http://blog.csdn.net/hguisu/article/details/7856239

前面我们说海量数据处理提到，从算法的角度去考虑处理海量数据。

**[1. Bloom Filter](http://blog.csdn.net/hguisu/article/details/7866173)**

Bloom Filter（BF）是一种空间效率很高的随机数据结构，它利用位数组很简洁地表示一个集合，并能判断一个元素是否属于这个集合。它是一个判断元素是否存在集合的快速的概率算法。Bloom Filter有可能会出现错误判断，但不会漏掉判断。也就是Bloom Filter判断元素不在集合，那肯定不在。如果判断元素存在集合中，有一定的概率判断错误。因此，Bloom Filter不适合那些“零错误”的应用场合。而在能容忍低错误率的应用场合下，Bloom Filter比其他常见的算法（如hash，折半查找）极大节省了空间。

【适用范围】   
可以用来实现数据字典，进行数据的判重，或者集合求交集   
【基本原理及要点】

原理要点：一是位数组， 而是k个独立hash函数。

**1）位数组：**

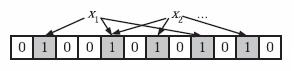
假设Bloom Filter使用一个m比特的数组来保存信息，初始状态时，Bloom Filter是一个包含m位的位数组，每一位都置为0，即BF整个数组的元素都设置为0。

http://my.csdn.net/uploads/201208/12/1344737647_1631.jpg

2）k个独立hash函数

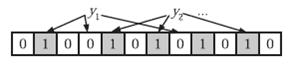
      为了表达S={x1, x2,…,xn}这样一个n个元素的集合，Bloom Filter使用k个相互独立的哈希函数（Hash Function），它们分别将集合中的每个元素映射到{1,…,m}的范围中。

当我们往Bloom Filter中增加任意一个元素x时候，我们使用k个哈希函数得到k个哈希值，然后将数组中对应的比特位设置为1。即第i个哈希函数映射的位置hashi(x)就会被置为1（1≤i≤k）。 注意，如果一个位置多次被置为1，那么只有第一次会起作用，后面几次将没有任何效果。在下图中，k=3，且有两个哈希函数选中同一个位置（从左边数第五位，即第二个“1“处）。



 3）判断元素是否存在集合

    在判断y是否属于这个集合时，我们只需要对y使用k个哈希函数得到k个哈希值，如果所有hashi(y)的位置都是1（1≤i≤k），即k个位置都被设置为1了，那么我们就认为y是集合中的元素，否则就认为y不是集合中的元素。下图中y1就不是集合中的元素（因为y1有一处指向了“0”位）。y2或者属于这个集合，或者刚好是一个false positive。



      显然这个判断并不保证查找的结果是100%正确的。

Bloom Filter的缺点：

       1）Bloom Filter无法从Bloom Filter集合中删除一个元素。因为该元素对应的位会牵动到其他的元素。所以一个简单的改进就是 counting Bloom filter，用一个counter数组代替位数组，就可以支持删除了。 此外，Bloom Filter的hash函数选择会影响算法的效果。

       2）还有一个比较重要的问题，如何根据输入元素个数n，确定位数组m的大小及hash函数个数，即hash函数选择会影响算法的效果**。**当hash函数个数k=(ln2)\*(m/n)时错误率最小。在错误率不大于E的情况下，m至少要等于m=n\*lg(1/E) 才能表示任意n个元素的集合。但m还应该更大些，因为还要保证bit数组里至少一半为0，则m应该m>=n\*lg(1/E)\*lge，大概是nlg(1/E)1.44倍。(lg表示以2为底的对数)。

举个例子我们假设错误率为0.01，则此时m应大概是n的13倍。这样k大概是8个。

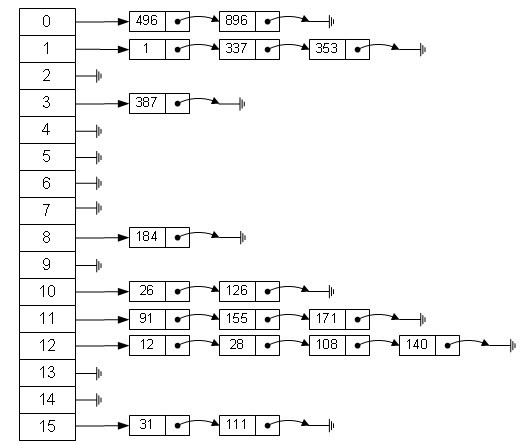
 注意：

         这里m与n的单位不同，m是以bit为单位，而n则是以元素个数为单位(准确的说是不同元素的个数)。通常单个元素的长度都是有很多bit的。所以使用bloom filter内存上通常都是节省的。

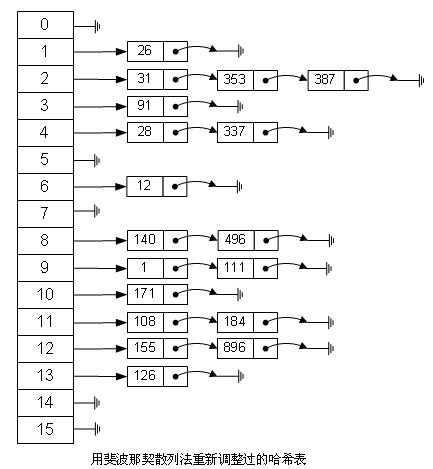
       一般BF可以与一些key-value的数据库一起使用，来加快查询。由于BF所用的空间非常小，所有BF可以常驻内存。这样子的话，对于大部分不存在的元素，我们只需要访问内存中的BF就可以判断出来了，只有一小部分，我们需要访问在硬盘上的key-value数据库。从而大大地提高了效率。

【扩展】   
Bloom filter将集合中的元素映射到位数组中，用k（k为哈希函数个数）个映射位是否全1表示元素在不在这个集合中。Counting bloom filter（CBF）将位数组中的每一位扩展为一个counter，从而支持了元素的删除操作。Spectral Bloom Filter（SBF）将其与集合元素的出现次数关联。SBF采用counter中的最小值来近似表示元素的出现频率。    
【问题实例】   
给你A,B两个文件，各存放50亿条URL，每条URL占用64字节，内存限制是4G，让你找出A,B文件共同的URL。如果是三个乃至n个文件呢？   
根据这个问题我们来计算下内存的占用，4G=2^32大概是40亿\*8大概是340亿bit，n=50亿，如果按出错率0.01算需要的大概是650亿个bit。 现在可用的是340亿，相差并不多，这样可能会使出错率上升些。另外如果这些urlip是一一对应的，就可以转换成ip，则大大简单了。

**2. Hash**

【什么是Hash】   
       Hash，一般翻译做“散列”，也有直接音译为“哈希”的，就是把任意长度的输入（又叫做预映射， pre-image），通过散列算法，变换成固定长度的输出，该输出就是散列值。这种转换是一种压缩映射，也就是，散列值的空间通常远小于输入的空间，不 同的输入可能会散列成相同的输出，而不可能从散列值来唯一的确定输入值。简单的说就是一种将任意长度的消息压缩到某一固定长度的消息摘要的函数。   
       HASH主要用于信息安全领域中加密算法，它把一些不同长度的信息转化成杂乱的128位的编码,这些编码值叫做HASH值. 也可以说，hash就是找到一种数据内容和数据存放地址之间的映射关系。   
      数组的特点是：寻址容易，插入和删除困难；而链表的特点是：寻址困难，插入和删除容易。那么我们能不能综合两者的特性，做出一种寻址容易，插入删除也容易的数据结构？答案是肯定的，这就是我们要提起的哈希表，哈希表有多种不同的实现方法，我接下来解释的是最常用的一种方法——拉链法，（也是树的一种存储结构，称为二叉链表）我们可以理解为“链表的数组”，如图：   
     左边很明显是个数组，数组的每个成员包括一个指针，指向一个链表的头，当然这个链表可能为空，也可能元素很多。我们根据元素的一些特征把元素分配到不同的链表中去，也是根据这些特征，找到正确的链表，再从链表中找出这个元素。   
元素特征转变为数组下标的方法就是散列法。

散列法当然不止一种，下面列出三种比较常用的：  
1，除法散列法 （求模数）  
最直观的一种，上图使用的就是这种散列法，公式：   
index = value % 16   
学过汇编的都知道，求模数其实是通过一个除法运算得到的，所以叫“除法散列法”。   
2，平方散列法   
求index是非常频繁的操作，而乘法的运算要比除法来得省时（对现在的CPU来说，估计我们感觉不出来），所以我们考虑把除法换成乘法和一个位移操作。公式：   
index = (value \* value) >> 28   
如果数值分配比较均匀的话这种方法能得到不错的结果，但我上面画的那个图的各个元素的值算出来的index都是0——非常失败。也许你还有个问题，value如果很大，value \* value不会溢出吗？答案是会的，但我们这个乘法不关心溢出，因为我们根本不是为了获取相乘结果，而是为了获取index。   
3，斐波那契（Fibonacci）散列法   
平方散列法的缺点是显而易见的，所以我们能不能找出一个理想的乘数，而不是拿value本身当作乘数呢？答案是肯定的。   
1，对于16位整数而言，这个乘数是40503   
2，对于32位整数而言，这个乘数是2654435769   
3，对于64位整数而言，这个乘数是11400714819323198485   
这几个“理想乘数”是如何得出来的呢？这跟一个法则有关，叫黄金分割法则，而描述黄金分割法则的最经典表达式无疑就是著名的斐波那契数列，如果你还有兴趣，就到网上查找一下“斐波那契数列”等关键字，我数学水平有限，不知道怎么描述清楚为什么，另外斐波那契数列的值居然和太阳系八大行星的轨道半径的比例出奇吻合，很神奇，对么？  
对我们常见的32位整数而言，公式：   
index = (value \* 2654435769) >> 28   
如果用这种斐波那契散列法的话，那我上面的图就变成这样了：

  
很明显，用斐波那契散列法调整之后要比原来的取摸散列法好很多。   
【适用范围】   
快速查找，删除的基本数据结构，通常需要总数据量可以放入内存。   
【基本原理及要点】   
hash函数选择，针对字符串，整数，排列，具体相应的hash方法。   
碰撞处理：

一种是open hashing，也称为拉链法；

另一种就是closed hashing，也称开地址法，opened addressing。   
【扩展】   
d-left hashing中的d是多个的意思，我们先简化这个问题，看一看2-left hashing。2-left hashing指的是将一个哈希表分成长度相等的两半，分别叫做T1和T2，给T1和T2分别配备一个哈希函数，h1和h2。在存储一个新的key时，同时用两个哈希函数进行计算，得出两个地址h1[key]和h2[key]。这时需要检查T1中的h1[key]位置和T2中的h2[key]位置，哪一个位置已经存储的（有碰撞的）key比较多，然后将新key存储在负载少的位置。如果两边一样多，比如两个位置都为空或者都存储了一个key，就把新key存储在左边的T1子表中，2-left也由此而来。在查找一个key时，必须进行两次hash，同时查找两个位置。   
【问题实例】   
1).海量日志数据，提取出某日访问百度次数最多的那个IP。   
IP的数目还是有限的，最多2^32个，所以可以考虑使用hash将ip直接存入内存，然后进行统计。

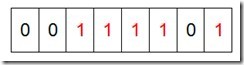
**3. Bit-map**

【什么是Bit-map】   
        所谓的[Bit-map](http://blog.csdn.net/hguisu/article/details/7880288" \t "_blank)就是用一个bit位来标记某个元素对应的Value， 而Key即是该元素。由于采用了Bit为单位来存储数据，因此在存储空间方面，可以大大节省。

        如果说了这么多还没明白什么是Bit-map，那么我们来看一个具体的例子，假设我们要对0-7内的5个元素(4,7,2,5,3)排序（这里假设这些元 素没有重复）。那么我们就可以采用Bit-map的方法来达到排序的目的。要表示8个数，我们就只需要8个Bit（1Bytes），首先我们开辟 1Byte的空间，将这些空间的所有Bit位都置为0(如下图：)

http://my.csdn.net/uploads/201208/13/1344854370_5847.jpg  
然后遍历这5个元素，首先第一个元素是4，那么就把4对应的位置为1（可以这样操作 p+(i/8)|(0x01<<(i%8)) 当然了这里的操作涉及到Big-ending和Little-ending的情况，这里默认为Big-ending）,因为是从零开始的，所以要把第五位 置为一（如下图）：

http://my.csdn.net/uploads/201208/13/1344854388_8094.jpg  
然后再处理第二个元素7，将第八位置为1,，接着再处理第三个元素，一直到最后处理完所有的元素，将相应的位置为1，这时候的内存的Bit位的状态如下：

  
然后我们现在遍历一遍Bit区域，将该位是一的位的编号输出（2，3，4，5，7），这样就达到了排序的目的。下面的代码给出了一个BitMap的用法：排序。

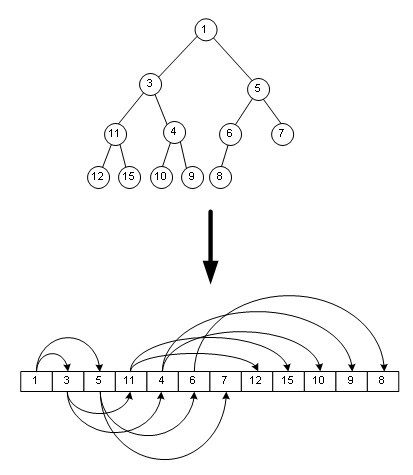
【适用范围】   
可进行数据的快速查找，判重，删除，一般来说数据范围是int的10倍以下

【基本原理及要点】  
使用bit数组来表示某些元素是否存在，比如8位电话号码   
【扩展】   
[Bloom filter](http://blog.csdn.net/hguisu/article/details/7866173)可以看做是对[bit-map](http://blog.csdn.net/hguisu/article/details/7880288)的扩展   
【问题实例】   
1)已知某个文件内包含一些电话号码，每个号码为8位数字，统计不同号码的个数。   
8位最多99 999 999，大概需要99m个bit（1024\*1024 \*99个bit ），大概10几m字节的内存即可。

申请内存空间的大小为:int a[1 + N/32] =（(99 999 999/32 +1)\*4 个字节/1024/1024 = 1.2M

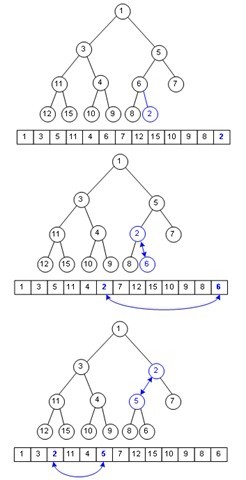
（可以理解为从0-99 999 999的数字，每个数字对应一个Bit位，所以只需要99M个Bit==1.2MBytes，这样，就用了小小的1.2M左右的内存表示了所有的8位数的电话）   
2)2.5亿个整数中找出不重复的整数的个数，内存空间不足以容纳这2.5亿个整数。   
将bit-map扩展一下，用2bit表示一个数即可，0表示未出现，1表示出现一次，2表示出现2次及以上，在遍历这些数的时候，如果对应位置的值是 0，则将其置为1；如果是1，将其置为2；如果是2，则保持不变。或者我们不用2bit来进行表示，我们用两个bit-map即可模拟实现这个2bit- map，都是一样的道理。

**4. 堆**

【什么是堆】  
概念：堆是一种特殊的二叉树，具备以下两种性质  
1）每个节点的值都大于（或者都小于，称为最小堆）其子节点的值  
2）树是完全平衡的，并且最后一层的树叶都在最左边  
这样就定义了一个最小堆。如下图用一个数组来表示堆： 

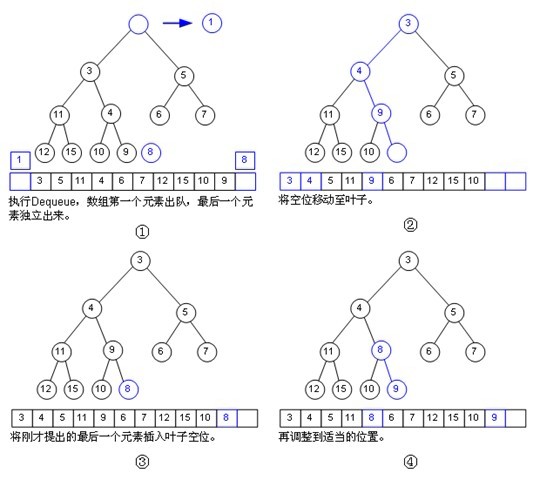
那么下面介绍二叉堆：二叉堆是一种完全二叉树，其任意子树的左右节点（如果有的话）的键值一定比根节点大，上图其实就是一个二叉堆。

你一定发觉了，最小的一个元素就是数组第一个元素，那么二叉堆这种有序队列如何入队呢？看图：



假设要在这个二叉堆里入队一个单元，键值为2，那只需在数组末尾加入这个元素，然后尽可能把这个元素往上挪，直到挪不动，经过了这种复杂度为Ο(logn)的操作，二叉堆还是二叉堆。

那如何出队呢？也不难，看图



出队一定是出数组的第一个元素，这么来第一个元素以前的位置就成了空位，我们需要把这个空位挪至叶子节点，然后把数组最后一个元素插入这个空位，把这个“空位”尽量往上挪。这种操作的复杂度也是Ο(logn)。

【适用范围】  
海量数据前n大，并且n比较小，堆可以放入内存

【基本原理及要点】  
最大堆求前n小，最小堆求前n大。方法，比如求前n小，我们比较当前元素与最大堆里的最大元素，如果它小于最大元素，则应该替换那个最大元素。这样最后得到的n个元素就是最小的n个。适合大数据量，求前n小，n的大小比较小的情况，这样可以扫描一遍即可得到所有的前n元素，效率很高。

【扩展】  
双堆，一个最大堆与一个最小堆结合，可以用来维护中位数。

【问题实例】  
1)100w个数中找最大的前100个数。  
用一个100个元素大小的最小堆即可。

**5. 双层桶**

【什么是双层桶】    
事实上，与其说双层桶划分是一种数据结构，不如说它是一种算法设计思想。面对一堆大量的数据我们无法处理的时候，我们可以将其分成一个个小的单元，然后根据一定的策略来处理这些小单元，从而达到目的。

【适用范围】   
第k大，中位数，不重复或重复的数字

【基本原理及要点】   
因为元素范围很大，不能利用直接寻址表，所以通过多次划分，逐步确定范围，然后最后在一个可以接受的范围内进行。可以通过多次缩小，双层只是一个例子，分治才是其根本（只是“只分不治”）。

【扩展】   
当有时候需要用一个小范围的数据来构造一个大数据，也是可以利用这种思想，相比之下不同的，只是其中的逆过程。

【问题实例】   
1).2.5亿个整数中找出不重复的整数的个数，内存空间不足以容纳这2.5亿个整数。

有点像鸽巢原理，整数个数为2^32,也就是，我们可以将这2^32个数，划分为2^8=256个区域(比如用单个文件代表一个区域)，然后将数据分离到不同的区 域，然后不同的区域在利用bitmap就可以直接解决了。也就是说只要有足够的磁盘空间，就可以很方便的解决。 当然这个题也可以用我们前面讲过的BitMap方法解决，正所谓条条大道通罗马~~~

2).5亿个int找它们的中位数。

这个例子比上面那个更明显。首先我们将int划分为2^16个区域，然后读取数据统计落到各个区域里的数的个数，之后我们根据统计结果就可以判断中位数落到那个区域，同时知道这个区域中的第几大数刚好是中位数。然后第二次扫描我们只统计落在这个区域中的那些数就可以了。

实际上，如果不是int是int64，我们可以经过3次这样的划分即可降低到可以接受的程度。即可以先将int64分成2^24个区域，然后确定区域的第几大数，在将该区域分成2^20个子区域，然后确定是子区域的第几大数，然后子区域里的数的个数只有2^20，就可以直接利用direct addr table进行统计了。

3).现在有一个0-30000的随机数生成器。请根据这个随机数生成器，设计一个抽奖范围是0-350000彩票中奖号码列表，其中要包含20000个中奖号码。

这个题刚好和上面两个思想相反，一个0到3万的随机数生成器要生成一个0到35万的 随机数。那么我们完全可以将0-35万的区间分成35/3=12个区 间，然后每个区间的长度都小于等于3万，这样我们就可以用题目给的随机数生成器来生成了，然后再加上该区间的基数。那么要每个区间生成多少个随机数呢？计 算公式就是：区间长度\*随机数密度，在本题目中就是30000\*（20000/350000）。最后要注意一点，该题目是有隐含条件的：彩票，这意味着你 生成的随机数里面不能有重复，这也是我为什么用双层桶划分思想的另外一个原因。

**6. 数据库索引及优化**

索引是对数据库表中一列或多列的值进行排序的一种结构，使用索引可快速访问数据库表中的特定信息。

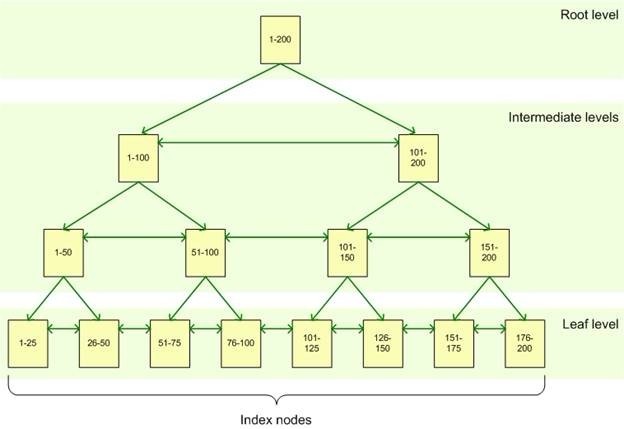
数据库索引

什么是索引

　　数据库索引好比是一本书前面的目录，能加快数据库的查询速度。  
　　例如这样一个查询：select \* from table1 where id=44。如果没有索引，必须遍历整个表，直到ID等于44的这一行被找到为止；有了索引之后(必须是在ID这一列上建立的索引)，直接在索引里面找 44（也就是在ID这一列找），就可以得知这一行的位置，也就是找到了这一行。可见，索引是用来定位的。  
　　索引分为聚簇索引和非聚簇索引两种，聚簇索引 是按照数据存放的物理位置为顺序的，而非聚簇索引就不一样了；聚簇索引能提高多行检索的速度，而非聚簇索引对于单行的检索很快。

概述

　　建立索引的目的是加快对表中记录的查找或排序。  
　　为表设置索引要付出代价的：一是增加了数据库的存储空间，二是在插入和修改数据时要花费较多的时间(因为索引也要随之变动)。



B树索引-Sql Server索引方式

为什么要创建索引

　　创建索引可以大大提高系统的性能。  
　　　　第一，通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。  
　　　　第二，可以大大加快数据的检索速度，这也是创建索引的最主要的原因。  
　　　　第三，可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。  
　　　　第四，在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。  
　　　　第五，通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。  
　　也许会有人要问：增加索引有如此多的优点，为什么不对表中的每一个列创建一个索引呢？因为，增加索引也有许多不利的方面。  
　　　　第一，创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。  
　　　　第二，索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。  
　　　　第三，当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

在哪建索引

　　索引是建立在数据库表中的某些列的上面。在创建索引的时候，应该考虑在哪些列上可以创建索引，在哪些列上不能创建索引。一般来说，应该在这些列上创建索引：  
　　在经常需要搜索的列上，可以加快搜索的速度；  
　　在作为主键的列上，强制该列的唯一性和组织表中数据的排列结构；  
　　在经常用在连接的列上，这些列主要是一些外键，可以加快连接的速度；在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的；  
　　在经常需要排序的列上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间；  
　　在经常使用在WHERE子句中的列上面创建索引，加快条件的判断速度。  
　　同样，对于有些列不应该创建索引。一般来说，不应该创建索引的的这些列具有下列特点：  
　　第一，对于那些在查询中很少使用或者参考的列不应该创建索引。这是因为，既然这些列很少使用到，因此有索引或者无索引，并不能提高查询速度。相反，由于增加了索引，反而降低了系统的维护速度和增大了空间需求。  
　　第二，对于那些只有很少数据值的列也不应该增加索引。这是因为，由于这些列的取值很少，例如人事表的性别列，在查询的结果中，结果集的数据行占了表中数据行的很大比例，即需要在表中搜索的数据行的比例很大。增加索引，并不能明显加快检索速度。  
　　第三，对于那些定义为text, image和bit数据类型的列不应该增加索引。这是因为，这些列的数据量要么相当大，要么取值很少,不利于使用索引。  
　　第四，当修改性能远远大于检索性能时，不应该创建索引。这是因为，修改性能和检索性能是互相矛盾的。当增加索引时，会提高检索性能，但是会降低修改性能。当减少索引时，会提高修改性能，降低检索性能。因此，当修改操作远远多于检索操作时，不应该创建索引。

数据库优化

　　此外，除了数据库索引之外，在LAMP结果如此流行的今天，数据库（尤其是MySQL）性能优化也是海量数据处理的一个热点。下面就结合自己的经验，聊一聊MySQL数据库优化的几个方面。  
　　首先，在数据库设计的时候，要能够充分的利用索引带来的性能提升，至于如何建立索引，建立什么样的索引，在哪些字段上建立索引，上面已经讲的很清楚 了，这里不在赘述。另外就是设计数据库的原则就是尽可能少的进行数据库写操作（插入，更新，删除等），查询越简单越好。如下：

**数据库设计：**

. 创建索引

. 查询语句

1）查询越简单越好:单表查询 > inner join >其他

        2）更新越少越好

　　其次，配置缓存是必不可少的，配置缓存可以有效的降低数据库查询读取次数，从而缓解数据库服务器压力，达到优化的目的，一定程度上来讲，这算是一个 “围魏救赵”的办法。可配置的缓存包括索引缓存(key\_buffer)，排序缓存(sort\_buffer)，查询缓存(query\_buffer)， 表描述符缓存(table\_cache)，如下：

**配置缓存：**

. 索引缓存（key\_buffer）

. 排序缓存 （sort\_buffer）

. 查询缓存  （query\_buffer）

. 表描述符缓存（table\_cache）

　　第三，切表，切表也是一种比较流行的数据库优化法。分表包括两种方式：横向分表 和纵向分表，其中，横向分表比较有使用意义，故名思议，横向切表就是指把记录分到不同的表中，而每条记录仍旧是完整的（纵向切表后每条记录是不完整的）， 例如原始表中有100条记录，我要切成2个表，那么最简单也是最常用的方法就是ID取摸切表法，本例中，就把ID为1,3,5,7。。。的记录存在一个表 中，ID为2,4,6,8,。。。的记录存在另一张表中。虽然横向切表可以减少查询强度，但是它也破坏了原始表的完整性，如果该表的统计操作比较多，那么 就不适合横向切表。横向切表有个非常典型的用法，就是用户数据：每个用户的用户数据一般都比较庞大，但是每个用户数据之间的关系不大，因此这里很适合横向 切表。最后，要记住一句话就是：分表会造成查询的负担，因此在数据库设计之初，要想好是否真的适合切表的优化：

**切表分表：**

. 纵向 ：字段较多时可以考虑，一般用处不到

. 横向 ：1）能有效降低表的大小，减少由于枷锁导致的等待

             2）查询会变得复杂，尤其是需要排序的查询

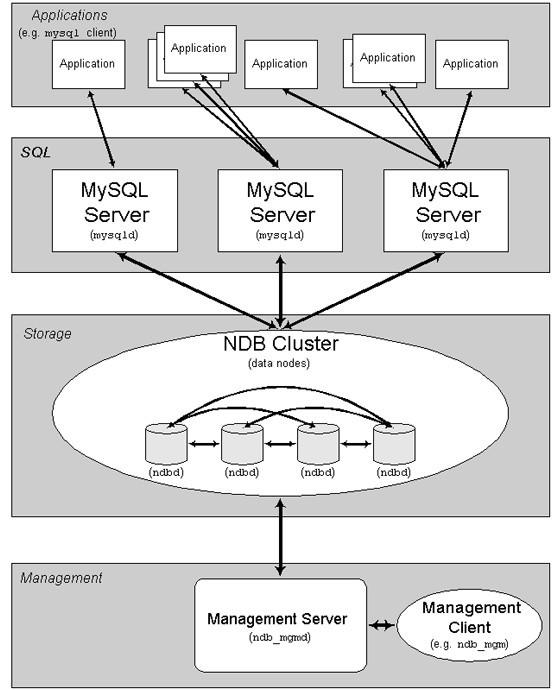
第四，日志分析，在数据库运行了较长一段时间以后，会积累大量的LOG日志，其实这里面的蕴涵的有用的信息量还是很大的。通过分析日志，可以找到系统性能的瓶颈，从而进一步寻找优化方案。

**数据库性能分析：**

. 查询吞吐量，数据量监控

. 慢查询分析：索引，I/O,cpu等。

以上讲的都是单机MySQL的性能优化的一些经验，但是随着信息大爆炸，单机的数据库服务器已经不能满足我们的需求，于是，多多节点，分布式数据库网络出现了，其一般的结构如下：



分布式数据库结构

这种分布式集群的技术关键就是“同步复制”。。。

**7. 倒排索引(搜索引擎之基石)**

引言：

在信息大爆炸的今天，有了搜索引擎的帮助，使得我们能够快速，便捷的找到所求。提到搜索引擎，就不得不说VSM模型，说到VSM，就不得不聊倒排索引。可以毫不夸张的讲，倒排索引是搜索引擎的基石。

VSM检索模型

VSM全称是Vector Space Model(向量空间模型)，是IR(Information Retrieval信息检索)模型中的一种，由于其简单，直观，高效，所以被广泛的应用到搜索引擎的架构中。98年的Google就是凭借这样的一个模 型，开始了它的疯狂扩张之路。废话不多说，让我们来看看到底VSM是一个什么东东。

在开始之前，我默认大家对线性代数里面的向量(Vector)有一定了解的。向量是既有大小又有方向的量，通常用有向线段表示，向量有：加、减、倍数、内积、距离、模、夹角的运算。

文档(Document)：一个完整的信息单元，对应的搜索引擎系统里，就是指一个个的网页。

标引项(Term)：文档的基本构成单位，例如在英文中可以看做是一个单词，在中文中可以看作一个词语。

查询(Query)：一个用户的输入，一般由多个Term构成。

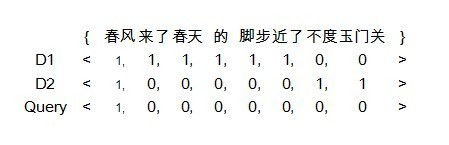
那么用一句话概况搜索引擎所做的事情就是：对于用户输入的Query，找到最相似的Document返回给用户。而这正是IR模型所解决的问题：

信息检索模型是指如何对查询和文档进行表示，然后对它们进行相似度计算的框架和方法。

举个简单的例子：

现在有两篇文章(Document)分别是 “春风来了，春天的脚步近了” 和 “春风不度玉门关”。然后输入的Query是“春风”，从直观上感觉，前者和输入的查询更相关一些，因为它包含有2个春，但这只是我们的直观感觉，如何量 化呢，要知道计算机是门严谨的学科^\_^。这个时候，我们前面讲的Term和VSM模型就派上用场了。

首先我们要确定向量的维数，这时候就需要一个字典库，字典库的大小，即是向量的维数。在该例中，字典为{春风,来了,春天, 的,脚步,近了,不度,玉门关} ，文档向量，查询向量如下图：

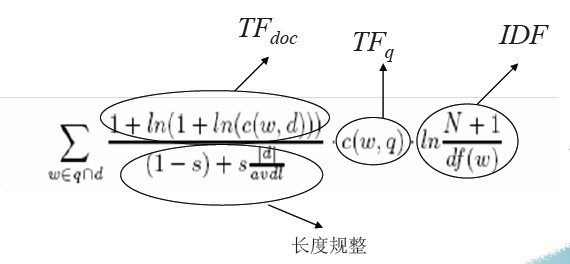


VSM模型示例

PS:为了简单起见，这里分词的粒度很大。

将Query和Document都量化为向量以后，那么就可以计算用户的查询和哪个 文档相似性更大了。简单的计算结果是D1和D2同Query的内积都是1，囧。当然了，如果分词粒度再细一些，查询的结果就是另外一个样子了，因此分词的 粒度也是会对查询结果（主要是召回率和准确率）造成影响的。

上述的例子是用一个很简单的例子来说明VSM模型的，计算文档相似度的时候也是采用最原始的内积的方法，并且只考虑了词频(TF)影响因子，而没有考虑反词频(IDF)，而现在比较常用的是cos夹角法，影响因子也非常多，据传Google的影响因子有100+之多。  
大名鼎鼎的Lucene项目就是采用VSM模型构建的，VSM的核心公式如下（由cos夹角法演变，此处省去推导过程）



VSM模型公式

从上面的例子不难看出，如果向量的维度(对汉语来将，这个值一般在30w-45w)变大，而且文档数量(通常都是海量的)变多，那么计算一次相关性，开销是非常大的，如何解决这个问题呢？不要忘记了我们这节的主题就是 倒排索引，主角终于粉墨登场了！！！

倒排索引非常类似我们前面提到的Hash结构。以下内容来自维基百科：

倒排索引（英语：Inverted index），也常被称为反向索引、置入档案或反向档案，是一种索引方法，被用来存储在全文搜索下某个单词在一个文档或者一组文档中的存储位置的映射。它是文档检索系统中最常用的数据结构。

有两种不同的反向索引形式：

* 一条记录的水平反向索引（或者反向档案索引）包含每个引用单词的文档的列表。
* 一个单词的水平反向索引（或者完全反向索引）又包含每个单词在一个文档中的位置。

后者的形式提供了更多的兼容性（比如短语搜索），但是需要更多的时间和空间来创建。

由上面的定义可以知道，一个倒排索引包含一个字典的索引和所有词的列表。其中字典索 引中包含了所有的Term(通俗理解为文档中的词)，索引后面跟的列表则保存该词的信息(出现的文档号，甚至包含在每个文档中的位置信息)。下面我们还采 用上面的方法举一个简单的例子来说明倒排索引。

例如现在我们要对三篇文档建立索引(实际应用中，文档的数量是海量的)：

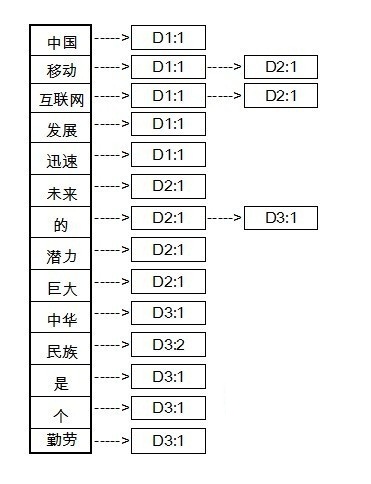
文档1(D1)：中国移动互联网发展迅速

文档2(D2)：移动互联网未来的潜力巨大

文档3(D3)：中华民族是个勤劳的民族

那么文档中的词典集合为：{中国，移动，互联网，发展，迅速，未来，的，潜力，巨大，中华，民族，是，个，勤劳}

建好的索引如下图：



倒排索引

在上面的索引中，存储了两个信息，文档号和出现的次数。建立好索引以后，我们就可以 开始查询了。例如现在有一个Query是”中国移动”。首先分词得到Term集合{中国，移动}，查倒排索引，分别计算query和d1,d2,d3的距 离。有没有发现，倒排表建立好以后，就不需要在检索整个文档库，而是直接从字典集合中找到“中国”和“移动”，然后遍历后面的列表直接计算。

对倒排索引结构我们已经有了初步的了解，但在实际应用中还有些需要解决的问题(主要是由海量数据引起的)。笔者列举一些问题，并给出相应的解决方案，抛砖以引玉，希望大家可以展开讨论：

1.左侧的索引表如何建立?怎么做才能最高效？

可能有人不假思索回答：左侧的索引当然要采取hash结构啊，这样可以快速的定位到 字典项。但是这样问题又来了，hash函数如何选取呢？而且hash是有碰撞的，但是倒排表似乎又是不允许碰撞的存在的。事实上，虽然倒排表和hash异 常的相思，但是两者还是有很大区别的，其实在这里我们可以采用前面提到的Bitmap的思想，每个Term(单词)对应一个位置(当然了，这里不是一个比 特位)，而且是一一对应的。如何能够做到呢，一般在文字处理中，有很多的编码，汉字中的GBK编码基本上就可以包含所有用到的汉字，每个汉字的GBK编码 是确定的，因此一个Term的”ID”也就确定了，从而可以做到快速定位。注：得到一个汉字的GBK号是非常快的过程，可以理解为O(1)的时间复杂度。

2.如何快速的添加删除更新索引？

有经验的码农都知道，一般在系统的“做加法”的代价比“做减法”的代价要低很多，在搜索引擎中中也不例外。因此，在倒排表中，遇到要删除一个文档，其实不是真正的删除，而是将其标记删除。这样一个减法操作的代价就比较小了。

3.那么多的海量文档，如果存储呢？有么有什么备份策略呢？

当然了，一台机器是存储不下的，分布式存储是采取的。一般的备份保存3份就足够了。

好了，倒排索引终于完工了，不足的地方请指正。谢谢

**8. 外排序**

适用范围：

大数据的排序，去重  
 基本原理及要点：

外部排序的两个独立阶段：

1）首先按内存大小，将外存上含n个记录的文件分成若干长度L的子文件或段。依次读入内存并利用有效的内部排序对他们进行排序，并将排序后得到的有序字文件重新写入外存，通常称这些子文件为归并段。

2）对这些归并段进行逐趟归并，使归并段逐渐由小到大，直至得到整个有序文件为之。

外排序的归并方法，置换选择 败者树原理，最优归并树  
 扩展：  
 问题实例：  
 1).有一个1G大小的一个文件，里面每一行是一个词，词的大小不超过16个字节，内存限制大小是1M。返回频数最高的100个词  
这个数据具有很明显的特点，词的大小为16个字节，但是内存只有1m做hash有些不够，所以可以用来排序。内存可以当输入缓冲区使用。

**9. trie树**

 适用范围：

数据量大，重复多，但是数据种类小可以放入内存

基本原理及要点：

实现方式，节点孩子的表示方式  
 扩展：

压缩实现。  
 问题实例：  
 1).有10个文件，每个文件1G， 每个文件的每一行都存放的是用户的query，每个文件的query都可能重复。要你按照query的频度排序 。  
 2).1000万字符串，其中有些是相同的(重复),需要把重复的全部去掉，保留没有重复的字符串。请问怎么设计和实现？  
 3).寻找热门查询：查询串的重复度比较高，虽然总数是1千万，但如果除去重复后，不超过3百万个，每个不超过255字节。

**10. 分布式处理 mapreduce**

基本原理及要点：

将数据交给不同的机器去处理，数据划分，结果归约。

扩 展：   
问题实例：   
  
1).The canonical example application of MapReduce is a process to count the appearances of   
  
each different word in a set of documents:   
void map(String name, String document):   
// name: document name   
// document: document contents   
for each word w in document:   
EmitIntermediate(w, 1);   
  
void reduce(String word, Iterator partialCounts):   
// key: a word   
// values: a list of aggregated partial counts   
int result = 0;   
for each v in partialCounts:   
result += ParseInt(v);   
Emit(result);   
Here, each document is split in words, and each word is counted initially with a "1" value by   
  
the Map function, using the word as the result key. The framework puts together all the pairs   
  
with the same key and feeds them to the same call to Reduce, thus this function just needs to   
  
sum all of its input values to find the total appearances of that word.   
  
2). 海量数据分布在100台电脑中，想个办法高效统计出这批数据的TOP10。   
  
3).一共有N个机器，每个机器上有N个数。每个机器最多存 O(N)个数并对它们操作。如何找到N^2个数的中数(median)？   
  
  
经典问题分析   
  
上千万or亿数据（有 重复），统计其中出现次数最多的前N个数据,分两种情况：可一次读入内存，不可一次读入。   
  
可用思路：trie树+堆，数据库索引，划分 子集分别统计，hash，分布式计算，近似统计，外排序   
  
所谓的是否能一次读入内存，实际上应该指去除重复后的数据量。如果去重后数据可 以放入内存，我们可以为数据建立字典，比如通过 map，hashmap，trie，然后直接进行统计即可。当然在更新每条数据的出现次数的时候，我们可以利用一个堆来维护出现次数最多的前N个数据，当然这样导致维护次数增加，不如完全统计后在求前N大效率高。   
  
如果数据无法放入内存。一方面我们可以考虑上面的字典方法能否被改进以适应这种情形，可以做的改变就是将字典存放到硬盘上，而不是内存，这可以参考数据库的存储方法。   
  
当 然还有更好的方法，就是可以采用分布式计算，基本上就是map-reduce过程，首先可以根据数据值或者把数据hash(md5)后的值，将数据按照范 围划分到不同的机子，最好可以让数据划分后可以一次读入内存，这样不同的机子负责处理各种的数值范围，实际上就是map。得到结果后，各个机子只需拿出各 自的出现次数最多的前N个数据，然后汇总，选出所有的数据中出现次数最多的前N个数据，这实际上就是reduce过程。   
  
实 际上可能想直接将数据均分到不同的机子上进行处理，这样是无法得到正确的解的。因为一个数据可能被均分到不同的机子上，而另一个则可能完全聚集到一个机子 上，同时还可能存在具有相同数目的数据。比如我们要找出现次数最多的前100个，我们将1000万的数据分布到10台机器上，找到每台出现次数最多的前 100个，归并之后这样不能保证找到真正的第100个，因为比如出现次数最多的第100个可能有1万个，但是它被分到了10台机子，这样在每台上只有1千 个，假设这些机子排名在1000个之前的那些都是单独分布在一台机子上的，比如有1001个，这样本来具有1万个的这个就会被淘汰，即使我们让每台机子选 出现次数最多的1000个再归并，仍然会出错，因为可能存在大量个数为1001个的发生聚集。因此不能将数据随便均分到不同机子上，而是要根据hash 后的值将它们映射到不同的机子上处理，让不同的机器处理一个数值范围。   
  
而外排序的方法会消耗大量的IO，效率不会很高。而上面的分布式方法，也可以用于单机版本，也就是将总的数据根据值的范围，划分成多个不同的子文件，然后逐个处理。处理完毕之后再对这些单词的及其出现频率进行一个归并。实际上就可以利用一个外排序的归并过程。   
  
另外还可以考虑近似计算，也就是我们可以通过结合自然语言属性，只将那些真正实际中出现最多的那些词作为一个字典，使得这个规模可以放入内存。