序员有什么问题能够难倒呢？

加上客户还提出的几个，汇总一下技术问题：

1、一个小时需要比较100w次，也就是每条数据和simhash库里的数据比较需要做到3.6毫秒。

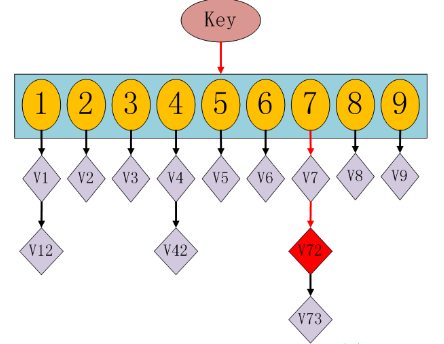
2、两条同一时刻发出的文本如果重复也只能保留一条。

3、希望保留2天的数据进行比较去重，按照目前的量级和未来的增长，2天大概在2000w — 5000w 中间。

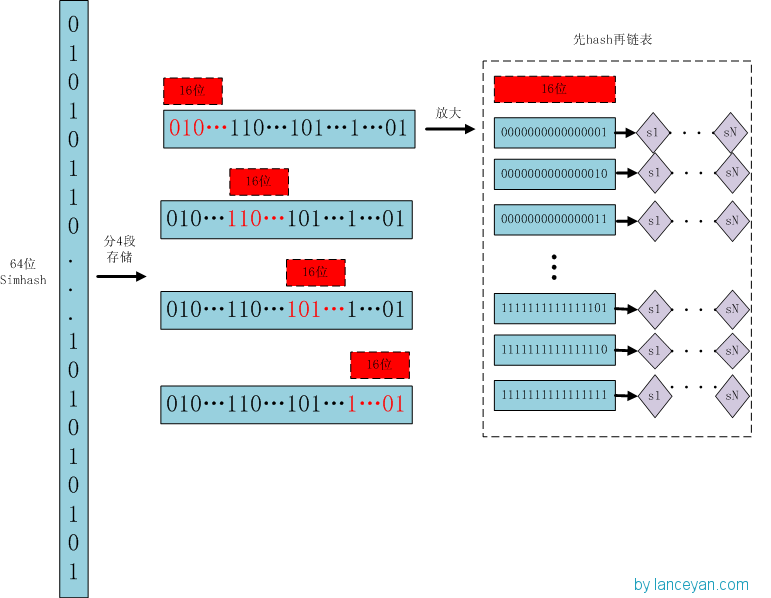
4、短文本和长文本都要去重，经过测试长文本使用simhash效果很好，短文本使用simhash 准备度不高。

目前我们估算一下存储空间的大小，就以JAVA 来说，存储一个simhash 需要一个原生态 lang 类型是64位 = 8 byte，如果是 Object 对象还需要额外的 8 byte，所以我们尽量节约空间使用原生态的lang类型。假设增长到最大的5000w数据， 5000w \* 8byte = 400000000byte = 400000000/( 1024 \* 1024) = 382 Mb，所以按照这个大小普通PC服务器就可以支持，这样第三个问题就解决了。

比较5000w次怎么减少时间呢？其实这也是一个查找的过程，我们想想以前学过的查找算法： 顺序查找、二分查找、二叉排序树查找、索引查找、哈希查找。不过我们这个不是比较数字是否相同，而是比较海明距离，以前的算法并不怎么通用，不过解决问题的过程都是通用的。还是和以前一样，不使用数学公式，使用程序猿大家都理解的方式。还记得JAVA里有个HashMap吗？我们要查找一个key值时，通过传入一个key就可以很快的返回一个value，这个号称查找速度最快的数据结构是如何实现的呢？看下hashmap的内部结构：



如果我们需要得到key对应的value，需要经过这些计算，传入key，计算key的hashcode，得到7的位置；发现7位置对应的value还有好几个，就通过链表查找，直到找到v72。其实通过这么分析，如果我们的hashcode设置的不够好，hashmap的效率也不见得高。借鉴这个算法，来设计我们的simhash查找。通过顺序查找肯定是不行的，能否像hashmap一样先通过键值对的方式减少顺序比较的次数。看下图：

[](http://www.lanceyan.com/wp-content/uploads/2013/09/simhash3.png)

****存储****：  
1、将一个64位的simhash code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），倒排索引  
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 code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。  
2、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，，因为64位分成了4块，每块16位，根据抽屉原理，如果汉明距离的阈值为3，那么相似度高的simhash值必然有一块是相同的，即至少有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：）

通过这样计算，我们的simhash查找过程全部降到了1毫秒以下。就加了一个hash效果这么厉害？我们可以算一下，原来是5000w次顺序比较，现在是少了2的16次方比较，前面16位变成了hash查找。后面的顺序比较的个数是多少？ 2^16 = 65536， 5000w/65536 = 763 次。。。。实际最后链表比较的数据也才 763次！所以效率大大提高！

到目前第一点降到3.6毫秒、支持5000w数据相似度比较做完了。还有第二点同一时刻发出的文本如果重复也只能保留一条和短文本相识度比较怎么解决。其实上面的问题解决了，这两个就不是什么问题了。

* 之前的评估一直都是按照线性计算来估计的，就算有多线程提交相似度计算比较，我们提供相似度计算服务器也需要线性计算。比如同时客户端发送过来两条需要比较相似度的请求，在服务器这边都进行了一个排队处理，一个接着一个，第一个处理完了在处理第二个，等到第一个处理完了也就加入了simhash库。所以只要服务端加了队列，就不存在同时请求不能判断的情况。
* simhash如何处理短文本？换一种思路，simhash可以作为局部敏感哈希第一次计算缩小整个比较的范围，等到我们只有比较700多次比较时，就算使用我们之前精准度高计算很慢的编辑距离也可以搞定。当然如果觉得慢了，也可以使用余弦夹角等效率稍微高点的相似度算法。

代码：

# !/usr/bin/python  
# coding=utf-8  
**class simhash**:  
 # 构造函数  
 **def** \_\_init\_\_(self, tokens='', hashbits=128):  
 self.hashbits = hashbits  
 self.hash = self.simhash(tokens);  
  
 # toString函数  
 **def** \_\_str\_\_(self):  
 **return** str(self.hash)  
  
 # 生成simhash值  
 **def simhash**(self, tokens):  
 v = [0] \* self.hashbits  
 **for** t **in** [self.\_string\_hash(x) **for** x **in** tokens]: # t为token的普通hash值  
 **for** i **in** range(self.hashbits):  
 bitmask = 1 << i  
 **if** t & bitmask:  
 v[i] += 1 # 查看当前bit位是否为1,是的话将该位+1  
 **else**:  
 v[i] -= 1 # 否则的话,该位-1  
 fingerprint = 0  
 **for** i **in** range(self.hashbits):  
 **if** v[i] >= 0:  
 fingerprint += 1 << i  
 **return** fingerprint # 整个文档的fingerprint为最终各个位>=0的和  
  
 # 求海明距离  
 **def hamming\_distance**(self, other):  
 x = (self.hash ^ other.hash) & ((1 << self.hashbits) - 1)  
 tot = 0;  
 **while** x:  
 tot += 1  
 x &= x - 1  
 **return** tot  
  
 # 求相似度  
 **def similarity**(self, other):  
 a = float(self.hash)  
 b = float(other.hash)  
 **if** a > b:  
 **return** b / a  
 **else**:  
 **return** a / b  
  
 # 针对source生成hash值 (一个可变长度版本的Python的内置散列)  
 **def \_string\_hash**(self, source):  
 **if** source == "":  
 **return** 0  
 **else**:  
 x = ord(source[0]) << 7  
 m = 1000003  
 mask = 2 \*\* self.hashbits - 1  
 **for** c **in** source:  
 x = ((x \* m) ^ ord(c)) & mask  
 x ^= len(source)  
 **if** x == -1:  
 x = -2  
 **return** x  
  
  
**if** \_\_name\_\_ == '\_\_main\_\_':  
 s = 'This is a test string for testing'  
 hash1 = simhash(s.split())  
  
 s = 'This is a test string for testing also'  
 hash2 = simhash(s.split())  
  
 s = 'hello world’  
 hash3 = simhash(s.split())  
  
 **print**(hash1.hamming\_distance(hash2), " ", hash1.similarity(hash2))  
 **print**(hash1.hamming\_distance(hash3), " ", hash1.similarity(hash3))