# 背景

假如需要过滤某些不安全网页，不安全网页的黑名单包含100亿个页面，每个网页的URL最多占用64字节。现在想要实现一种网页过滤系统，可以根据网页的URL判断该网页是否在黑名单上，请设计该系统，要求该系统允许有万分之一以下的判断错误率，并且使用的额外空间不要超过30G。



对于涉及网页黑名单系统，垃圾邮件过滤系统，爬虫的网址判断重复系统，同时可以容忍一定程度的失误率，对空间要求较严格的情况，可以使用布隆过滤器。

为了说明Bloom Filter存在的重要意义，举一个实例：假设要你写一个网络蜘蛛（web crawler）。由于网络间的链接错综复杂，蜘蛛在网络间爬行很可能会形成“环”。为了避免形成“环”，就需要知道蜘蛛已经访问过那些URL。给一个URL，怎样知道蜘蛛是否已经访问过呢？稍微想想，就会有如下几种方案：

1、将访问过的URL保存到数据库。

2、用HashSet将访问过的URL保存起来。那只需接近O(1)的代价就可以查到一个URL是否被访问过了。

3、URL经过MD5或SHA-1等单向哈希后再保存到HashSet或数据库。

4、BitMap方法。建立一个BitSet，将每个URL经过一个哈希函数映射到某一位。

方法1~3都是将访问过的URL完整保存，方法4则只标记URL的一个映射位。以上方法在数据量较小的情况下都能完美解决问题，但是当数据量变得非常庞大时问题就来了。

方法1的缺点：数据量变得非常庞大后关系型数据库查询的效率会变得很低。而且每来一个URL就启动一次数据库查询是不是太小题大做了？

方法2的缺点：太消耗内存。随着URL的增多，占用的内存会越来越多。就算只有1亿个URL，每个URL只算50个字符，就需要5GB内存。

方法3：由于字符串经过MD5处理后的信息摘要长度只有128Bit，SHA-1处理后也只有160Bit，因此方法3比方法2节省了好几倍的内存。

方法4：消耗内存是相对较少的，但缺点是单一哈希函数发生冲突的概率太高。还记得数据结构课上学过的Hash表冲突的各种解决方法么？若要降低冲突发生的概率到1%，就要将BitSet的长度设置为URL个数的100倍。

实质上上面的算法都忽略了一个重要的隐含条件：允许小概率的出错，不一定要100%准确！也就是说少量url实际上没有没网络蜘蛛访问，而将它们错判为已访问的代价是很小的——大不了少抓几个网页呗。

# 概述

Bloom Filter(布隆过滤器)是由Bloom在1970年提出的一种多哈希函数映射的快速查找算法。它是一种空间高效的概率型数据结构，通常应用在一些需要快速判断某个元素是否属于集合，但是并不严格要求100%正确的场合。

布隆过滤器可以精确地代表一个集合（不是准确代表集合，精确程度由用户的具体设计决定，做到100%的正确是不可能的），**可精确（不是准确）判断某一元素是否在此集合中**。

布隆过滤器的优势在于，利用很少的空间可以做到精确率较高。

**哈希表与布隆过滤器：**

哈希表也能用于判断元素是否在集合中，但是Bloom Filter只需要哈希表的1/8或1/4的空间复杂度就能完成同样的问题。Bloom Filter可以插入元素，但是不可以删除已有元素。集合中的元素越多，误报率越大，但是不会漏报。

**参考：**

http://www.cnblogs.com/heaad/archive/2011/01/02/1924195.html

http://blog.csdn.net/jiaomeng/article/details/1495500

http://pages.cs.wisc.edu/~cao/papers/summary-cache/node8.html 哈希函数个数k、位数组大小m 测试论证

# 原理

如果想判断一个元素是不是在一个集合中，一般想到的是将所有元素保存起来，然后通过比对来判定是否在集合内：链表、树等数据结构都是这种思路。但是随着集合中元素数目的增加，我们需要的存储空间越来越大，检索速度也越来越慢（O(n),O(logn)）。

可以利用Bitmap：只要检查对应点是不是1就可以知道集合中有没有这个数。这就是布隆过滤器的基本思想。

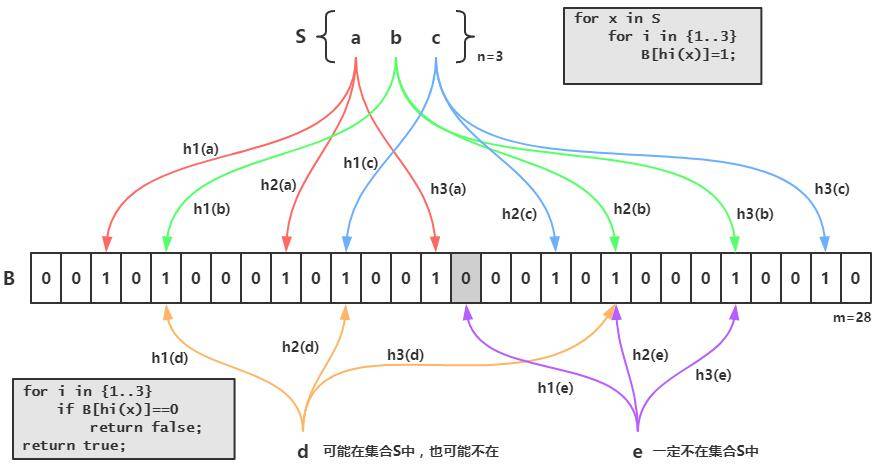






# 算法

Bloom filter可以看做是对bitmap的扩展。只是使用多个hash映射函数，从而减低hash发生冲突的概率。算法如下：



1、创建m位的bitset，初始化为0，选中k个不同的哈希函数

2、第i个hash函数对字符串str哈希的结果记为h(i,str)，范围是（0，m-1）

3、将字符串记录到bitset的过程（插入）：对于一个字符串str，分别记录h(1，str)，h(2，str)...，h(k，str)。然后将bitset的h(1，str)，h(2，str)...，h(k，str)位置1。也就是将一个str映射到bitset的k个二进制位。

4、检查字符串是否存在（查找）：对于字符串str，分别计算h(1，str)、h(2，str),...,h(k，str)。然后检查BitSet的第h(1，str)、h(2，str)，...，h(k，str) 位是否为1，若其中任何一位不为1则可以判定str一定没有被记录过。若全部位都是1，则“认为”字符串str存在。但是若一个字符串对应的Bit全为1，实际上是不能100%的肯定该字符串被Bloom Filter记录过的。（因为有可能该字符串的所有位都刚好是被其他字符串所对应）这种将该字符串划分错的情况，称为false positive 。

5、删除字符串（删除）：字符串加入了就被不能删除了，因为删除会影响到其他字符串。实在需要删除字符串的可以使用Counting bloomfilter(CBF)。

Bloom Filter使用了k个哈希函数，每个字符串跟k个bit对应。从而降低了冲突的概率。

# 参数

## Bitarray大小

**最优的哈希函数个数，位数组m大小**

哈希函数的选择对性能的影响应该是很大的，一个好的哈希函数要能近似等概率的将字符串映射到各个Bit。选择k个不同的哈希函数比较麻烦，一种简单的方法是选择一个哈希函数，然后送入k个不同的参数。

在原始个数位n时，那这里的k应该取多少呢？位数组m大小应该取多少呢？这里有个计算公式：



当满足这个条件时，错误率最小。

假设错误率为0.01， 此时m 大概是 n 的13倍，k大概是8个。 这里的n是元素记录的个数，m是bit位个数。如果每个元素的长度原大于13，使用Bloom Filter就可以节省内存。

## 错误率估计

布隆过滤器的bitarray大小如何确定？



大小为m，样本数量为n，失误率为p。

假设n=100亿，p=0.01%，单个样本大小不影响布隆过滤器大小，只影响了哈希函数的实现细节。

根据上述公式，求得m=19.19n，向上取整（肯定是向上取整，不然剩下的一个元素没有地方放了）为20n。

2000亿bit，约为25G。



求得k=14。

求得m=20n，k=14，则使用：



求得真实失误率p=0.006%，占用大小为25G。

# 特点

## 优点

相比于其它的数据结构，布隆过滤器在空间和时间方面都有巨大的优势。布隆过滤器存储空间和插入/查询时间都是常数O(k)，另外，散列函数相互之间没有关系，方便由硬件并行实现。布隆过滤器不需要存储元素本身，在某些对保密要求非常严格的场合有优势。

布隆过滤器可以表示全集，其它任何数据结构都不能。

## 缺点

但是布隆过滤器的缺点和优点一样明显。

误算率是其中之一。随着存入的元素数量增加，误算率随之增加。但是如果元素数量太少，则使用散列表足矣。

另外，一般情况下不能从布隆过滤器中删除元素。我们很容易想到把位数组变成整数数组，每插入一个元素相应的计数器加1, 这样删除元素时将计数器减掉就可以了。然而要保证安全地删除元素并非如此简单。首先我们必须保证删除的元素的确在布隆过滤器里面。这一点单凭这个过滤器是无法保证的。另外计数器回绕也会造成问题。

在降低误算率方面，有不少工作，使得出现了很多布隆过滤器的变种。

# 实现示例

#define SIZE 15\*1024\*1024

char a[SIZE]; /\* 15MB\*8 = 120M bit空间 \*/

memset(a,0,SIZE);

int seeds[] = { 5, 7, 11, 13, 31, 37, 61};

int hashcode(int cap,int seed, string key){

int hash = 0;

for (int i=0;i<key.length();i++){

hash = (seed\*hash +key.charAt(i));

}

return hash & (cap-1);

}

对每个字符串str求哈希就可以使用 hashcode(SIZE\*8,seeds[i],str) ,i 的取值范围就是（0，k）。

# 应用

布隆过滤器的巨大用处就是，能够迅速判断一个元素是否在一个集合中。因此他有如下三个使用场景：

1. 网页爬虫对URL的去重，避免爬取相同的URL地址
2. 反垃圾邮件，从数十亿个垃圾邮件列表中判断某邮箱是否垃圾邮箱（同理，垃圾短信）
3. 缓存击穿，将已存在的缓存放到布隆过滤器中，当黑客访问不存在的缓存时迅速返回避免缓存及DB挂掉。

布隆过滤器的典型应用有：

1. 数据库防止穿库。

Google Bigtable，HBase和Cassandra以及Postgresql使用BloomFilter来减少不存在的行或列的磁盘查找。避免代价高昂的磁盘查找会大大提高数据库查询操作的性能。

1. 业务场景中判断用户是否阅读过某视频或文章，比如抖音或头条，当然会导致一定的误判，但不会让用户看到重复的内容。
2. 缓存宕机、缓存击穿场景，一般判断用户是否在缓存中，如果在则直接返回结果，不在则查询db，如果来一波冷数据，会导致缓存大量击穿，造成雪崩效应，这时候可以用布隆过滤器当缓存的索引，只有在布隆过滤器中，才去查询缓存，如果没查询到，则穿透到db。如果不在布隆器中，则直接返回。
3. WEB拦截器，如果相同请求则拦截，防止重复被攻击。用户第一次请求，将请求参数放入布隆过滤器中，当第二次请求时，先判断请求参数是否被布隆过滤器命中。可以提高缓存命中率。Squid网页代理缓存服务器在cache digests中就使用了布隆过滤器。Google Chrome浏览器使用了布隆过滤器加速安全浏览服务
4. Venti文档存储系统也采用布隆过滤器来检测先前存储的数据。
5. SPIN模型检测器也使用布隆过滤器在大规模验证问题时跟踪可达状态空间。

拼写检查一类的字典应用

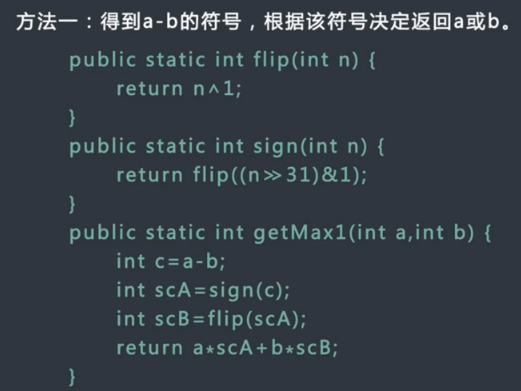
数据库系统

网络领域（爬虫，web cache sharing）

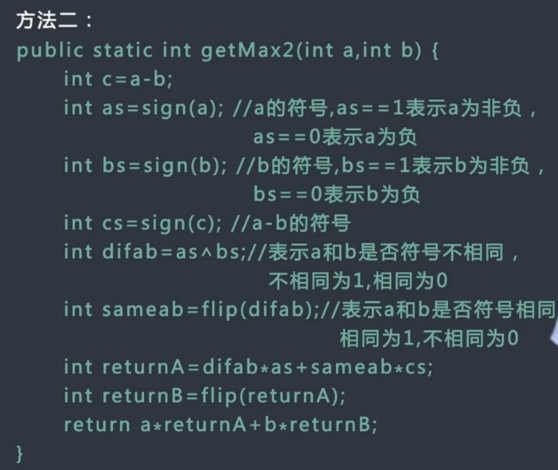
## 求较大值

**题目：**给定两个32位整数a和b，返回a和b中较大的，但是不能用任何比较判断。

**分析：**



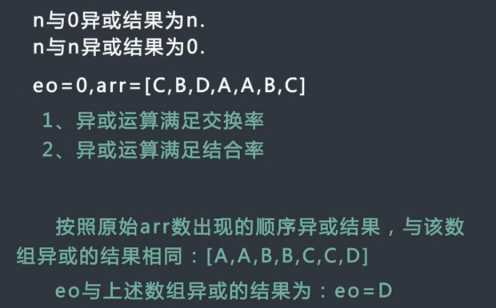
注：方法一可能存在问题，当a-b溢出时，会发生错误。



## 查找奇数次数字

**题目：**给定一个整型数组arr，其中只有一个数出现了奇数次，其他的数都出现了偶数次，请打印这个数。要求时间复杂度为O(N)，额外空间复杂度为O(1)。

**分析：**



**拓展：**给定一个整型数组arr，其中有两个数出现了奇数次，其他的数都出现了偶数次，请打印这两个数。要求时间复杂度为O(N)，额外空间复杂度为O(1)。



## 加密解密