教你如何迅速秒杀掉：99%的海量数据处理面试题

本文经过大量细致的优化后，收录于我的新书《编程之法》第六章中，新书目前已上架京东/当当/亚马逊

作者：July

出处：结构之法算法之道 blog

前言

一般而言，标题含有“秒杀”，“99%”，“史上最全/最强”等词汇的往往都脱不了哗众取宠之嫌，但进一步来讲，如果读者读罢此文，却无任何收获，那么，我也甘愿背负这样的罪名 :-)，同时，此文可以看做是对这篇文章：十道海量数据处理面试题与十个方法大总结的一般抽象性总结。

毕竟受文章和理论之限，本文将摒弃绝大部分的细节，只谈方法/模式论，且注重用最通俗最直白的语言阐述相关问题。最后，有一点必须强调的是，全文行文是基于面试题的分析基础之上的，具体实践过程中，还是得具体情况具体分析，且各个场景下需要考虑的细节也远比本文所描述的任何一种解决方法复杂得多。

OK，若有任何问题，欢迎随时不吝赐教。谢谢。

何谓海量数据处理？

所谓海量数据处理，无非就是基于海量数据上的存储、处理、操作。何谓海量，就是数据量太大，所以导致要么是无法在较短时间内迅速解决，要么是数据太大，导致无法一次性装入内存。

那解决办法呢?针对时间，我们可以采用巧妙的算法搭配合适的数据结构，如Bloom filter/Hash/bit-map/堆/数据库或倒排索引/trie 树，针对空间，无非就一个办法：大而化小，分而治之（hash 映射），你不是说规模太大嘛，那简单啊，就把规模大化为规模小的，各个击破不就完了嘛。

至于所谓的单机及集群问题，通俗点来讲，单机就是处理装载数据的机器有限(只要考虑 cpu，内存，硬盘的数据交互)，而集群，机器有多辆，适合分布式处理，并行计算(更多考虑节点和节点间的数据交互)。

再者，通过本 blog 内的有关海量数据处理的文章：Big Data Processing，我们已经大致知道，处理海量数据问题，无非就是：

1. 分而治之/hash 映射 + hash 统计 + 堆/快速/归并排序；
2. 双层桶划分
3. Bloom filter/Bitmap；
4. Trie 树/数据库/倒排索引；
5. 外排序；
6. 分布式处理之 Hadoop/Mapreduce。

下面，本文第一部分、从 set/map 谈到 hashtable/hash_map/hash_set，简要介绍下 set/map/multiset/multimap，及 hash_set/hash_map/hash_multiset/hash_multimap 之区别(万丈高楼平地起，基础最重要)，而本文第二部分，则针对上述那 6 种方法模式结合对应的海量数据处理面试题分别具体阐述。

第一部分、从 set/map 谈到 hashtable/hash_map/hash_set

稍后本文第二部分中将多次提到 hash_map/hash_set，下面稍稍介绍下这些容器，以作为基础准备。一般来说，STL 容器分两种，

- 序列式容器(vector/list/deque/stack/queue/heap)，
- 关联式容器。关联式容器又分为 set(集合)和 map(映射表)两大类，以及这两大类的衍生体 multiset(多键集合)和 multimap(多键映射表)，这些容器均以 RB-tree 完成。此外，还有第 3 类关联式容器，如 hashtable(散列表)，以及以 hashtable 为底层机制完成的 hash_set(散列集合)/hash_map(散列映射表)/hash_multiset(散列多键集合)/hash_multimap(散列多键映射表)。也就是说，

  set/map/multiset/multimap 都内含一个 RB-tree，而 hash_set/hash_map/hash_multiset/hash_multimap 都内含一个 hashtable。

所谓关联式容器，类似关联式数据库，每笔数据或每个元素都有一个键值(key)和一个实值(value)，即所谓的 Key-Value(键-值对)。当元素被插入到关联式容器中时，容器内部结构(RB-tree/hashtable)便依照其键值大小，以某种特定规则将这个元素放置于适当位置。

包括在非关联式数据库中，比如，在 MongoDB 内，文档(document)是最基本的数据组织形式，每个文档也是以 Key-Value（键-值对）的方式组织起来。一个文档可以有多个 Key-Value 组合，每个 Value 可以是不同的类型，比如 String、Integer、List 等等。

```
{ "name" : "July",
  "sex" : "male",
    "age" : 23 }
```

set/map/multiset/multimap

set，同 map 一样，所有元素都会根据元素的键值自动被排序，因为 set/map 两者的所有各种操作，都只是转而调用 RB-tree 的操作行为，不过，值得注意的

是，两者都不允许两个元素有相同的键值。

不同的是：set 的元素不像 map 那样可以同时拥有实值(value)和键值(key)，set 元素的键值就是实值，实值就是键值，而 map 的所有元素都是 pair，同时拥有实值(value)和键值(key)，pair 的第一个元素被视为键值，第二个元素被视为实值。

至于 multiset/multimap，他们的特性及用法和 set/map 完全相同，唯一的差别就在于它们允许键值重复，即所有的插入操作基于 RB-tree 的 insert_equal() 而非 insert_unique()。

hash_set/hash_map/hash_multiset/hash_multimap

hash_set/hash_map，两者的一切操作都是基于 hashtable 之上。不同的是，hash_set 同 set 一样，同时拥有实值和键值，且实质就是键值，键值就是实值，而 hash_map 同 map 一样，每一个元素同时拥有一个实值(value)和一个键值(key)，所以其使用方式，和上面的 map 基本相同。但由于 hash_set/hash_map 都是基于 hashtable 之上，所以不具备自动排序功能。为什么?因为 hashtable 没有自动排序功能。

至于 hash_multiset/hash_multimap 的特性与上面的 multiset/multimap 完全相同，唯一的差别就是它们 hash_multiset/hash_multimap 的底层实现机制是 hashtable（而 multiset/multimap，上面说了，底层实现机制是 RB-tree），所以它们的元素都不会被自动排序，不过也都允许键值重复。

所以，综上，说白了，什么样的结构决定其什么样的性质，因为 set/map/multiset/multimap 都是基于 RB-tree 之上，所以有自动排序功能，而 hash_set/hash_map/hash_multiset/hash_multimap 都是基于 hashtable 之上，所以不含有自动排序功能，至于加个前缀 multi_无非就是允许键值重复而已。如下图所示：

此外，

- 关于什么 hash，请看 blog 内此篇文章；
- 关于红黑树，请参看 blog 内系列文章，
- 关于 hash_map 的具体应用：请看这里，关于 hash_set：请看此文。

OK，接下来，请看本文第二部分、处理海量数据问题之六把密匙。

第二部分、处理海量数据问题之六把密匙

密匙一、分而治之/Hash 映射 + Hash_map 统计 + 堆/快速/归并排序

1、海量日志数据，提取出某日访问百度次数最多的那个 IP。

既然是海量数据处理，那么可想而知，给我们的数据那就一定是海量的。针对这个数据的海量，我们如何着手呢?对的，无非就是分而治之/hash 映射 + hash 统计 + 堆/快速/归并排序，说白了，就是先映射，而后统计，最后排序：

1. 分而治之/hash 映射：针对数据太大，内存受限，只能是：把大文件化成(取模映射)小文件，即 16 字方针：大而化小，各个击破，缩小规模，逐个解决
2. hash_map 统计：当大文件转化了小文件，那么我们便可以采用常规的 hash_map(ip，value)来进行频率统计。
3. 堆/快速排序：统计完了之后，便进行排序(可采取堆排序)，得到次数最多的 IP。

具体而论，则是：“首先是这一天，并且是访问百度的日志中的 IP 取出来，逐个写入到一个大文件中。注意到 IP 是 32 位的，最多有个 2^32 个 IP。同样可以采用映射的方法，比如%1000，把整个大文件映射为 1000 个小文件，再找出每个小文中出现频率最大的 IP（可以采用 hash_map 对那 1000 个文件中的所有 IP 进行频率统计，然后依次找出各个文件中频率最大的那个 IP）及相应的频率。然后再在这 1000 个最大的 IP 中，找出那个频率最大的 IP，即为所求。”--十道海量数据处理面试题与十个方法大总结。

关于本题，还有几个问题，如下：

1、Hash 取模是一种等价映射，不会存在同一个元素分散到不同小文件中的情况，即这里采用的是 mod1000 算法，那么相同的 IP 在 hash 取模后，只可能落在同一个文件中，不可能被分散的。因为如果两个 IP 相等，那么经过 Hash(IP)之后的哈希值是相同的，将此哈希值取模（如模 1000），必定仍然相等。

2、那到底什么是 hash 映射呢？简单来说，就是为了便于计算机在有限的内

存中处理 big 数据，从而通过一种映射散列的方式让数据均匀分布在对应的内存位置(如大数据通过取余的方式映射成小树存放在内存中，或大文件映射成多个小文件)，而这个映射散列方式便是我们通常所说的 hash 函数，设计的好的 hash 函数能让数据均匀分布而减少冲突。尽管数据映射到了另外一些不同的位置，但数据还是原来的数据，只是代替和表示这些原始数据的形式发生了变化而已。

OK，有兴趣的，还可以再了解下一致性 hash 算法，见 blog 内此文第五部分：http://blog.csdn.net/v_july_v/article/details/6879101。

2、寻找热门查询，300 万个查询字符串中统计最热门的 10 个查询

原题：搜索引擎会通过日志文件把用户每次检索使用的所有检索串都记录下来，每个查询串的长度为 1-255 字节。假设目前有一千万个记录（这些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。一个查询串的重复度越高，说明查询它的用户越多，也就是越热门），请你统计最热门的 10 个查询串，要求使用的内存不能超过 1G。

解答：由上面第 1 题，我们知道，数据大则划为小的，如如一亿个 Ip 求 Top 10，可先%1000 将 ip 分到 1000 个小文件中去，并保证一种 ip 只出现在一个文件中，再对每个小文件中的 ip 进行 hashmap 计数统计并按数量排序，最后归并或者最小堆依次处理每个小文件的 top10 以得到最后的结。

但如果数据规模比较小，能一次性装入内存呢?比如这第 2 题，虽然有一千万个 Query，但是由于重复度比较高，因此事实上只有 300 万的 Query，每个 Query255Byte，因此我们可以考虑把他们都放进内存中去（300 万个字符串假设没有重复，都是最大长度，那么最多占用内存 3M*1K/4=0.75G。所以可以将所有字符串都存放在内存中进行处理），而现在只是需要一个合适的数据结构，在这里，HashTable 绝对是我们优先的选择。

所以我们放弃分而治之/hash 映射的步骤，直接上 hash 统计，然后排序。

So，针对此类典型的 TOP K 问题，采取的对策往往是：hashmap + 堆。如下所示：

1. hash_map 统计：先对这批海量数据预处理。具体方法是：维护一个 Key 为 Query 字串，Value 为该 Query 出现次数的 HashTable，即 hash_map(Query，Value)，每次读取一个 Query，如果该字串不在 Table 中，那么加入该字串，并且将 Value 值设为 1；如果该字串在 Table 中，那么将该字串的计数加一即可。最终我们在 O(N)的时间复杂度内用 Hash 表完成了统计；
2. 堆排序：第二步、借助堆这个数据结构，找出 Top K，时间复杂度为 N ‘logK。即借助堆结构，我们可以在 log 量级的时间内查找和调整/移动。因此，维护一个 K(该题目中是 10)大小的小根堆，然后遍历 300 万的 Query，分别和根元素进行对比。所以，我们最终的时间复杂度是：O（N） + N' * O（logK），（N 为 1000 万，N’ 为 300 万）。

别忘了这篇文章中所述的堆排序思路：“维护 k 个元素的最小堆，即用容量为 k 的最小堆存储最先遍历到的 k 个数，并假设它们即是最大的 k 个数，建堆费时 O（k），并调整堆(费时 O（logk）)后，有 k1>k2>...kmin（kmin 设为小顶堆中最小元素）。继续遍历数列，每次遍历一个元素 x，与堆顶元素比较，若 x>kmin，则更新堆（x 入堆，用时 logk），否则不更新堆。这样下来，总费时 O（k*logk+（n-k）*logk）=O（n*logk）。此方法得益于在堆中，查找等各项操作时间复杂度均为 logk。” --第三章续、Top K 算法问题的实现。

当然，你也可以采用 trie 树，关键字域存该查询串出现的次数，没有出现为 0。最后用 10 个元素的最小推来对出现频率进行排序。

3、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 字节，内存限制大小是 1M。返回频数最高的 100 个词。

由上面那两个例题，分而治之 + hash 统计 + 堆/快速排序这个套路，我们已经开始有了屡试不爽的感觉。下面，再拿几道再多多验证下。请看此第 3 题：又是文件很大，又是内存受限，咋办?还能怎么办呢?无非还是：

1. 分而治之/hash 映射：顺序读文件中，对于每个词 x，取 hash(x)%5000，然后按照该值存到 5000 个小文件（记为 x0,x1,...x4999）中。这样每个文件大概是 200k 左右。如果其中的有的文件超过了 1M 大小，还可以按照类似的方法继续往下分，直到分解得到的小文件的大小都不超过 1M。
2. hash_map 统计：对每个小文件，采用 trie 树/hash_map 等统计每个文件中出现的词以及相应的频率。
3. 堆/归并排序：取出出现频率最大的 100 个词（可以用含 100 个结点的最小堆）后，再把 100 个词及相应的频率存入文件，这样又得到了 5000 个文件。最后就是把这 5000 个文件进行归并（类似于归并排序）的过程了。

4、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。

如果每个数据元素只出现一次，而且只出现在某一台机器中，那么可以采取以下步骤统计出现次数 TOP10 的数据元素：

1. 堆排序：在每台电脑上求出 TOP10，可以采用包含 10 个元素的堆完成

   （TOP10 小，用最大堆，TOP10 大，用最小堆，比如求 TOP10 大，我们首先取前 10 个元素调整成最小堆，如果发现，然后扫描后面的数据，并与堆顶元素比较，如果比堆顶元素大，那么用该元素替换堆顶，然后再调整为最小堆。最后堆中的元素就是 TOP10 大）。
2. 求出每台电脑上的 TOP10 后，然后把这 100 台电脑上的 TOP10 组合起来，共 1000 个数据，再利用上面类似的方法求出 TOP10 就可以了。

但如果同一个元素重复出现在不同的电脑中呢，如下例子所述：

这个时候，你可以有两种方法：

- 遍历一遍所有数据，重新 hash 取摸，如此使得同一个元素只出现在单独的一台电脑中，然后采用上面所说的方法，统计每台电脑中各个元素的出现次数找出 TOP10，继而组合 100 台电脑上的 TOP10，找出最终的 TOP10。
- 或者，暴力求解：直接统计统计每台电脑中各个元素的出现次数，然后把同一个元素在不同机器中的出现次数相加，最终从所有数据中找出 TOP10。

5、有 10 个文件，每个文件 1G，每个文件的每一行存放的都是用户的 query，每个文件的 query 都可能重复。要求你按照 query 的频度排序。

方案 1：直接上：

1. hash 映射：顺序读取 10 个文件，按照 hash(query)%10 的结果将 query 写入到另外 10 个文件（记为 a0,a1,..a9）中。这样新生成的文件每个的大小大约也 1G（假设 hash 函数是随机的）。
2. hash_map 统计：找一台内存在 2G 左右的机器，依次对用 hash_map(query, query_count)来统计每个 query 出现的次数。注：

hash_map(query,query_count)是用来统计每个 query 的出现次数，不是存储他们的值，出现一次，则 count+1。

3. 堆/快速/归并排序：利用快速/堆/归并排序按照出现次数进行排序，将排序好的 query 和对应的 query_cout 输出到文件中，这样得到了 10 个排好序的文件（记为$b_0, b_1, \cdots, b_{10}$）。最后，对这 10 个文件进行归并排序（内排序与外排序相结合）。根据此方案 1，这里有一份实现：
https://github.com/ooooola/sortquery/blob/master/querysort.py。

除此之外，此题还有以下两个方法：

方案 2：一般 query 的总量是有限的，只是重复的次数比较多而已，可能对于所有的 query，一次性就可以加入到内存了。这样，我们就可以采用 trie 树/hash_map 等直接来统计每个 query 出现的次数，然后按出现次数做快速/堆/归并排序就可以了。

方案 3：与方案 1 类似，但在做完 hash，分成多个文件后，可以交给多个文件来处理，采用分布式的架构来处理（比如 MapReduce），最后再进行合并。

6、 给定 a、b 两个文件，各存放 50 亿个 url，每个 url 各占 64 字节，内存限制是 4G，让你找出 a、b 文件共同的 url？

可以估计每个文件安的大小为 5G×64=320G，远远大于内存限制的 4G。所以不可能将其完全加载到内存中处理。考虑采取分而治之的方法。

1. 分而治之/hash 映射：遍历文件 a，对每个 url 求取$hash(url)\%1000$，然后根据所取得的值将 url 分别存储到 1000 个小文件（记为$a_0, a_2, \cdots, a_{999}$，这里漏写个了 a1）中。这样每个小文件的大约为 300M。遍历文件 b，采取和 a 相同的方式将 url 分别存储到 1000 小文件中（记为$b_0, b_1, \cdots, b_{999}$）。这样处理后，所有可能相同的 url 都在对应的小文件（a_0 vs b_0, a_1 vs b_1, ..., a_0 vs b_{999}）中，不对应的小文件不可能有相同的 url。然后我们只要求出 1000 对小文件中相同的 url 即可。

2. hash_set 统计：求每对小文件中相同的 url 时，可以把其中一个小文件的 url 存储到 hash_set 中。然后遍历另一个小文件的每个 url，看其是否在刚才构建的 hash_set 中，如果是，那么就是共同的 url，存到文件里面就可以了。

OK，此第一种方法：分而治之/hash 映射 + hash 统计 + 堆/快速/归并排序，再看最后 4 道题，如下：

7、怎么在海量数据中找出重复次数最多的一个？

方案：先做 hash，然后求模映射为小文件，求出每个小文件中重复次数最多的一个，并记录重复次数。然后找出上一步求出的数据中重复次数最多的一个就是所求（具体参考前面的题）。

8、上千万或上亿数据（有重复），统计其中出现次数最多的前 N 个数据。

方案：上千万或上亿的数据，现在的机器的内存应该能存下。所以考虑采用 hash_map/搜索二叉树/红黑树等来进行统计次数。然后利用堆取出前 N 个出现次数最多的数据。

9、一个文本文件，大约有一万行，每行一个词，要求统计出其中最频繁出现的前 10 个词，请给出思想，给出时间复杂度分析。

方案 1：如果文件比较大，无法一次性读入内存，可以采用 hash 取模的方法，将大文件分解为多个小文件，对于单个小文件利用 hash_map 统计出每个小文件中 10 个最常出现的词，然后再进行归并处理，找出最终的 10 个最常出现的词。

方案 2：通过 hash 取模将大文件分解为多个小文件后，除了可以用 hash_map 统计出每个小文件中 10 个最常出现的词，也可以用 trie 树统计每个词出现的次数，时间复杂度是 O(n*le)（le 表示单词的平准长度），最终同样找出出现最频繁的前 10 个词（可用堆来实现），时间复杂度是 O(n*lg10)。

10. 1000 万字符串，其中有些是重复的，需要把重复的全部去掉，保留没有重复的字符串。请怎么设计和实现？

- 方案 1：这题用 trie 树比较合适，hash_map 也行。
- 方案 2：from xjbzju:，1000w 的数据规模插入操作完全不现实，以前试过在 stl 下 100w 元素插入 set 中已经慢得不能忍受，觉得基于 hash 的实现不会比红黑树好太多，使用 vector+sort+unique 都要可行许多，建议还是先 hash 成小文件分开处理再综合。

上述方案 2 中读者 xbzju 的方法让我想到了一些问题，即是 set/map，与 hash_set/hash_map 的性能比较?共计 3 个问题，如下：

- 1、hash_set 在千万级数据下，insert 操作优于 set？这位 blog：http://t.cn/zOibP7t 给的实践数据可靠不?
- 2、那 map 和 hash_map 的性能比较呢？谁做过相关实验?

- 3、那查询操作呢，如下段文字所述?

可以发现在hash_table中最长的链表也只有5个元素，**长度为1和长度为2的链表中的数据占全部数据的89%以上。因此绝大数查询将仅仅访问哈希表1次到2次。**这样的查询效率当然会比set（内部使用红黑树——类似于二叉平衡树）高的多。有了这个图示，无疑已经可以证明hash_set会比set快速高效了。但hash_set还可以动态的增加表的大小，因此我们再实现一个表大小可增加的hash_table。

或者小数据量时用 map，构造快，大数据量时用 hash_map?

rbtree PK hashtable

据朋友№邦卡猫№的做的红黑树和 hash table 的性能测试中发现：当数据量基本上 int 型 key 时，hash table 是 rbtree 的 3-4 倍，但 hash table 一般会浪费大概一半内存。

因为 hash table 所做的运算就是个%，而 rbtree 要比较很多，比如 rbtree 要看 value 的数据 ，每个节点要多出 3 个指针（或者偏移量） 如果需要其他功能，比如，统计某个范围内的 key 的数量，就需要加一个计数成员。

且 1s rbtree 能进行大概 50w+次插入，hash table 大概是差不多 200w 次。不过很多的时候，其速度可以忍了，例如倒排索引差不多也是这个速度，而且单线程，且倒排表的拉链长度不会太大。正因为基于树的实现其实不比 hashtable 慢到哪里去，所以数据库的索引一般都是用的 B/B+树，而且 B+树还对磁盘友好(B 树能有效降低它的高度，所以减少磁盘交互次数)。比如现在非常流行的 NoSQL 数据库，像 MongoDB 也是采用的 B 树索引。关于 B 树系列，请参考本 blog 内此篇文章：从 B 树、B+树、B*树谈到 R 树。更多请待后续实验论证。

11. 一个文本文件，找出前 10 个经常出现的词，但这次文件比较长，说是上亿行或十亿行，总之无法一次读入内存，问最优解。

方案 1：首先根据用 hash 并求模，将文件分解为多个小文件，对于单个文件利用上题的方法求出每个文件件中 10 个最常出现的词。然后再进行归并处理，找出最终的 10 个最常出现的词。

12. 100w 个数中找出最大的 100 个数。

方案 1：采用局部淘汰法。选取前 100 个元素，并排序，记为序列 L。然后一次扫描剩余的元素 x，与排好序的 100 个元素中最小的元素比，如果比这个最小的

要大，那么把这个最小的元素删除，并把 x 利用插入排序的思想，插入到序列 L 中。依次循环，知道扫描了所有的元素。复杂度为 O(100w*100)。

方案 2：采用快速排序的思想，每次分割之后只考虑比轴大的一部分，知道比轴大的一部分在比 100 多的时候，采用传统排序算法排序，取前 100 个。复杂度为 O(100w*100)。

方案 3：在前面的题中，我们已经提到了，用一个含 100 个元素的最小堆完成。复杂度为 O(100w*lg100)。

接下来，咱们来看第二种方法，双层捅划分。

密匙二、多层划分

多层划分----其实本质上还是分而治之的思想，重在“分”的技巧上！

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。

问题实例：

13、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 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。

14、5 亿个 int 找它们的中位数。

1. 思路一：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。

2. 思路二@绿色夹克衫：同样需要做两遍统计，如果数据存在硬盘上，就需要读取 2 次。

   方法同基数排序有些像，开一个大小为 65536 的 Int 数组，第一遍读取，统计 Int32 的高 16 位的情况，也就是 0-65535，都算作 0,65536 - 131071 都算作 1。就相当于用该数除以 65536。Int32 除以 65536 的结果不会超过 65536 种情况，因此开一个长度为 65536 的数组计数就可以。每读取一个数，数组中对应的计数+1，考虑有负数的情况，需要将结果加 32768 后，记录在相应的数组内。

   第一遍统计之后，遍历数组，逐个累加统计，看中位数处于哪个区间，比如处于区间 k，那么 0- k-1 的区间里数字的数量 sum 应该<n/2（2.5 亿）。而 k+1 - 65535 的计数和也<n/2，第二遍统计同上面的方法类似，但这次只统计处于区间 k 的情况，也就是说(x / 65536) + 32768 = k。统计只统计低 16 位的情况。并且利用刚才统计的 sum，比如 sum = 2.49 亿，那么现在就是要在低 16 位里面找 100 万个数(2.5 亿-2.49 亿)。这次计

数之后，再统计一下，看中位数所处的区间，最后将高位和低位组合一下就是结果了。

密匙三：Bloom filter/Bitmap

Bloom filter

关于什么是 **Bloom filter**，请参看 blog 内此文：

- 海量数据处理之 Bloom Filter 详解

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：

对于原理来说很简单，位数组+k 个独立 hash 函数。将 hash 函数对应的值的位数组置 1，查找时如果发现所有 hash 函数对应位都是 1 说明存在，很明显这个过程并不保证查找的结果是 100%正确的。同时也不支持删除一个已经插入的关键字，因为该关键字对应的位会牵动到其他的关键字。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。

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。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 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>=nlg(1/E)*lge 大概就是 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 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。

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。

可以看下上文中的第 6 题：

“6、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，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。

同时，上文的第 5 题：给定 a、b 两个文件，各存放 50 亿个 url，每个 url 各占 64 字节，内存限制是 4G，让你找出 a、b 文件共同的 url？如果允许有一定的错误率，可以使用 Bloom filter，4G 内存大概可以表示 340 亿 bit。将其中一个文件中的 url 使用 Bloom filter 映射为这 340 亿 bit，然后挨个读取另外一个文件的 url，检查是否与 Bloom filter，如果是，那么该 url 应该是共同的 url（注意会有一定的错误率）。”

Bitmap

- 关于什么是 Bitmap，请看 blog 内此文第二部分：

  http://blog.csdn.net/v_july_v/article/details/6685962。

下面关于 Bitmap 的应用，可以看下上文中的第 13 题，以及另外一道新题：

“13、在 2.5 亿个整数中找出不重复的整数，注，内存不足以容纳这 2.5 亿个整数。

方案 1：采用 2-Bitmap（每个数分配 2bit，00 表示不存在，01 表示出现一次，10 表示多次，11 无意义）进行，共需内存 2^32 * 2 bit=1 GB 内存，还可以接受。然后扫描这 2.5 亿个整数，查看 Bitmap 中相对应位，如果是 00 变 01，01 变 10，10 保持不变。所描完事后，查看 bitmap，把对应位是 01 的整数输出即可。

方案 2：也可采用与第 1 题类似的方法，进行划分小文件的方法。然后在小文件中找出不重复的整数，并排序。然后再进行归并，注意去除重复的元素。”

15、给 40 亿个不重复的 unsigned int 的整数，没排过序的，然后再给一个数，如何快速判断这个数是否在那 40 亿个数当中？

方案 1：frome oo，用位图/Bitmap 的方法，申请 512M 的内存，一个 bit 位代表一个 unsigned int 值。读入 40 亿个数，设置相应的 bit 位，读入要查询的数，查看相应 bit 位是否为 1，为 1 表示存在，为 0 表示不存在。

密匙四、Trie 树/数据库/倒排索引

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. 上面的**第 2 题**：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。
2. 上面的**第 5 题**：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。
3. 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？
4. 上面的**第 8 题**：一个文本文件，大约有一万行，每行一个词，要求统计出其中最频繁出现的前 10 个词。其解决方法是：用 trie 树统计每个词出现的次数，时间复杂度是 O(n*le)（le 表示单词的平准长度），然后是找出出现最频繁的前 10 个词。

更多有关 Trie 树的介绍，请参见此文：从 Trie 树（字典树）谈到后缀树。

数据库索引

适用范围：大数据量的增删改查

基本原理及要点：利用数据的设计实现方法，对海量数据的增删改查进行处理。

- 关于数据库索引及其优化，更多可参见此文：http://www.cnblogs.com/pkuoliver/archive/2011/08/17/mass-data-topic-7-index-and-optimize.html；
- 关于 MySQL 索引背后的数据结构及算法原理，这里还有一篇很好的文章：http://blog.codinglabs.org/articles/theory-of-mysql-index.html；
- 关于 B 树、B+ 树、B* 树及 R 树，本 blog 内有篇绝佳文章：http://blog.csdn.net/v_JULY_v/article/details/6530142。

倒排索引(Inverted index)

适用范围：搜索引擎，关键字查询

基本原理及要点：为何叫倒排索引？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。
以英文为例，下面是要被索引的文本：

T0 = "it is what it is"

T1 = "what is it"

T2 = "it is a banana"

我们就能得到下面的反向文件索引：

"a": {2}

"banana": {2}

"is": {0, 1, 2}

"it": {0, 1, 2}

"what": {0, 1}

检索的条件"what","is"和"it"将对应集合的交集。

正向索引开发出来用来存储每个文档的单词的列表。正向索引的查询往往满足每个文档有序频繁的全文查询和每个单词在校验文档中的验证这样的查询。在正向索引中，文档占据了中心的位置，每个文档指向了一个它所包含的索引项的序列。也就是说文档指向了它包含的那些单词，而反向索引则是单词指向了包含它的文档，很容易看到这个反向的关系。

扩展：

问题实例：文档检索系统，查询那些文件包含了某单词，比如常见的学术论文的关键字搜索。

关于倒排索引的应用，更多请参见：

- 第二十三、四章：杨氏矩阵查找，倒排索引关键词 Hash 不重复编码实践，
- 第二十六章：基于给定的文档生成倒排索引的编码与实践。

密匙五、外排序

适用范围：大数据的排序，去重

基本原理及要点：外排序的归并方法，置换选择败者树原理，最优归并树

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 明显不够，所以可以用来排序。内存可以当输入缓冲区使用。

关于多路归并算法及外排序的具体应用场景，请参见 blog 内此文：

- 第十章、如何给 10^7 个数据量的磁盘文件排序

密匙六、分布式处理之 Mapreduce

MapReduce 是一种计算模型，简单的说就是将大批量的工作（数据）分解（MAP）执行，然后再将结果合并成最终结果（REDUCE）。这样做的好处是可以在任务被分解后，可以通过大量机器进行并行计算，减少整个操作的时间。但如果你要我再通俗点介绍，那么，说白了，Mapreduce 的原理就是一个归并排序。

适用范围：数据量大，但是数据种类小可以放入内存

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。

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:
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)？

更多具体阐述请参见 blog 内：

- 从 Hadhoop 框架与 MapReduce 模式中谈海量数据处理，
- 及 MapReduce 技术的初步了解与学习。

其它模式/方法论，结合操作系统知识

至此，六种处理海量数据问题的模式/方法已经阐述完毕。据观察，这方面的面试题无外乎以上一种或其变形，然题目为何取为是：秒杀 99%的海量数据处理面试题，而不是 100%呢。OK，给读者看最后一道题，如下：

非常大的文件，装不进内存。每行一个 int 类型数据，现在要你随机取 100 个数。

我们发现上述这道题，无论是以上任何一种模式/方法都不好做，那有什么好的别的方法呢？我们可以看看：操作系统内存分页系统设计(说白了，就是映射+建索引)。

Windows 2000 使用基于分页机制的虚拟内存。每个进程有 4GB 的虚拟地址空间。基于分页机制，这 4GB 地址空间的一些部分被映射了物理内存，一些部分映射硬盘上的交换文 件，一些部分什么也没有映

射。程序中使用的都是 4GB 地址空间中的虚拟地址。而访问物理内存，需要使用物理地址。 关于什么是物理地址和虚拟地址，请看：

- 物理地址 (physical address): 放在寻址总线上的地址。放在寻址总线上，如果是读，电路根据这个地址每位的值就将相应地址的物理内存中的数据放到数据总线中传输。如果是写，电路根据这个 地址每位的值就将相应地址的物理内存中放入数据总线上的内容。物理内存是以字节(8 位)为单位编址的。
- 虚拟地址 (virtual address): 4G 虚拟地址空间中的地址，程序中使用的都是虚拟地址。 使用了分页机制之后，4G 的地址空间被分成了固定大小的页，每一页或者被映射到物理内存，或者被映射到硬盘上的交换文件中，或者没有映射任何东西。对于一 般程序来说，4G 的地址空间，只有一小部分映射了物理内存，大片大片的部分是没有映射任何东西。物理内存也被分页，来映射地址空间。对于 32bit 的 Win2k，页的大小是 4K 字节。CPU 用来把虚拟地址转换成物理地址的信息存放在叫做页目录和页表的结构里。

物理内存分页，一个物理页的大小为 4K 字节，第 0 个物理页从物理地址 0x00000000 处开始。由于页的大小为 4KB，就是 0x1000 字节，所以第 1 页从物理地址 0x00001000 处开始。第 2 页从物理地址 0x00002000 处开始。可以看到由于页的大小是 4KB，所以只需要 32bit 的地址中高 20bit 来寻址物理页。

返回上面我们的题目：**非常大的文件，装不进内存。每行一个 int 类型数据，现在要你随机取 100 个数**。针对此题，我们可以借鉴上述操作系统中内存分页的设计方法，做出如下解决方案：

操作系统中的方法，先生成 4G 的地址表，在把这个表划分为小的 4M 的小文件做个索引，二级索引。30 位前十位表示第几个 4M 文件，后 20 位表示在这个 4M 文件的第几个，等等，基于 key value 来设计存储，用 key 来建索引。

但如果现在只有 10000 个数，然后怎么去随机从这一万个数里面随机取 100 个数？请读者思考。更多海里数据处理面试题，请参见此文第一部分：http://blog.csdn.net/v_july_v/article/details/6685962。

参考文献

1. 十道海量数据处理面试题与十个方法大总结；
2. 海量数据处理面试题集锦与 Bit-map 详解；
3. 十一、从头到尾彻底解析 Hash 表算法；
4. 海量数据处理之 Bloom Filter 详解；
5. 从 Trie 树（字典树）谈到后缀树；
6. 第三章续、Top K 算法问题的实现；
7. 第十章、如何给 10^7 个数据量的磁盘文件排序；
8. 从 B 树、B+树、B*树谈到 R 树；
9. 第二十三、四章：杨氏矩阵查找，倒排索引关键词 Hash 不重复编码实践；
10. 第二十六章：基于给定的文档生成倒排索引的编码与实践；
11. 从 Hadhoop 框架与 MapReduce 模式中谈海量数据处理；
12. 第十六~第二十章：全排列，跳台阶，奇偶排序，第一个只出现一次等问题；
13. http://blog.csdn.net/v_JULY_v/article/category/774945；
14. STL 源码剖析第五章，侯捷著；
15. 2012 百度实习生招聘笔试题：

    http://blog.csdn.net/hackbuteer1/article/details/7542774；
16. Redis/MongoDB 绝佳站点：http://blog.nosqlfan.com/；

17. 国外一面试题网站：http://www.careercup.com/。

后记

经过上面这么多海量数据处理面试题的轰炸，我们依然可以看出这类问题是有一定的解决方案/模式的，所以，不必将其神化。然这类面试题所包含的问题还是比较简单的，若您在这方面有更多实践经验，欢迎在本文评论下与大家不吝分享。

不过，相信你也早就意识到，若单纯论海量数据处理面试题，本 blog 内的有关海量数据处理面试题的文章已涵盖了你能在网上所找到的 70~80%。但有点，必须负责任的敬告大家：无论是这些海量数据处理面试题也好，还是算法也好，面试时，70~80%的人不是倒在这两方面，而是倒在基础之上(诸如语言，数据库，操作系统，网络协议等等)，所以，无论任何时候，基础最重要，没了基础，便什么都不是。

最后，推荐几个求职/算法/刷题的相关课程，包括 LeetCode 直播刷题等等，感兴趣的可以看下：https://www.julyedu.com/category/index/1。

OK，本文若有任何问题，欢迎随时不吝留言，评论，赐教，谢谢。完。