# 海量数据处理

所谓海量数据，就是数据量太大，要么在短时间内无法计算出结果，要么数据太大，无法一次性装入内存。

针对时间，我们可以使用巧妙的算法搭配合适的数据结构，如bitmap/堆/trie树等

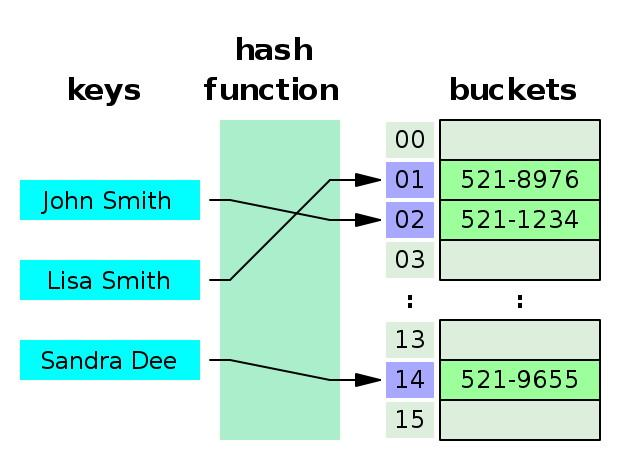
针对空间，就一个办法，大而化小，分而治之。常采用hash映射。

## Hash映射/分而治之

这里的Hash映射是指通过一种映射散列的方式，将海量数据均匀分布在对应的内存或更小的文件中。

使用hash映射有个最重要的特点是: hash值相同的两个串不一定一样，但是两个一样的字符串hash值一定相等（如果不相等会存在严重安全问题，比如两个人的账号信息经过哈希后映射到同一个值）。

具体过程：



哈希函数如下：

int hash = 0;

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

hash = (R\*hash +s.charAt(i)%M);

}

大文件映射成多个小文件。具体操作是，比如要拆分到100(M)个文件：

1、对大文件中的每条记录求hash值，然后对M取余数，即 hash(R)%M，结果为K

2、将记录R按结果K分配到第K个文件，从而完成数据拆分

这样，两条相同的记录肯定会被分配到同一个文件。

## Bitmap

也就是用1个(或几个)bit位来标记某个元素对应的value(如果是1bitmap，就只能是元素是否存在;如果是x-bitmap,还可以是元素出现的次数等信息)。使用bit位来存储信息，在需要的存储空间方面可以大大节省。应用场景有：

1、排序（如果是1-bitmap,就只能对无重复的数排序）

2、判断某个元素是否存在

比如，某文件中有若干8位数字的电话号码，要求统计一共有多少个不同的电话号码？

分析：8位最多99 999 999, 如果1Byte表示1个号码是否存在，需要95MB空间，但是如果1bit表示1个号码是否存在，则只需要 95/8=12MB 的空间。这时，数字k(0~99 999 999)与bit位的对应关系是：

#define SIZE 15\*1024\*1024

char a[SIZE];

memset(a,0,SIZE);

// a[k/8]这个字节中的 `k%8` 位命中,置为1

// 这里要注意 big-endian 和 little-endian的问题 ，假设这里是big-endian

a[k/8] = a[k/8] | (0x01 << (k%8))

## Bloom filter(布隆过滤器)

Bloom Filter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。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。

### 特点

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。

### 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：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: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: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。

**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:k=(ln2)\*(m/n), 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

### 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

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）

### 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

## 双层桶划分

双层桶不是一种数据结构，只是一种算法思维。分而治之思想。

当我们有一大推数据需要处理时，局限于各种资源限制(主要说内存)不能一次处理完成，这是需要将一大堆数据分成多个小段数据。通过处理各个小段数据完成最终任务。

双层这里是虚指，并不是一定把数据分成2份，也可能多份。比如下面几个问题：

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。

2、5亿个int找它们的中位数

第一个问题，2.5亿(2^32=4,294,967,296)个数,我们将这2^32个数分到2^8=256个区域(文件中)。每个文件中的平均数字个数差不多 2^24个(1千7百万个)。 02^24 第一个文件，2^242^25第二个文件

假设32位机，装下这些数字需要的内存是 2^24\*4=2^26=64MB,也可以不用将文件一次性读入内存而是采用流式读取。

然后对每个文件使用bitmap处理，每2bit(2-bitmap)表示一个整数，00表示整数未出现，01表示出现一次，10表示出现两次及其以上。这样，每个文件2^24个数字，最大数2^32/(8/2)=2^30=1GB内存

这个问题倒是更新是bitmap的应用，没有很好体现双层桶分治的优势。

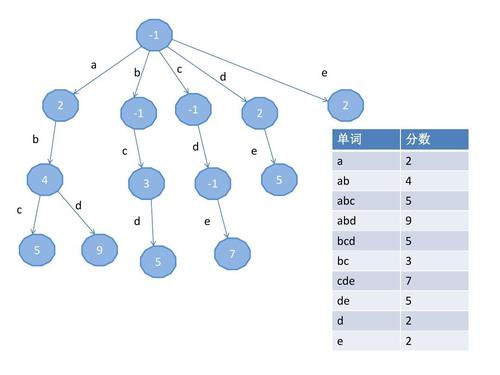
第二个问题，首先我们将int划分为2^16个区域，然后读取数据统计落到各个区域里的数的个数，之后我们根据统计结果就可以判断中位数落到那个区域，同时知道这个区域中的第几大数刚好是中位数。然后第二次扫描我们只统计落在这个区域中的那些数就可以了。

适用问题领域是：top-k，中位数，不重复或重复的数字

## Trie树

 Trie，又叫前缀树，字典树等等。它有很多变种，如后缀树，Radix Tree/Trie，PATRICIA tree，以及bitwise版本的crit-bit tree。当然很多名字的意义其实有交叉。

与二叉查找树不同，键不是直接保存在节点中，而是由节点在树中的位置决定。一个节点的所有子孙都有相同的前缀，也就是这个节点对应的字符串，而根节点对应空字符串。一般情况下，不是所有的节点都有对应的值，只有叶子节点和部分内部节点所对应的键才有相关的值。



trie中的键通常是字符串，但也可以是其它的结构。trie的算法可以很容易地修改为处理其它结构的有序序列，比如一串数字或者形状的排列。比如，bitwise trie中的键是一串位元，可以用于表示整数或者内存地址。

## 数据库索引

索引使用的数据结构多是B树或B+树。

B树和B+树广泛应用于文件存储系统和数据库系统中，mysql使用的是B+树，oracle使用的是B树，Mysql也支持多种索引类型，如b-tree 索引，哈希索引，全文索引等。

一般来说，索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储的磁盘上。索引查找过程中就要产生磁盘I/O消耗，相对于内存存取，I/O存取的消耗要高几个数量级，所以评价一个数据结构作为索引的优劣最重要的指标就是在查找过程中磁盘I/O操作次数的渐进复杂度。换句话说，索引的结构组织要尽量减少查找过程中磁盘I/O的存取次数。

### 磁盘数据查找过程

盘面：每一个盘片都有2个上下盘面，每个盘面都可以存储数据

柱面：所有盘面上的同一磁道构成一个圆柱，叫做柱面。磁盘读写按柱面进行; 只在同一柱面所有的磁头全部读/写完毕后磁头才转移到下一柱面，因为选取磁头只需通过电子切换即可，而选取柱面则必须通过机械切换。电子切换相当快，比在机械上磁头向邻近磁道移动快得多，所以，数据的读/写按柱面进行，而不按盘面进行。也就是说，一个磁道写满数据后，就在同一柱面的下一个盘面来写，一个柱面写满后，才移到下一个扇区开始写数据。读数据也按照这种方式进行，这样就提高了硬盘的读/写效率。

磁道：磁盘在格式化时被划分出许多同心圆，这些同心圆轨迹叫做磁道 track。磁道从外向内从0开始编号。

扇区：信息以脉冲串的形式记录在这些轨迹中，这些同心圆不是连续记录数据，而是被划分成一段段圆弧，每段圆弧叫做一个扇区，扇区从“1”开始编号 。扇区也叫块号。

磁盘在物理上划分为柱面, 磁道，扇区。想要读取扇区的数据，需要将磁头放到这个扇区上方:

1、先找到柱面，也就是寻道。磁头是不能动的，但可以沿着磁盘半径方向运动，耗时记为寻道事件 t(seek)

2、将目标扇区旋转到磁头下，这个过程耗时是旋转时间t(r)

一个磁盘扇区数据读取的时间t = t(seek)+t(r)+t(数据传输) , 在数据库查找数据时，查找时间与访问的磁盘盘块成正比，内存处理时间可以忽略不计。

### B树

2-3树：一个节点最多有2个key，红黑树就是2-3树的一种实现。

B树又叫多路平衡查找树。B树可以看做是对2-3树的扩展，允许每个节点有M-1个key，并以升序排列，这里的M就是B树的阶。

B树的度d(d>=2) ，有一些特征：

1、根节点至少有2个子节点

2、所有的叶节点具有相同的深度 h，也就是树高

3、每个叶子节点至少包含一个key和2个指针，最多2d-1个key和2d个指针，叶节点的指针都是null。每个节点的关键字个数在【d-1,2d-1】之间

4、每个非叶子节点，key和指针互相间隔，节点两端是指针，因此节点中指针个数=key的个数+1

5、每个指针要么是null，要么指向另一个节点

如果某个指针在节点node最左边且不为null，则其指向节点的所有key小于v(key1)，其中v(key1)为node的第一个key的值。 如果某个指针在节点node最右边且不为null，则其指向节点的所有key大于v(keym)，其中v(keym)为node的最后一个key的值。 如果某个指针在节点node的左右相邻key分别是keyi和keyi+1且不为null，则其指向节点的所有key小于v(keyi+1)且大于v(keyi)。

使用数据结构表示如下：

typedef struct Item{

int key;

Data data;

}

#define m 3 //B树的阶

typedef struct BTNode{

int degree; //B树的度

int keynums; //每个节点key的个数

Item items[m];

struct BTNode \*p[m];

}BTNode,\* BTree;

typedef struct{

BTNode \*pt; //指向找到的节点

int i; // 节点中关键字的序号 (0,m-1)

int tag; //1:查找成功，0：查找失败

}Result;

Status btree\_insert(root,target);

Status btree\_delete(root,target);

Result btree\_find(root,target);

### 建立索引

当为一张空表创建索引时，数据库系统将为你分配一个索引页，该索引页在你插入数据前一直是空的。此页此时既是根结点，也是叶结点。每当你往表中插入一行数据，数据库系统即向此根结点中插入一行索引记录。

插入和删除新的数据记录都会破坏B-Tree的性质，因此在插入删除时，需要对树进行一个分裂、合并、转移等操作以保持B-Tree性质

### 查找操作

从root节点出发，对每个节点，找到等于target的key，则查找成功；或者找到大于target的最小k[i], 到 k[i] 左指针指向的子节点继续查找，直到页节点，如果找不到，说明关键字target不在B树中。

分析下时间复杂度：

对于一个度为d的B-Tree,每个节点的索引key个数是d-1, 索引key个数为N，树高h上限是：

2d^h-1=N ==> h=logd^((N+1) /2) ？？？

因此，检索一个key，查找节点的个数的复杂度是O(logd^N)

比如d=2，N=1,000,000 (1百万)，h差不多20个

d=3,N=1,000,000 (1百万) ,h差不多13个(3^11=1,594,323)

d=4,N=1,000,000 (1百万) ,h差不多10个

d=5,N=1,000,000 (1百万) ,h差不多9个 (5^9 = 1,953,125)

d=6,N=1,000,000 (1百万) ,h差不多8个(6^8 = 1,679,616)

d=7,N=1,000,000 (1百万) ,h差不多8个

d=8,N=1,000,000 (1百万) ,h差不多7个

d=9,N=1,000,000 (1百万) ,h差不多7个

d=10,N=1,000,000 (1百万) ,h差不多6个

d=100时，h差不多3个

数据库系统在设计时，通常将一个节点的大小设为一个页大小(通常4k)，这样保证一个节点在物理上也存储在一个页里，加上计算机存储分配都是按页对其，这样保证一个节点只需要一次I/O.

实际应用中，d都是比较大，通常超过100，因此1百万的数据通常最多访问3个节点，也就是3次I/O, 因此使用B树作为索引结构查询效率非常高。

### 插入数据

插入数据时，需要更新索引，索引中也要添加一条记录。索引中添加一条记录的过程是：

沿着搜索的路径从root一直到叶节点

每个节点的关键字个数在【d-1,2d-1】之间，当节点的关键字个数是2t-1时，再加入target就违反了B树定义，需要对该节点进行分裂：已中间节点为界，分成2个包含d-1个关键字的子节点（另外还有一个分界关键字，2\*(d-1)+1=2d-1），同时把该分界关键字提升到该叶子的父节点中，如果这导致父节点关键字个数超过2d-1,就继续向上分裂，直到根节点。

如下演示动画，往度d=2的B树中插入： 6 10 4 14 5 11 15 3 2 12 1 7 8 8 6 3 6 21 5 15 15 6 32 23 45 65 7 8 6 5 4

### B树和B+树的区别

B树和B+树的区别在于：

B+树的非叶子节点只包含导航信息，不包含实际记录的信息，这可以保证一个固定大小节点可以放入更多个关键字，也就是更大的度d，从而树高h可以更小，从而相比B树有更优秀的查询效率

所有的叶子节点和相邻的节点使用链表方式相连，便于区间查找和遍历

## 倒排索引(Inverted Index)

### 概述

也叫反向索引。是文档检索系统中最常用的数据结构。

倒排索引的索引表中的每一项都包含一个属性值和具有该属性值的各记录的地址。因为不是由记录来确定属性值，而是由属性来确定记录，因而成为倒排索引（inverted index）。

带有倒排索引的文件成为倒排索引文件，简称为倒排文件（inverted file）。

常规的索引是文档到关键词的映射，如果对应的文档是Elasticsearch就是使用倒排索引(inverted index)的结构来做快速的全文搜索。ElasticSearch 不仅用于全文搜索, 还有非常强大的统计功能 (facets)。

携程，58，美团的分享中都提到ES构建实时日志系统，帮助定位系统问题。

**参考：**

Elasticsearch权威指南

### 倒排列表

倒排列表记录了某个单词位于哪些文档中。在给定的文档语料中，一般会有多个文档包含某单词，每个文档有唯一的编号（DocID），单词在这个文档中出现的次数（TF）及单词在文档中哪些位置出现等信息，与一个文档相关的信息被称做倒排索引项（Posting），包含这个单词的一系列倒排索引项形成了列表结构，这就是某个单词对应的倒排列表。





### 更新策略

**完全重建策略：**当新增文件达到一定数量，将新增文档和原先的老文档整合，然后利用静态索引创建方法对所有文档重建索引，新索引建立完成后老索引会被遗弃。此代价非常高，但是主流商业搜索引擎一般采用采用方法来维护索引的更新。

**再合并策略：**当新增文档进入系统，解析文档，之后更新内存中维护的临时索引，文档中出现的每个单词，在其倒排序列表末尾追加倒排表列表项；一旦临时索引将指定内存消耗光，即进行一次索引合并，这里需要倒排文件里的倒排列表存放顺序已经按照索引单词字典顺序由低到高排序，这样直接顺序扫描合并即可。其缺点是：因为要生成新的倒排索引文件，所以对老索引中的很多单词，尽管其在倒排列表并未发生任何变化，也需要将其从老索引中取出并写入新索引中，这样对磁盘消耗是没必要的。

**原地更新策略：**试图改进再合并策略，在原地合并倒排表，这需要提前分配一定的空间给未来插入，如果提前分配的空间不够了需要迁移。实际显示，其索引更新的效率比再合并策略要低。

**混合策略：**出发点是能够结合不同索引更新策略的长处，将不同索引更新策略混合，以形成更高效的方法。

### 应用

倒排索引的应用：对simhash的分块处理

1. 把64位的二进制simhash签名均分成4块，每块16位。根据抽屉原理，如果两个签名的海明距离在3以内，它们必有一块完全相同。
2. 然后把分成的4块中的每一块分别作为前16位来进行查找，建立倒排索引；

跳跃链表、跳跃表、跳表

GIS中的POI（Point of Interest）查询

## 外排序

### 概述

对磁盘文件的排序。将待处理的数据不能一次装入内存，先读入部分数据排序后输出到临时文件，采用「排序-归并」的策略。在归并阶段将这些临时文件组合为一个大的有序文件，也即排序结果。

### 应用

多路归并,最小堆

比如，要对900 MB的数据进行排序，但机器上只有100 MB的可用内存时，外归并排序按如下方法操作：

1、读入100 MB的数据至内存中，用某种常规方式（如快速排序、堆排序、归并排序等方法）在内存中完成排序。

2、将排序完成的数据写入磁盘。

3、重复步骤1和2直到所有的数据都存入了不同的100 MB的块（临时文件）中。在这个例子中，有900 MB数据，单个临时文件大小为100 MB，所以会产生9个临时文件。

4、读入每个临时文件（顺串）的前10 MB（ = 100 MB / (9块 + 1)）的数据放入内存中的输入缓冲区，最后的10 MB作为输出缓冲区。（实践中，将输入缓冲适当调小，而适当增大输出缓冲区能获得更好的效果。）

5、执行九路归并算法，将结果输出到输出缓冲区。一旦输出缓冲区满，将缓冲区中的数据写出至目标文件，清空缓冲区。一旦9个输入缓冲区中的一个变空，就从这个缓冲区关联的文件，读入下一个10M数据，除非这个文件已读完。这是“外归并排序”能在主存外完成排序的关键步骤 -- 因为“归并算法”(merge algorithm)对每一个大块只是顺序地做一轮访问(进行归并)，每个大块不用完全载入主存。

为了增加每一个有序的临时文件的长度，可以采用置换选择排序（Replacement selection sorting）。它可以产生大于内存大小的顺串。具体方法是在内存中使用一个最小堆进行排序，设该最小堆的大小为M。算法描述如下：

1、初始时将输入文件读入内存，建立最小堆。

2、将堆顶元素输出至输出缓冲区。然后读入下一个记录：

若该元素的关键码值不小于刚输出的关键码值，将其作为堆顶元素并调整堆，使之满足堆的性质；

否则将新元素放入堆底位置，将堆的大小减1。

3、重复第2步，直至堆大小变为0。

4、此时一个顺串已经产生。将堆中的所有元素建堆，开始生成下一个顺串。

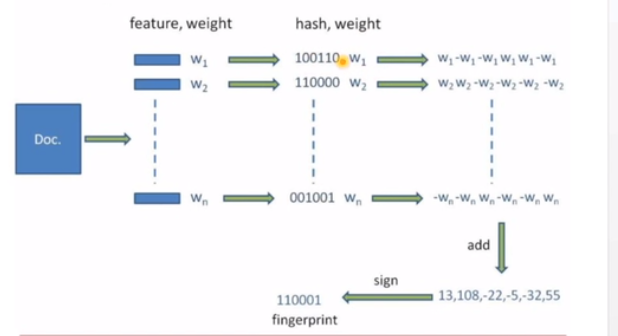
此方法能生成平均长度为2M的顺串，可以进一步减少访问外部存储器的次数，节约时间，提高算法效率。

## simhash算法

### 概述

问题的起源：设计比较两篇文章相似度的算法。

### 算法



simHash算法分为5个步骤：

1. **分词**：对待考察文档进行分词，把得到的分词称为特征，然后为每一个特征设置N等级别的权重。如给定一段语句：“CSDN博客结构值法算法之道的作者July”，分词后为：“CSDN 博 客 结 构 之 法 算 法 之 道 的 作 者 July”，然后为每个特征向量赋予权值：CSDN(4)博客(5)结构(3)之(1)法(2)算法(3)之(1)道(2)的(1)作者(5)July(5)，权重代表了这个特征在整条语句中的重要程度。
2. **Hash**：通过hash算法计算各个特征向量的hash值，hash值为二进制数组成的n位签名。
3. **加权**：W=Hash\*weight，W(CSDN)=100101\*4=4-4-44-44，W(博客)=101011\*5=5-55-555。
4. **合并**：将上述每个特征的加权结果累加，变成一个序列串。如“4+5，-4+-5，-4+5,4+-5，-4+5,4+5”，得到“9,-9,1,-1,1”。
5. **降维**：对于n位签名的累加结果，如果大于0则置1，否则置0，从而得到该语句的simhash值，最后我们便可以根据不同语句simhash的海明举例来判断它们的相似度。例如把上面计算出来的“9,-9,1,-1,1,9”降维，得到“101011”，从而形成它们的simhash签名。

### 分词权重

分词的权值计算：

词频-逆文档频率，TF-IDF（term frequency-inverse document frequency）是一种用于资讯检索与资讯探勘的常用加权技术。TD-IDF是一种统计方法，用以评估一字词对于一个文件集或者一个语料库中的其中一份文件的重要程度。

字词的重要性随着它在文件中出现的次数成正比增加，但同时会随着它在语料库中出现的频率成反比下降：Weight=TF\*IDF。

如果某个分词在一篇文章中出现的频率TF高，并且在其他文章中很少出现，则认为此词或者短语具有很好的类别区分能力，适合用来分类。

事实上，该技术在自然语言处理用途广泛，可以配合其他方法一起使用，如余弦距离（反比与相似度），LDA主体模型，用于聚类、标签传递算法等后续分析中。

### 应用

每篇文档得到simHash签名值后，接着计算两个签名的海明距离即可。根据经验值，对64位的simhash值，海明距离在3以内的可认为相似度比较高。

海明距离的求法：两个二进制异或值中1的个数，即两个二进制数位数不同的个数。

## 跳跃链表

### 概述

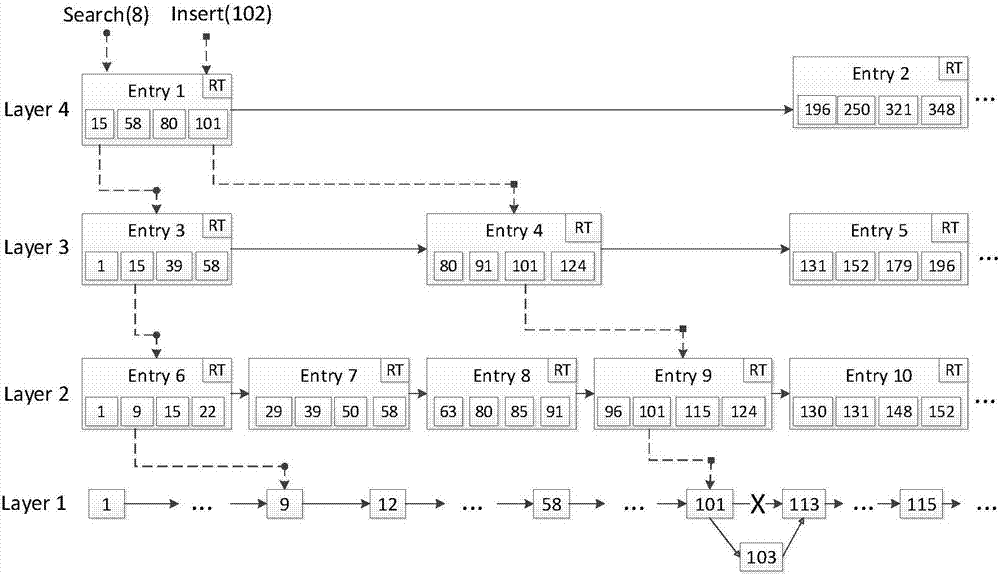
跳跃链表是一种随机化数据结构，基于并联的链表，其效率与RBTree相当。具有简单、高效、动态的特点。

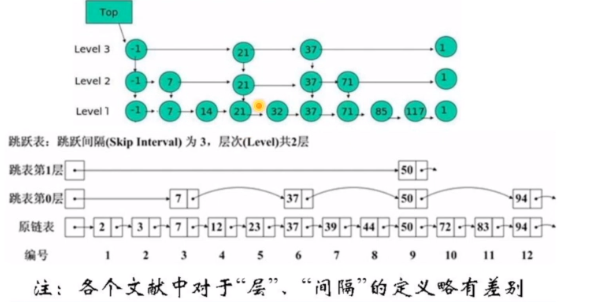
跳跃链表对有序的链表附加辅助结构，在链表中的查找可以快速的跳过部分结点（因此得名）。

查找、增加、删除的期望时间都是O(logN)。

跳跃链表在并行计算中非常有用，数据插入可以在跳表的不同部分并行进行，而不用全局的数据结构重新平衡。

跳跃列表是按层建造的。底层是一个普通的有序链表。每个更高层都充当下面列表的“快速跑道”，这里在层i中的元素按某个固定的概率p出现在层i+1中。平均起来，每个元素都在1/(1-p)个列表中出现。





### 复杂度

## MD5算法

## Map-Reduce

MapReduce是Google提出的一个软件架构，用于大规模数据集（大于1TB）的并行运算。

概念“Map（映射）”和“Reduce（归纳）”，及他们的主要思想，都是从函数式编程语言借来的，还有从矢量编程语言借来的特性。MapReduce的伟大之处就在于让不熟悉并行编程的程序员也能充分发挥分布式系统的威力。

### 原理

举一个例子：10年内所有论文(当然有很多很多篇)里面出现最多的几个单词。

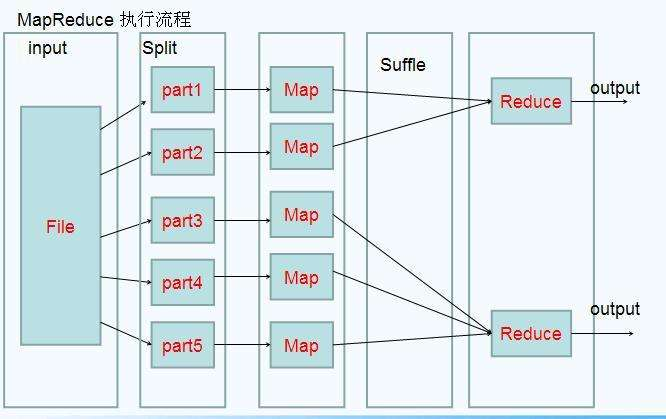
我们把论文集分层N份，一台机器跑一个作业。这个方法跑得快，但是有部署成本，需要把程序copy到别的机器，要把论文分N份，且还需要最后把N个运行结果整合起来。这其实就是Mapreduce本质。

map函数和reduce函数是交给用户实现的，这两个函数定义了任务本身。

map函数：接受一个键值对（key-value pair），产生一组中间键值对。MapReduce框架会将map函数产生的中间键值对里键相同的值传递给一个reduce函数。Map操作是可以高度并行的。

reduce函数：接受一个键，以及相关的一组值，将这组值进行合并产生一组规模更小的值（通常只有一个或零个值）。

注：Map阶段：把大任务分成子任务，Reduce任务：子任务并发处理，然后合并结果。



**注意点：**

1. 备份的考虑，分布式存储的设计细节，以及容灾策略（分布式文件非常大）；
2. 任务分配策略与任务进度跟踪的细节设计，节点状态的呈现；
3. 多用户权限的控制。

### 应用

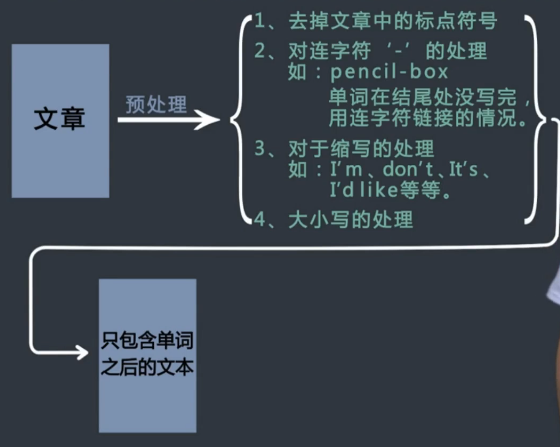
常见海量处理题目解题关键：

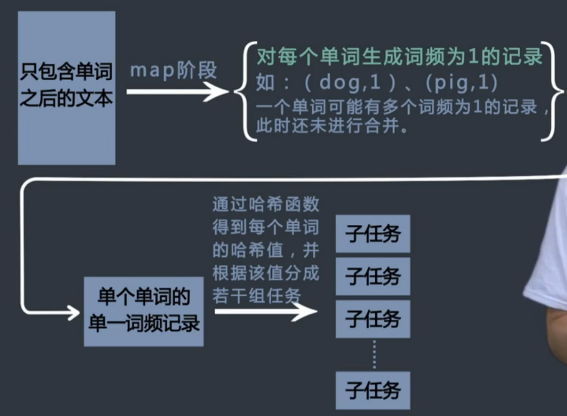
1. 分而治之，通过哈希函数将大任务分流到机器，或分流成小文件；
2. 常用的HashMap或bitmap。

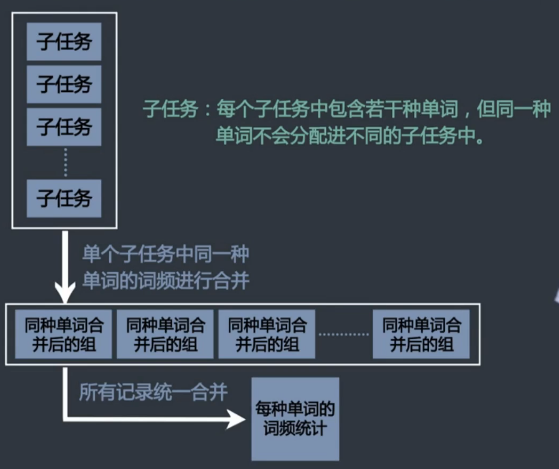
难点：通讯、时间和空间的估算。

#### 词频统计

**题目：**用Map-Reduce方法统计一篇文章中每个单词出现的个数。

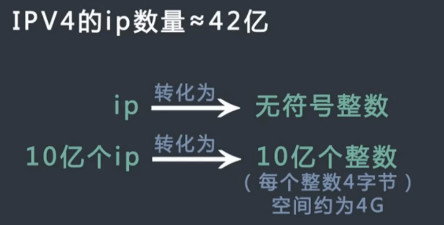


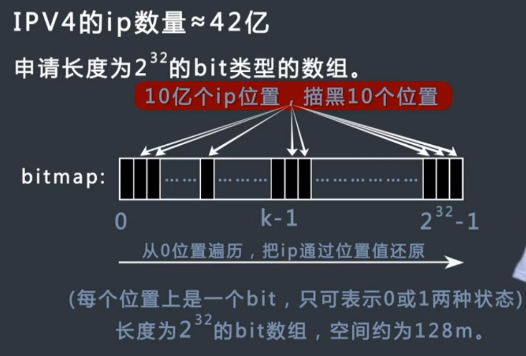




#### IPV4地址排序

**题目：**请对10亿个IPV4的IP地址进行排序，每个IP只会出现一次。





注：其本质就是将10亿个IP地址映射到232大小的数组上（因为IP地址的格式是127.0.0.1，即4\*8=32位即可表示一个IP地址），并且通过标志位来标识该IP是否出现。

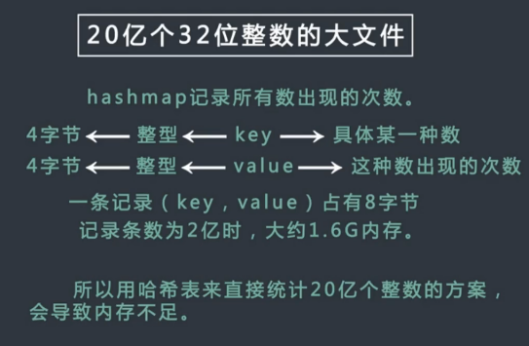
#### 10亿人年龄排序

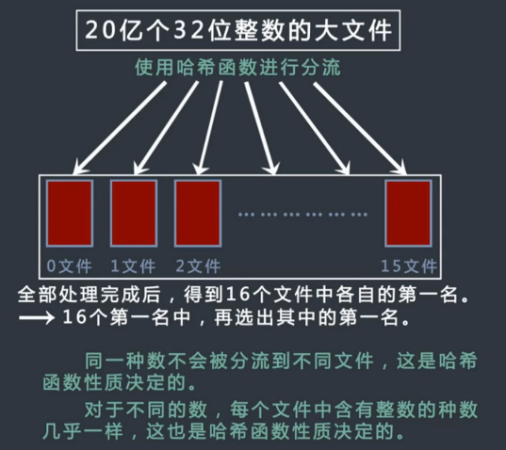
请对10亿人的年龄进行排序。



#### 出现次数最多的数

**题目：**有一个包含20亿个全是32位整数的大文件，在其中找到出现次数最多的数，但是内存限制只有2G。



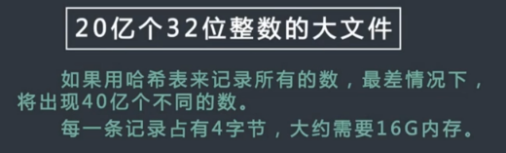


#### 未出现的数字

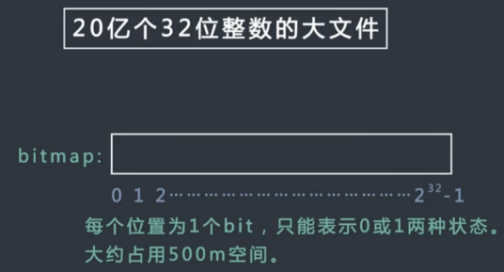
**题目：**32位无符号整数的范围是0~4294967295。现在有一个正好包含40亿个无符号整数的文件，所以在整个范围中必然有没出现过的数。可以使用最多10M的内存，只用找到一个没出现过的数即可，该如何找？

分析：

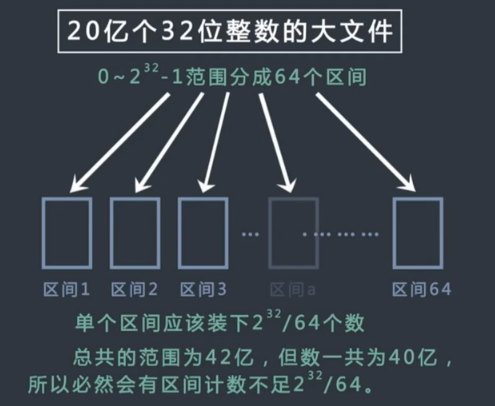
使用直接存储的方法：

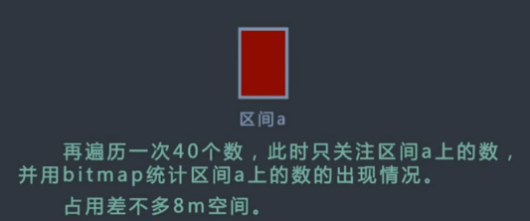


使用bitmap进行存储：



利用哈希函数分流：



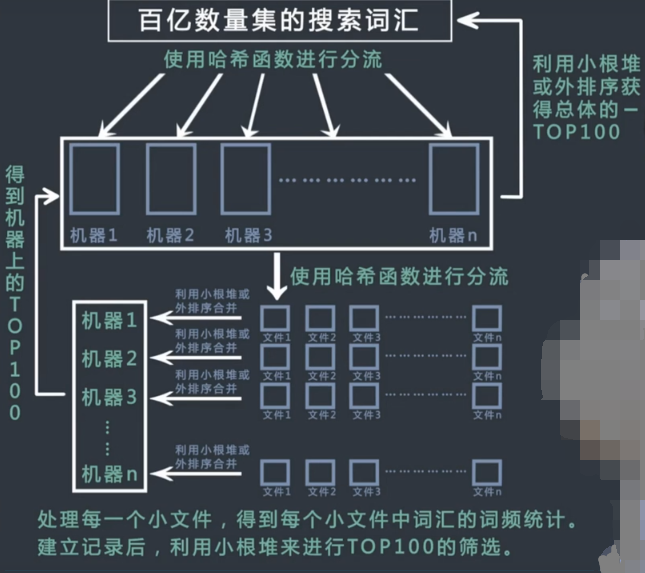


总结：

1. 根据内存限制决定区间大小，根据区间大小，得到有多少个变量，来记录每个区间的数出现的次数；
2. 统计区间上的数的出现次数，找到不足的区间；
3. 利用bitmap对不满的区间，进行这个区间上的数的词频统计。

#### 搜索最热100词

**题目：**某搜索公司一天的用户搜索词汇是海量的，假设有百亿的数据量，请设计一种求出每天最热100词的可行方法。

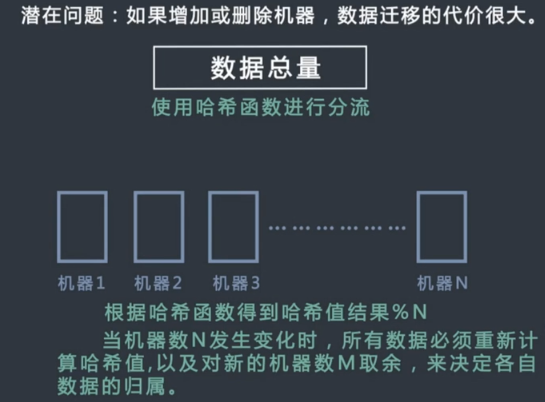


#### 一致性哈希算法

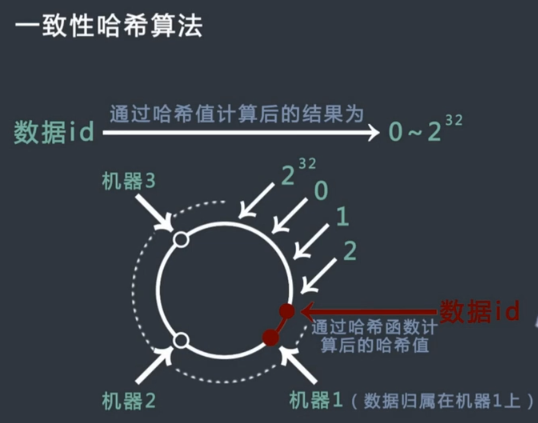
**题目：**工程师常使用服务器集群来设计和实现数据缓存，以下是常见的策略：

1. 无论是添加、查询还是删除数据，都先将数据的di通过哈希函数转换为一个哈希值，记为key；
2. 如果目前机器有N台，则计算key%N的值，这个值就是该数据所属的机器编号，无论是添加、删除还是查询操作，都只在这台机器上进行。

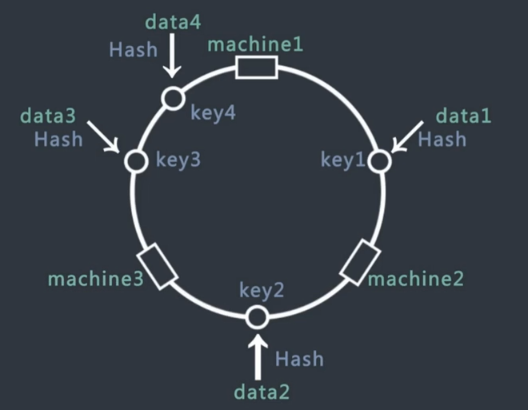
请分析这种缓存策略可能带来的问题，并提出改进的方案。



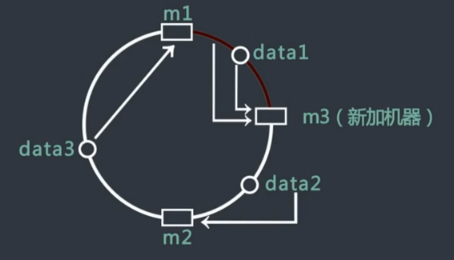
假设数据计算哈希值后的范围为0~232



注：如何确定数据是在哪个机器上，首先数据计算哈希值后找到在环中的对应位置，然后顺时针查找与该位置最近的机器，则数据就定位在这个机器上。

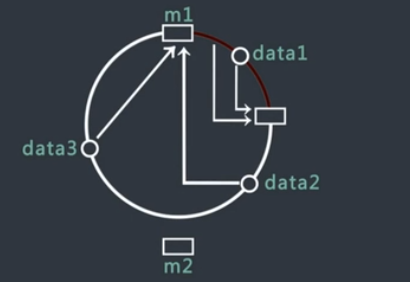


假如有新的机器加入：



假设新机器计算哈希值后位于m1和m2之间的位置，原来这段位置是归m2管理的，现在则data1部分需要归属于m3管理，data2仍然归属m2管理，则需要将旧数据data1从m2迁移到m3上，这样调整代价是相对比较小的。

假如需要删除机器：



注：删除机器时只需要将待删除机器归属的数据移动到下一台机器上即可，即将上图原来归属于m2的数据data2转移到m1上。

### Hadoop

谷歌技术有"三宝"，GFS、MapReduce和大表（BigTable）。

Hadoop实际上就是谷歌三宝的开源实现，Hadoop MapReduce对应Google MapReduce，HBase对应BigTable，HDFS对应GFS。

HDFS（或GFS）为上层提供高效的非结构化存储服务，HBase（或BigTable）是提供结构化数据服务的分布式数据库，Hadoop MapReduce（或Google MapReduce）是一种并行计算的编程模型，用于作业调度。

Hadoop 使用java实现。

# 估算

在处理海量问题之前，我们往往要先估算下数据量，能否一次性载入内存？如果不能，应该用什么方式拆分成小块以后映射进内存？每次拆分的大小多少合适？以及在不同方案下，大概需要的内存空间和计算时间。

比如,我们来了解下以下常见问题时间 和 空间 估算 :

8位的电话号码，最多有99 999 999个

IP地址

1G内存，2^32 ,差不多40亿，40亿Byte\*8 = 320亿 bit

海量处理问题常用的分析解决问题的思路是：

分而治之/Hash映射 + hash统计/trie树/红黑树/二叉搜索树 + 堆排序/快速排序/归并排序

双层桶划分

Bloom filter 、Bitmap

Trie树/数据库/倒排索引

外排序

分布处理之 Hadoop/Mapreduce