第 6 章　海量数据处理[1]

本章导读

所谓海量数据处理，是指基于海量数据的存储、处理、和操作。正因为数据量太大，所以导致要么无法在较短时间内迅速解决，要么无法一次性装入内存。

事实上，针对时间问题，可以采用巧妙的算法搭配合适的数据结构（如布隆过滤器、哈希、位图、堆、数据库、倒排索引、Trie 树）来解决；而对于空间问题，可以采取分而治之（哈希映射）的方法，也就是说，把规模大的数据转化为规模小的，从而各个击破。

此外，针对常说的单机及集群问题，通俗来讲，单机就是指处理装载数据的机器有限（只要考虑 CPU、内存、和硬盘之间的数据交互），而集群的意思是指机器有多台，适合分布式处理或并行计算，更多考虑节点与节点之间的数据交互。

一般说来，处理海量数据问题，有以下十种典型方法：

1. 哈希分治；
2. simhash 算法；
3. 外排序；
4. MapReduce；
5. 多层划分；
6. 位图；
7. 布隆过滤器；
8. Trie 树；
9. 数据库；

---

[1] 本章为书籍初稿第 6 章，并在博客文章《教你如何迅速秒杀掉：99%的海量数据处理面试题》：http://blog.csdn.net/v_july_v/article/details/7382693 的基础上优化。本初稿还会经过一轮的改进优化，最终将被收录于今 2014 年内出版的新书中。如有任何改进意见，欢迎通过微博：http://weibo.com/julyweibo 向我反馈，thanks。July、二零一四年八月十八日晚。

10. 倒排索引。

受理论之限，本章将摒弃绝大部分的细节，只谈方法和模式论，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，全章行文是基于面试题的分析基础之上的，具体实践过程中，还得视具体情况具体分析，且各个场景下需要考虑的细节也远比本章所描述的任何一种解决方案复杂得多。

6.1 关联式容器

一般来说，STL 容器分两种：序列式容器和关联式容器。

序列式容器包括 vector、list、deque、stack、queue、heap 等容器。而关联式容器，每笔数据或每个元素都有一个键值（key）和一个实值（value），即所谓的键-值对（Key-Value）。当元素被插入到关联式容器中时，容器的内部结构（可能是红黑树，也可能是哈希表）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。

在 C++ 11 标准之前，旧标准规定标准的关联式容器分为 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），这些容器均基于红黑树（red-black tree）实现。此外，还有另一类非标准的关联式容器，即 hashtable（哈希表，又称散列表），以及以 hashtable 为底层实现机制的 hash_set（散列集合）、hash_map（散列映射表）、hash_multiset（散列多键集合）、和 hash_multimap（散列多键映射表）。

也就是说，set、map、multiset、和 multimap 都内含一个 red-black tree，而 hash_set、hash_map、hash_multiset、和 hash_multimap 都内含一个 hashtable[2]。

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 和 map 两者的所有各种操作，都只是转而调用 red-black tree 的操作行为。不过，值得注意的是，两者都不允许任意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，同时拥有实值和键值，pair 的第一个元素被视为键值，第二个元素被视为实值。

---

[2] C++ 11 标准之后，准备引入非标准的关联式容器 hashtable，但为了避免与已经存在的 hash_map 等第三方容器产生名字冲突，故命名了基于 hash 函数实现的 unordered_set、unordered_map，和 unordered_multiset、unordered_multimap，相当于 hash_set、hash_ma，和 hash_multiset、hash_multimap。

但采用哪种名字不是本书的重点，所以下文在用到无序的关联式容器时，依然会继续沿用旧标准中的 hash_set/hash_map/hash_multiset/hash_multimap。

至于 multiset 和 multimap，它们的特性及用法与 set 和 map 几乎相同，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 red-black tree 的 insert_equal()而非 insert_unique()。

hash_set、hash_map、hash_multiset、和 hash_multimap

hash_set 和 hash_map，两者的一切操作都是基于 hashtable。不同的是，hash_set 同 set 一样，同时拥有实值和键值，且实值就是键值，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和一个键值，所以其使用方式和 map 基本相同。但由于 hash_set 和 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 和 hash_multimap 的特性与 multiset 和 multimap 完全相同，唯一的差别就是 hash_multiset 和 hash_multimap 的底层实现机制是 hashtable（区别于 multiset 和 multimap 的底层实现机制 red-black 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。

综上所述，什么样的结构决定其什么样的性质，因为 **set、map、multiset、和 multimap 的实现都是基于 red-black tree 的**，所以有自动排序功能，而 **hash_set、hash_map、hash_multiset、和 hash_multimap 的实现都是基于 hashtable 的**，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。

6.2 哈希分治

方法介绍

对于海量数据而言，由于无法一次性装进内存处理，不得不把海量的数据通过 hash 映射的方法分割成相应的小块数据，然后再针对各个小块数据通过 hash_map 进行统计或其他操作。

那什么是 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大数据，我们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数据存放在内存中，或大文件映射成多个小文件），而这种映射散列的方式便是我们通常所说的 hash 函数，好的 hash 函数能让数据均匀分布而减少冲突。

问题实例

1. 寻找 TOP IP

海量日志数据，提取出某日访问百度次数最多的那个 IP。

分析：百度作为国内第一大搜索引擎，每天访问它的 IP 数量巨大，如果想一次性把所有 IP 数据装进内存处理，则内存容量通常不够，故针对数据太大，内存受限的情况，可以把大文件转化成（取模映射）小文件，从而大而化小，逐个处理。

换言之，先映射，而后统计，最后排序。

解法：具体分为以下 3 个步骤。

1. 分而治之/hash 映射。首先将该日访问百度的所有 IP 从访问日志中提取出来，然后逐个写入到一个大文件中，接着采取 Hash 映射的方法（比如 hash(IP) % 1000[3]），把整个大文件的数据映射到 1000 个小文件中。

2. hash_map 统计。当大文件转化成了小文件，那么我们便可以采用 hash_map(ip, value)来分别对 1000 个小文件中的 IP 进行频率统计，找出每个小文件中出现频率最大的 IP，总共 1000 个 IP。

---

[3] hash 取模是一种等价映射，不会存在同一个 IP 分散到不同小文件中的情况。换言之，这里采用的是%1000 算法，那么同一个 IP 在 hash 后，只可能落在同一个小文件中，不可能被分散。

3. 堆/快速排序。统计出 1000 个频率最大的 IP 后，依据它们各自频率的大小进行排序（可采取堆排序），找出那个出现频率最大的 IP，即为所求。

2. 寻找热门查询

搜索引擎会通过日志文件把用户每次检索所使用的所有查询串都记录下来，每个查询串的长度为 1-255 字节。假设目前有 1000 万个查询记录（但因为这些查询串的重复度比较高，所以虽然总数是 1000 万，但如果除去重复后，不超过 300 万个查询字符串），请统计其中最热门的 10 个查询串，要求使用的内存不能超过 1G。

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。

分析：如果是一亿个 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 hash_map 统计并按数量排序，最后用归并或者最小堆依次处理每个小文件的 top 10 以得到最后的结果。

但对于本题，数据规模比较小，能一次性装入内存。因为根据题目描述，虽然有 1000 万个 Query，但是由于重复度比较高，故去除重复后，事实上只有 300 万的 Query，每个 Query 为 255Byte，因此我们可以考虑把他们全部放进内存中去（假设 300 万个字符串没有重复，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）。

所以我们放弃分而治之/hash 映射的步骤，直接用 hash_map 统计，然后排序。事实上，针对此类典型的 TOP K 问题，采取的对策一般都是：分而治之/hash 映射（如有必要） + hash_map + 堆。

解法：

1. hash_map 统计。先对这批海量数据进行预处理，用 hash_map 完成频率统计。具体做法是：维护一个 Key 为 Query，Value 为该 Query 出现次数的 hash_map，即 hash_map(Query, Value)，每次读取一个 Query，如果该 Query 不在 hash_map 中，那么将该 Query 放入 hash_map 中，并将它的 Value 值设为 1；如果该 Query 在 hash_map 中，那么将该 Query 的计数 Value 加 1 即可。最终我们用 hash_map 在 O(n)的时间复杂度内完成了所有 Query 的频率统计。

2. 堆排序。借助堆这个数据结构，找出 Top K，时间复杂度为 N'O(logK)。即借助堆结构，我们可以在 log 量级的时间内查找或调整移动。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，分别和根元素进行比较。所以，最终的时间复杂度是：O(n) + N' O(logK），其中，N 为 1000 万，N'为 300 万。

关于第 2 步堆排序，进一步讲，可以维护 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(每次调整堆费时 O(logk))后，有 k1 > k2 ... > kmin（kmin 设为最小堆中最小元素）。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 O(logk)），否则不更新堆。这样下来，总费时 O(k + k logk + (n-k)logk) = O(n logk)。此方法得益于在堆中，查找等各项操作的时间复杂度均为 O(logk)。

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

3. 寻找频数最高的 100 个词

有一个 1G 大小的文件，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。

解法：

1.分而治之/hash 映射。按先后顺序读取文件，对于每个词 x，执行 hash(x)%5000，然后将该值存到 5000 个小文件（记为 x0, x1, ..., x4999）中。如此每个文件的大小大概是 200k 左右。当然，如果其中有的小文件超过了 1M 大小，则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。最后把这 5000 个文件进行归并（可以用归并排序）。

4. 寻找 TOP 10

海量数据分布在 100 台电脑中，请想个办法高效统计出这批数据的 TOP 10。

解法一：如果同一个数据元素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为 TOP 10 的数据元素。

1.堆排序。在每台电脑上求出 TOP 10，可以采用包含 10 个元素的堆完成（求 TOP 10 小用最大堆，求 TOP 10 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，假设这 10 个元素就是 TOP 10 大，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，否则不调整。最后堆中的元素就是 TOP 10 大）。

2.组合归并。求出每台电脑上的 TOP 10 后，然后把这 100 台电脑上的 TOP 10 组合起来，共 1000 个数据，再利用上面类似的方法求出 TOP 10 就可以了。

解法二：但如果同一个元素重复出现在不同的电脑中呢？举个例子，给定两台机器，第一台的数据及各自出现频率为：a(50)，b(50)，c(49)，d(49)，e(0)，f(0)，第二台的数据及各自出现频率为：a(0)，b(0)，c(49)，d(49)，e(50)，f(50)，求 TOP 2。其中，括号里的数字代表某个数据出现的频率，如 a(50)表示 a 出现了 50 次。

这个时候，有两种方法可以解决：

- 要么遍历一遍所有数据，重新 hash 取模，如此使得同一个元素只出现在单独的一台电脑中，然后采取上面所说的方法，统计每台电脑中各个元素的出现次数找出 TOP 10，继而组合 100 台电脑上的 TOP 10，找出最终的 TOP 10。
- 要么暴力求解，直接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 TOP 10。

5. 查询串的重新排列

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。

解法一：分为以下 3 个步骤，如下：

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。这样新生成的每个文件的大小约为 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 b0, b1, ..., b9）。最后，对这 10 个文件进行归并排序（内排序与外排序相结合）。

解法二：一般 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。

解法三：与解法 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. 寻找共同的 URL

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，请找出 a、b 文件共同的 url。

解法：

可以估计每个文件的大小为 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$\text{hash}(url)\%1000$，然后根据所取得的值将 url 分别存储到 1000 个小文件（记为$a_0, a_2, \cdots, a_{999}$~~，这里漏写个子 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 $b_1, \ldots, a_0$ 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。

举一反三

1. 寻找最大的 100 个数

100 万个数中找出最大的 100 个数。

分析：可以采用局部淘汰法。选取前 100 个元素，并排序，记为序列 L。然后依次扫描剩余的元素 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，利用插入排序的思想，将 x 插入到序列 L 中。依次循环，直到扫描了所有的元素。复杂度为 O(100 万×100)。

也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(100 万×100)。

包括也可以用一个含 100 个元素的最小堆完成。复杂度为 O(100 万×lg100)。

2. 统计 10 个频繁出现的词

一个文本文件，有上亿行甚至十亿行，每行一个词，要求统计出其中出现最频繁的前 10 个词，请给出思路和时间复杂度的分析。

提示：因为文件比较大，无法一次读入内存，故可以用 hash 并求模，将文件分解为多个小文件，对于单个文件利用 hash_map 统计出每个文件中 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。

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）。然后找出出现最频繁的前 10 个词，可用堆来实现，前面的题中已经讲到了，时间复杂度是 O(n×lg10)。所以总的时间复杂度，是 O(n×le)与 O(n lg10)中较大的那一个。

3. 寻找出现次数最多的数字

怎么在海量数据中找出重复次数最多的一个？

提示：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。最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。

5. 统计出现次数最多的前 N 个数据

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。

提示：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。然后取出前 N 个出现次数最多的数据，可以用堆机制完成。

6. 1000 万字符串的去重

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。请问怎么设计和实现？

提示：这题用 trie 树比较合适，hash_map 也行。当然，也可以先 hash 成小文件分开处理再综合。

6.3 simhash 算法

方法介绍

背景

如果某一天，面试官问你如何设计一个比较两篇文章相似度的算法？你可能会回答几个比较传统点的方案：

- 一种方案是先将两篇文章分别进行分词，得到一系列特征向量，然后计算特征向量之间的距离（可以计算它们之间的欧氏距离、海明距离或者夹角余弦等等），从而通过距离的大小来判断两篇文章的相似度。
- 另外一种方案是利用传统的 hash，通过 hash 的方式为每一个 web 文档生成一个指纹（finger print）。

下面，我们来分析下这两种方法。

- 采取第一种方法，若是只比较两篇文章的相似性还好，但如果是海量数据呢，有着数以百万甚至亿万的网页，要求你计算这些网页的相似度。你还会去计算任意两个网页之间的距离或夹角余弦么？想必你不会了。
- 而第二种方案中所说的传统加密方式 md5，其设计的目的是为了让整个分布尽可能地均匀，但如果输入内容一旦出现哪怕轻微的变化，hash 值就会发生很大的变化。举个例子，我们假设有以下三段文本：

```
the cat sat on the mat
the cat sat on a mat
we all scream for ice cream
```

使用传统 hash 可能会得到如下的结果：

```
irb(main):006:0> p1 = 'the cat sat on the mat'
irb(main):007:0> p1.hash => 415542861
irb(main):005:0> p2 = 'the cat sat on a mat'
irb(main):007:0> p2.hash => 668720516
irb(main):007:0> p3 = 'we all scream for ice cream'
irb(main):007:0> p3.hash => 767429688 "
```

可理想当中的 hash 函数，需要对几乎相同的输入内容，产生相同或者相近的 hash 值，换言之，hash

值的相似程度要能直接反映输入内容的相似程度，故 md5 等传统 hash 方法也无法满足我们的需求。

出世

车到山前必有路，2002 年，来自 Princeton 的 Moses Charikar 提出 simhash 算法，随后 Google 把它发扬光大，专门用来解决亿万级别的网页的去重任务，并把应用的结果以论文的形式发表在 www07 会议上。Google 的这篇论文“detecting near-duplicates for web crawling”中展示了 simhash 算法中随机超平面的一个极其巧妙的实现，bit 差异的期望正好等于原始向量的余弦。

simhash 作为 Locality Sensitive Hash（局部敏感哈希）的一种，其主要思想是降维，将高维的特征向量映射成低维的特征向量，通过两个向量的 Hamming Distance 来确定文章是否重复或者高度近似。

其中，Hamming Distance，又称汉明距离，在信息论中，两个等长字符串之间的汉明距离是两个字符串对应位置的不同字符的个数。也就是说，它就是将一个字符串变换成另外一个字符串所需要替换的字符个数。例如：1011101 与 1001001 之间的汉明距离是 2。至于我们常说的字符串编辑距离则是一般形式的汉明距离。

如此，通过比较多个文档的 simhash 值的海明距离，可以获取它们的相似度。

流程

simhash 算法分为 5 个步骤：分词、hash、加权、合并、降维，具体过程如下所述：

- 分词。给定一段语句，进行分词，得到有效的特征向量，然后为每一个特征向量设置 1-5 等 5 个级别的权重（如果是给定一个文本，那么特征向量可以是文本中的词，其权重可以是这个词出现的次数）。例如给定一段语句：“CSDN 博客结构之法算法之道的作者 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)，其中括号里的数字代表这个单词在整条语句中的重要程度，数字越大代表越重要。
- Hash。通过 hash 函数计算各个特征向量的 hash 值，hash 值为二进制数 01 组成的 n-bit 签名。比如“CSDN”的 hash 值 Hash(CSDN)为 100101，“博客”的 hash 值 Hash(博客)为“101011”。就这样，字符串就变成了一系列数字。
- 加权。在 hash 值的基础上，给所有特征向量进行加权，即 W = Hash * weight，且遇到 1 则 hash 值和权值正相乘，遇到 0 则 hash 值和权值负相乘。例如给“CSDN”的 hash 值“100101”加权得到：W(CSDN) = 100101*4 = 4 -4 -4 4 -4 4，给“博客”的 hash 值“101011”加权得到：W(博客)=101011*5 = 5 -5 5 -5 5 5，其余特征向量类似此般操作。

- 合并。将上述各个特征向量的加权结果累加，变成只有一个序列串。拿前两个特征向量举例，例如“CSDN”的“4 -4 -4 4 -4 4”和“博客”的“5 -5 5 -5 5 5”进行累加，得到“4+5 (-4)+(-5) (-4)+5 4+(-5) (-4)+5 4+5”，得到“9 -9 1 -1 1 9”。
- 降维。对于 n-bit 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 的海明距离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（某位大于 0 记为 1，小于 0 记为 0），得到的 01 串为：“1 0 1 0 1 1”，从而形成它们的 simhash 签名。

其流程如图 6-1 所示：

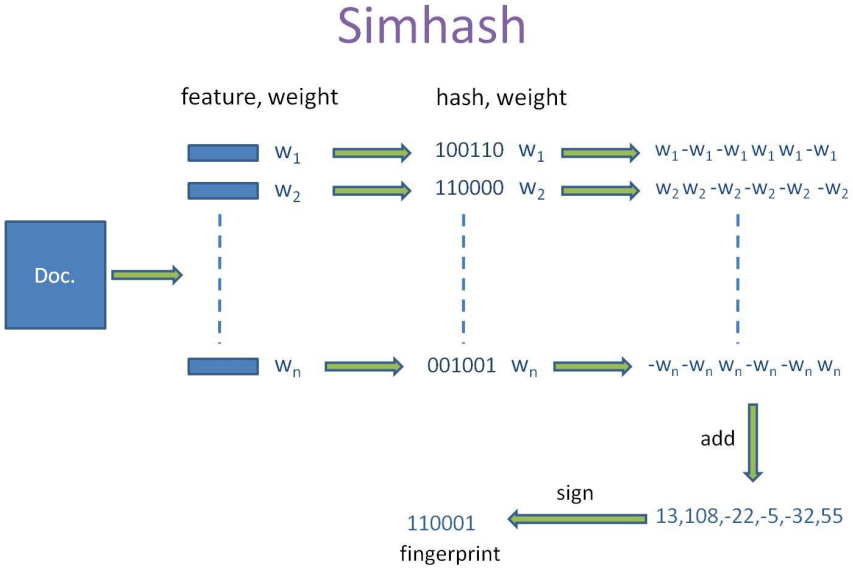

图 6-1

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。

解释下海明距离的求法。异或时，只有在两个比较的位不同时其结果是 1 ，否则结果为 0，两个二进制“异或”后得到 1 的个数即为海明距离的大小。

举个例子，上面我们计算到的“CSDN 博客”的 simhash 签名值为“1 0 1 0 1 1”，假定我们计算出另外一个短语的签名值为“1 0 1 0 0 0”，那么根据异或规则，我们可以计算出这两个签名的海明距离为 2，从而判定这两个短语的相似度是比较高的。

换言之，现在问题转换为：对于 64 位的 simhash 值，我们只要找到海明距离在 3 以内的所有签名，即可找出所有相似的短语。

但关键是，如何将其扩展到海量数据呢？譬如如何在海量的样本库中查询与其海明距离在 3 以内的记录呢？

- 一种方案是查找待查询文本的 64 位 simhash code 的所有 3 位以内变化的组合，此方案大约需要 4 万多次的查询。
- 另一种方案是预生成库中所有样本 simhash code 的 3 位变化以内的组合，此方案则大约需要占据 4 万多倍的原始空间。

这两种方案，要么时间复杂度高，要么空间复杂度高，能否有一种方案可以达到时空复杂度的绝佳平衡呢？

答案是肯定的。我们可以把 64 位的二进制 simhash 签名均分成 4 块，每块 16 位。根据鸽巢原理（也称抽屉原理），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。如图 6-2 所示：

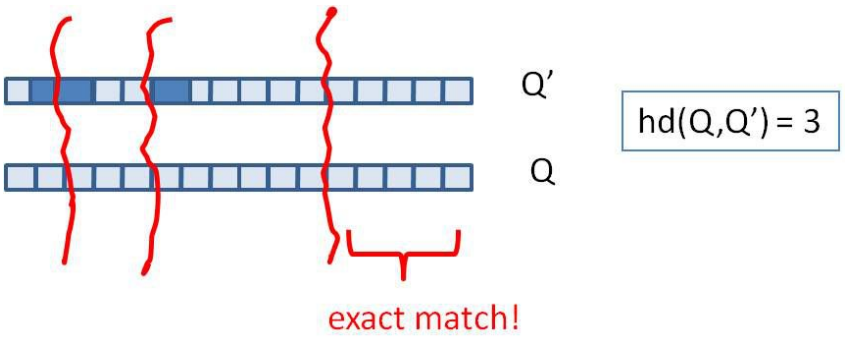

图 6-2

然后把分成的 4 块中的每一个块分别作为前 16 位来进行查找，建倒排索引。

具体如图 6-3 所示：

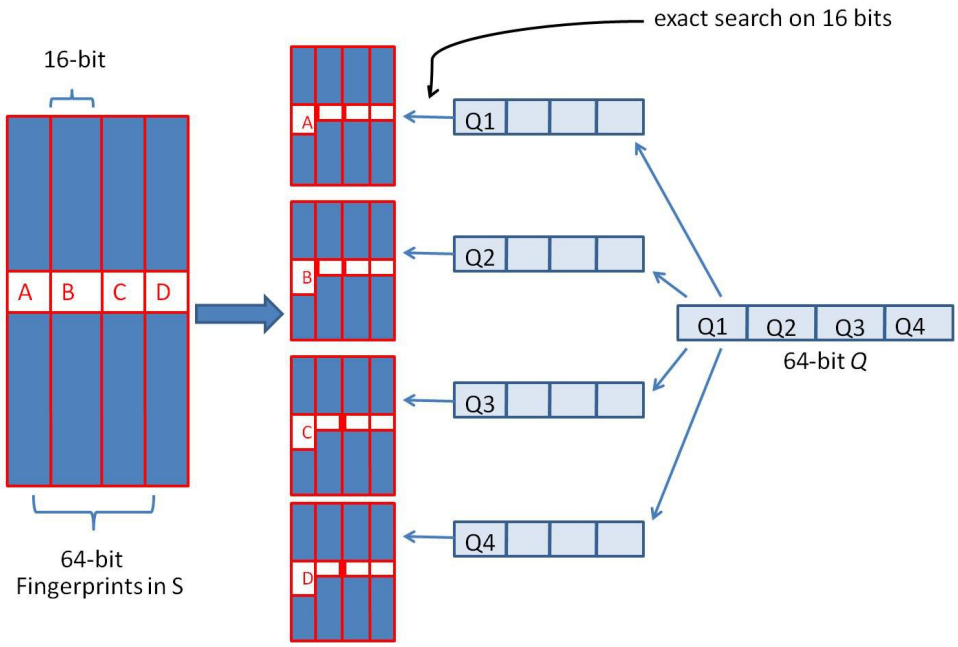

图 6-3

如此，如果样本库中存有 2^{34}（差不多 10 亿）的 simhash 签名，则每个 table 返回 $2^{(34-16)}=262144$ 个候选结果，大大减少了海明距离的计算成本。

举个例子，假设数据是均匀分布，16 位的数据，产生的象限为 2^{16} 个，则平均每个象限分布的文档数则为 $2^{34}/2^{16}=2^{(34-16)}$），四个块返回的总结果数为 4×262144（大概 100 万）。这样，原本需要比较 10 亿次，经过索引后，大概只需要处理 100 万次。

问题实例

1. 网页的存储

搜索引擎中的网络爬虫每天会下载互联网上的诸多网页，通常为了防止一个网页被爬虫重复下载，一般会把已经下载好的网页存到一个巨大的 hash 表中（然后下载一个新的网页时，从 hash 表中查找判断是否已经下载过这个新的网页，避免重复下载）。

但网页以什么样的形式存储在 hash 表中呢？如果以字符串的形式存储的话，那么可能大部分 URL 会被拖着很长，例如 baidu 搜“结构之法”，对应的链接比较长（100 个字符以上）：

http://www.baidu.com/s?wd=%E7%BB%93%E6%9E%84%E4%B9%8B%E6%B3%95&rsv_bp=0&tn=baidu&rsv_spt=3&ie=utf-8&rsv_sug3=3&rsv_sug4=53&rsv_sug1=2&rsv_sug2=0&inputT=1423&rsv_sug=1。

请问，如何解决这个以字符串形式存储网址导致的内存空间浪费的问题？

分析：为每一个网页分配一个随机数，这个随机数可以看做是这个网页的信息指纹。好比每一个人都有不同的手指指纹一样，只要产生随机数的算法足够好，那么每一个网页的信息指纹几乎都是独一无二的。由于这个信息指纹可以采用固定的 128bit，即 16 个字节的整数空间，如此，网页的信息指纹形式取代字符串存储于 hash 表中，不但节省存储空间，而且也利于查找。

2. 网页重复的判定

搜索引擎每天会检索大量网页，但有些网站的网页（比如一些推荐类新闻阅读工具）是转载其他网站的，如何从亿万级别的网页中迅速判定某个网页跟另一个网页是重复的，则是搜索引擎要解决的问题。那么，如何判断重复网页呢？

分析：给定两个网页，首先不可能也没必要把其中一个网页跟另一个网页一行一行内容的对比，可以提取出这两个网页的关键特征词，然后比对这些特征词是否相同，简而言之，就是抽取网页特征，然后比对网页特征。但很多时候，程序并不能智能而准确的判断到底哪些词是关键词，故这个方法仍有诸多不足。事实上，应用本节的思路，我们可以计算两个网页的 simhash 指纹，然后比对它们的 simhash 指纹即可。

举一反三

1. 视频网站的反重复

国内某一视频网站采取 UCG 的形式，让用户上传视频，但某一个视频被多个用户重复上传的情况屡屡发生，如何判定某用户上传的视频是否是重复视频呢？

分析：提取视频的关键帧，然后在视频库中查找匹配已有视频的关键帧，看是否存在同样关键帧的视频，如果已经存在，则当前视频不入库，否则入库。这种提取视频关键帧的方法跟提取网页信息指纹的方法原理一致。

2. 网盘网站的秒传

用过网盘的朋友可能知道，我们把一个文件上传到某些网盘上时，经常会遇到刚点击上传，一两秒之后便显示上传成功的“秒传”现象。网盘上传速度如此之快，其背后的原理是什么呢？

分析：类似于视频库的查重，网盘也需要处理某用户上传的文件是否已经存在于网盘库中。当用户上传某个文件时，系统计算出此文件的指纹，然后到后台的文件库中查找是否已存在相同文件。如果文件库

中已有相同文件，则把当前用户要上传的文件名迅速链接到已有文件所在的位置。如此，就不必再占用带宽再传一遍，从而导致用户明明上传的是一个很大的文件，但瞬间就显示文件已上传成功的有趣现象。

6.4 外排序

方法介绍

所谓外排序，顾名思义，就是在内存外面的排序，因为当要处理的数据量很大，而不能一次装入内存时，此时只能放在读写较慢的外存储器（通常是硬盘）上。

外排序通常采用的是一种“排序-归并”的策略。

- 在排序阶段，先读入能放在内存中的数据，将其排序输出到一个临时文件，依此进行，将待排序数据组织为多个有序的临时文件；
- 而后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。

假定现在有 20 个数据的文件 A：{5 11 0 18 4 14 9 7 6 8 12 17 16 13 19 10 2 1 3 15}，但一次只能使用仅装 4 个数据的内容，所以，我们可以每趟对 4 个数据进行排序，即 5 路归并，具体方法如下述步骤：

- 我们先把“大”文件 A，分割为 a1，a2，a3，a4，a5 等 5 个小文件，每个小文件 4 个数据
    - a1 文件为：5 11 0 18
    - a2 文件为：4 14 9 7
    - a3 文件为：6 8 12 17
    - a4 文件为：16 13 19 10
    - a5 文件为：2 1 3 15
- 然后依次对 5 个小文件分别进行排序
    - a1 文件完成排序后：0 5 11 18
    - a2 文件完成排序后：4 7 9 14
    - a3 文件完成排序后：6 8 12 17
    - a4 文件完成排序后：10 13 16 19
    - a5 文件完成排序后：1 2 3 15
- 最终多路归并，完成整个排序

整个大文件 A 文件完成排序后：0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

问题实例

给 10^7 个数据量的磁盘文件排序

输入：给定一个文件，里面最多含有 n 个不重复的正整数（也就是说可能含有少于 n 个不重复正整数），且其中每个数都小于等于 n，n=10^7。输出：得到按从小到大升序排列的包含所有输入的整数的列表。条件：最多有大约 1MB 的内存空间可用，但磁盘空间足够。且要求运行时间在 5 分钟以下，10 秒为最佳结果。

解法一：位图方案

你可能会想到把磁盘文件进行归并排序，但题目要求你只有 1MB 的内存空间可用，所以，归并排序这个方法不行。

熟悉位图的朋友可能会想到用位图来表示这个文件集合。例如正如《编程珠玑》一书上所述，用一个 20 位长的字符串来表示一个所有元素都小于 20 的简单的非负整数集合，边框用如下字符串来表示集合 {1,2,3,5,8,13}：

0 1 1 1 0 1 0 0 1 0 0 0 0 1 0 0 0 0 0 0

上述集合中各数对应的位置则置 1，没有对应的数的位置则置 0。

参考《编程珠玑》一书上的位图方案，针对我们的 10^7 个数据量的磁盘文件排序问题，我们可以这么考虑，由于每个 7 位十进制整数表示一个小于 1000 万的整数。我们可以使用一个具有 1000 万个位的字符串来表示这个文件，其中，当且仅当整数 i 在文件中存在时，第 i 位为 1。采取这个位图的方案是因为我们面对的这个问题的特殊性：

输入数据限制在相对较小的范围内，

数据没有重复，

其中的每条记录都是单一的整数，没有任何其他与之关联的数据。

所以，此问题用位图的方案分为以下三步进行解决：

- 第一步，将所有的位都置为 0，从而将集合初始化为空。
- 第二步，通过读入文件中的每个整数来建立集合，将每个对应的位都置为 1。
- 第三步，检验每一位，如果该位为 1，就输出对应的整数。

经过以上三步后，产生有序的输出文件。令 n 为位图向量中的位数（本例中为 1000 0000），程序可

以用伪代码表示如下：

```
//磁盘文件排序位图方案的伪代码
//第一步，将所有的位都初始化为 0
for i ={0,....n}
   bit[i]=0;
//第二步，通过读入文件中的每个整数来建立集合，将每个对应的位都置为 1。
for each i in the input file
   bit[i]=1;

//第三步，检验每一位，如果该位为 1，就输出对应的整数。
for i={0...n}
  if bit[i]==1
    write i on the output file
```

上述的位图方案，共需要扫描输入数据两次，具体执行步骤如下：

第一次，只处理 1—4999999 之间的数据，这些数都是小于 5000000 的，对这些数进行位图排序，只需要约 5000000/8=625000Byte，也就是 0.625M，排序后输出。第二次，扫描输入文件时，只处理 4999999-10000000 的数据项，也只需要 0.625M（可以使用第一次处理申请的内存）。因此，总共也只需要 0.625M 位图的的方法有必要强调一下，就是位图的适用范围为针对不重复的数据进行排序，若数据有重复，位图方案就不适用了。

不过很快，我们就将意识到，用此位图方法，严格说来还是不太行，空间消耗 $10^7/8$ 还是大于 1M（1M=1024×1024 空间，小于 $10^7/8$）。

既然如果用位图方案的话，我们需要约 1.25MB（若每条记录是 8 位的正整数的话，则 10000000/(1024×1024×8) 约等于 1.2M）的空间，而现在只有 1MB 的可用存储空间，那么究竟该作何处理呢？

解法二：多路归并

诚然，在面对本题时，通过计算分析出可以用如 2 的位图法解决，但实际上，很多的时候，我们都面临着这样一个问题，文件太大，无法一次性放入内存中计算处理，那这个时候咋办呢？分而治之，大而化小，也就是把整个大文件分为若干大小的几块，然后分别对每一块进行排序，最后完成整个过程的排序。k 趟算法可以在 kn 的时间开销内和 n/k 的空间开销内完成对最多 n 个小于 n 的无重复正整数的排序。

比如可分为 2 块（k=2，1 趟反正占用的内存只有 1.25/2M），1~4999999，和 5000000~9999999。先遍

历一趟，首先排序处理 1~4999999 之间的整数（用 5000000/8=625000 个字的存储空间来排序 0~4999999 之间的整数），然后再第二趟，对 5000001~1000000 之间的整数进行排序处理。

解法总结

关于本章中位图和多路归并两种方案的时间复杂度及空间复杂度的比较，如下：

	时间复杂度	空间复杂度
位图	O(N)	0.625M
多路归并	O(Nlogn)	1M

其中多路归并的时间复杂度为 O（k×n/k×logn/k ），严格来说，还要加上读写磁盘的时间，而此算法绝大部分时间也是浪费在这上面。

6.5 分布式处理之 MapReduce

方法介绍

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，通过大量机器进行分布式并行计算，减少整个操作的时间。也就是说，MapReduce 的原理就是一个归并排序。

它的适用范围为数据量大，但是数据种类小可以放入内存的场景。基本原理及要点是将数据交给不同的机器去处理，数据划分，结果归并。

MapReduce 模式的主要思想是将要执行的问题（如程序）自动拆分成 Map（映射）和 Reduce（化简）的方式，其流程图如图 6-4 所示。

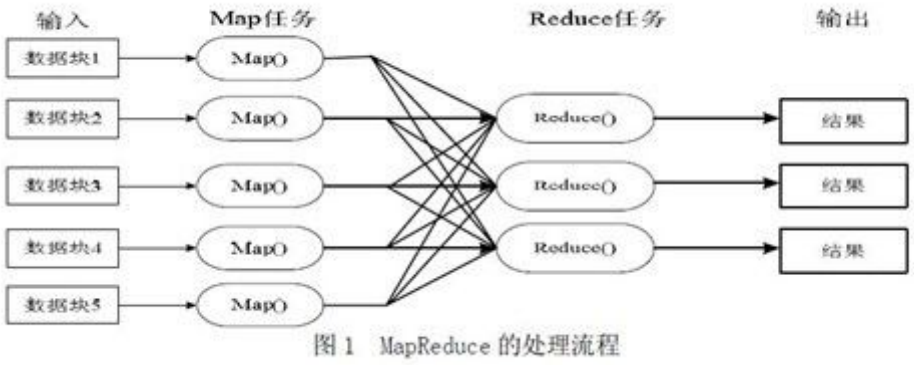

图 1　MapReduce 的处理流程

图 6-4

在数据被分割后通过 Map 函数的程序将数据映射成不同的区块，分配给计算机机群处理达到分布式运算的效果，在通过 Reduce 函数的程序将结果汇整，从而输出开发者需要的结果。

MapReduce 借鉴了函数式程序设计语言的设计思想，其软件实现是指定一个 Map 函数，把键值对(key/value)映射成新的键值对(key/value)，形成一系列中间结果形式的 key/value 对，然后把它们传给 Reduce(规约)函数，把具有相同中间形式 key/value 合并在一起。Map 和 Reduce 函数具有一定的关联性。函数描述如表 6-1 所示：

表 1　Map 函数和 Reduce 函数的描述

函数	输入	输出	说明
Map	<k1,v1>	List(<k2,v2>)	1.将小数据集进一步解析成一批<key,value>对，输入Map函数中进行处理。 2.每一个输入的<k1,v1>会输出一批<k2,v2>。<k2,v2>是计算的中间结果。
Reduce	<k2,List(v2)>	<k3,v3>	输入的中间结果<k2,List(v2)>中的List(v2)表示是一批属于同一个k2的value

表 6-1

MapReduce 致力于解决大规模数据处理的问题，因此在设计之初就考虑了数据的局部性原理，利用局部性原理将整个问题分而治之。MapReduce 集群由普通 PC 机构成，为无共享式架构。在处理之前，将数据集分布至各个节点。处理时，每个节点就近读取本地存储的数据处理（map），将处理后的数据进行合并（combine）、排序（shuffle and sort）后再分发（至 reduce 节点），避免了大量数据的传输，提高了处理效率。无共享式架构的另一个好处是配合复制（replication）策略，集群可以具有良好的容错性，一部分节点的 down 机对集群的正常工作不会造成影响。

Hadoop 是一个实现了 MapReduce 模式的开源分布式并行编程框架。图 6-5 是有关 Hadoop 的作业调优参数及原理，左边是 map 任务运行示意图，右边是 reduce 任务运行示意图。

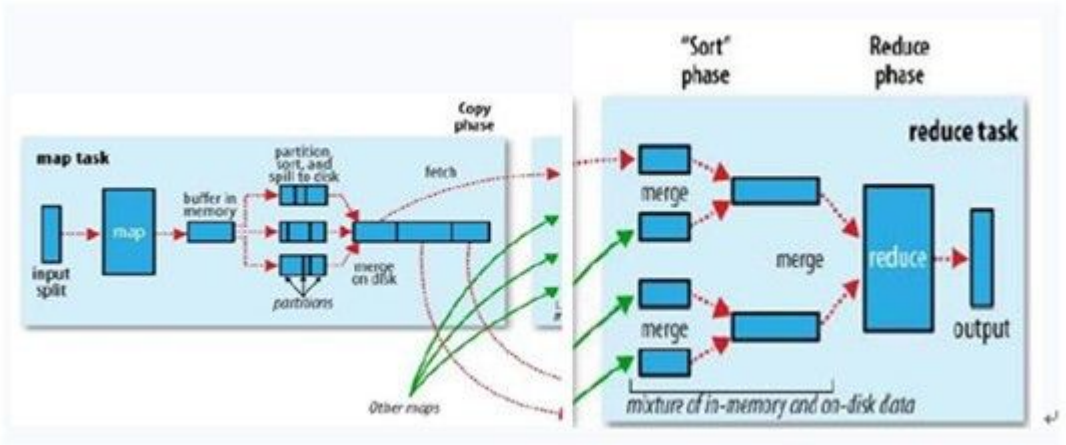

图 6-5　Hadoop 的作业调优参数及原理

如图 2 所示，其中 map 阶段，当 map 任务开始运算，并产生中间数据后并非直接而简单的写入磁盘，它首先利用内存缓冲区来对已经产生的 buffer 进行缓存，并在内存缓存区中进行一些预排序来优化整个 map 的性能。而上图右边的 reduce 阶段则经历了三个阶段，分别 Copy->Sort->Reduce。我们能明显地看出，其中的 Sort 是采用的归并排序（merge sort）。

问题实例

1. TOP 10 的统计

海量数据分布在 100 台电脑中，请问如何高效统计出这批数据的 TOP10。

2. 寻找 N^2 个数的中数

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)?

6.6 多层划分

方法介绍

多层划分法，本质上还是分而治之的思想，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，然后在一个可以接受的范围内进行查找。

问题实例

1. 寻找不重复的数

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

分析：类似于鸽巢原理，因为整数个数为 2^{32}，所以，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，最后不同的区域再利用 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。

2. 寻找中位数

5 亿个 int 找它们的中位数。

分析：首先将 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，之后根据统计结果就可以判断中位数落到哪个区域，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能够接受的程度。即可以先将 int64 分成 2^{24} 个区域，确定区域的第几大数，然后再将该区域分成 2^{20} 个子区域，确定是子区域的第几大数，最后当子区域里的数的个数只有 2^{20} 个时，就可以直接利用直接寻址表 direct addr table 进行统计了。

6.7 位图

方法介绍

什么是位图

所谓的位图（Bit-map）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。

位图通过使用 bit 数组来表示某些元素是否存在，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。

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位图的方法来达到排序的目的。要表示 8 个数，就只需要 8 个 bit（等于 1 字节），首先我们开辟 1 字节的空间，将这些空间的所有 bit 位都置为 0，如图 6-6：

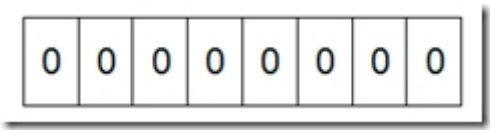

图 6-6

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) | (0×01 << (i % 8)) 当然，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 0 开始的，所以要把第五个位置为一，如图 6-7：

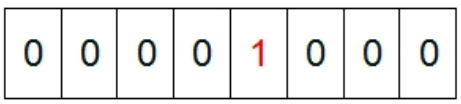

图 6-7

然后再处理第二个元素 7，将第八个位置为 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，如图 6-8：

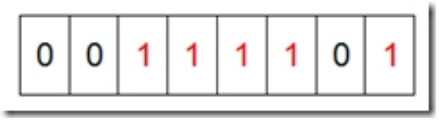

图 6-8

然后我们现在遍历一遍 bit 区域，将该位是 1 的位的编号（2，3，4，5，7）输出，这样就达到了排序的目的。

问题实例

1. 电话号码的统计

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，大概十几兆字节的内存即可。

2. 2.5 亿个数的去重

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

分析：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。

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。”

3. 整数的快速查询

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？

分析：可以用位图/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。

6.8 布隆过滤器

方法介绍

我们经常会碰到这样的问题：判断一个元素是否在一个集合中，常见的做法是用 hashtable 实现集合，然后遇到一个新的元素后，在 hashtable 中查找，如果能找到则存在于集合中，反之不存在。但 hashtable 有一个弊端是耗费空间多大。这节，咱们来看一个新的方法 - Bloom Filter。

布隆过滤器（Bloom Filter）是一种空间效率很高的随机数据结构，它可以看做是对位图（bit-map）的扩展。其原理是：当一个元素被加入集合时，通过 K 个 Hash 函数将这个元素映射成一个位阵列（Bit array）中的 K 个点，并将它们置为 1。

所以在检索一个元素是否在一个集合中时，我们只要看看这些点是不是都是 1 就能大约判断出集合中有没有它了：如果这些点有任何一个 0，则被检索元素一定不在；如果都是 1，则被检索元素很可能存在集合中。

但布隆过滤器有一定的误判率：在判断一个元素是否属于某个集合时，有可能会把不属于这个集合的元素误认为属于这个集合（false positive）。因此，它不适合那些“零错误”的应用场合。而在能容忍低错误率的应用场合下，布隆过滤器通过极少的错误换取了存储空间的极大节省。

1. 集合表示和元素查询

下面我们具体来看布隆过滤器是如何用位数组表示集合的。如图 6-9，初始状态时，布隆过滤器是一个包含 m 位的位数组，每一位都置为 0。

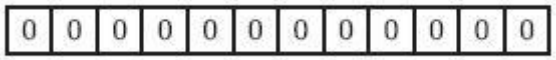

图 6-9

对于 $S=\{x_1, x_2, \dots, x_n\}$这样一个 n 个元素的集合，布隆过滤器使用 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$\{1, \dots, m\}$的范围中。对任意一个元素 x，第 i 个哈希函数映射的位置 $h_i(x)$就会被置为 1（$1 \leqslant i \leqslant k$）。

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。在图 6-10 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）。

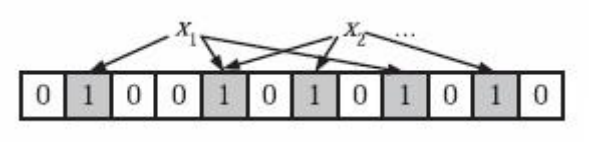

图 6-10

于此，在判断 y 是否属于这个集合时，我们对 y 应用 k 次哈希函数，如果所有 $h_i(y)$的位置都是 1（1≤i≤k），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。例如，在图 6-11 中的 y_1 就不是集合中的元素（因为 y1 有一处指向了“0”位）。而 y_2 可能属于这个集合，也可能刚好是一个 false positive。

图 6-11

2. 错误率估计

前面我们已经提到了，Bloom Filter 在判断一个元素是否属于它表示的集合时会有一定的错误率（false positive rate），下面我们就来估计错误率的大小。

为了简化模型，我们假设 kn < m 且各个哈希函数是完全随机的。每插入一个新元素，第一个哈希函数就会把过滤器中的某个比特位置成 1，因此任意一个比特位被置成 1 的概率为 1/m，反之，它没被置成 1（依然是 0）的概率则是 1 - 1/m。如果这个元素的 k 个哈希函数都没有把某个比特位置成 1，即在做完 kn 次哈希后，某一位还是 0（意味着 k 次哈希都没有选中它）的概率就是（1-1/m）的 k 次方。如果插入第二个元素，某个比特位依然没有被置成 1 的概率为（1-1/m）的 2k 次方，所以如果插入 n 个元素都还没有把某个比特位设置成 1 的概率为$(1-1/m)^{kn}$。

也就是说，当集合 S = {x1, x_2, ⋯, x_n}的所有元素都被 k 个哈希函数映射到 m 位的位数组中时，这个位数组中某一位还是 0 的概率是：

$$p' = \left(1-\frac{1}{m}\right)^{kn} \approx e^{-kn/m}$$

为了简化运算，可以令 $p = e^{-kn/m}$，则有：

$$\lim_{x\to\infty}\left(1-\frac{1}{x}\right)^{-x} = e$$

令 ρ 为位数组中 0 的比例，则 ρ 的数学期望 E(ρ)= p'。在 ρ 已知的情况下，要求的错误率（false positive rate）为：

$$(1-\rho)^k \approx (1-p')^k \approx (1-p)^k$$

(1- ρ)为位数组中 1 的比例，(1- ρ)^k 就表示 k 次哈希都刚好选中 1 的区域，即 false positive rate。上式中第二步近似在前面已经提到了，现在来看第一步近似。p'只是 ρ 的数学期望，在实际中 ρ 的值有可能偏离它的数学期望值。M. Mitzenmacher 已经证明，位数组中 0 的比例非常集中地分布在它的数学期望值的附近。因此，第一步的近似得以成立。分别将 p 和 p'代入上式中，得：

$$f' = \left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^k = (1-p')^k$$

$$f = \left(1-e^{-kn/m}\right)^k = (1-p)^k$$

相比 p’和 f’，使用 p 和 f 通常在分析中更为方便。

3. 最优的哈希函数个数

既然布隆过滤器要靠多个哈希函数将集合映射到位数组中，那么应该选择几个哈希函数才能使元素查询时的错误率降到最低呢？这里有两个互斥的理由：如果哈希函数的个数多，那么在对一个不属于集合的元素进行查询时得到 0 的概率就大；但另一方面，如果哈希函数的个数少，那么位数组中的 0 就多。为了得到最优的哈希函数个数，我们需要根据上一小节中的错误率公式进行计算。

先用 p 和 f 进行计算。注意到 f = exp(k ln(1 − e$^{-kn/m}$))，我们令 g = k ln(1 − e$^{-kn/m}$)，只要让 g 取到最小，f 自然也取到最小。由于 p = e$^{-kn/m}$，我们可以将 g 写成

$$g = -\frac{m}{n}\ln(p)\ln(1-p)$$

根据对称性法则可以很容易看出当 p = 1/2，也就是 k = ln2 • (m/n)时，g 取得最小值。在这种情况下，最小错误率 f 等于(1/2)^k≈ (0.6185)$^{m/n}$。另外，注意到 p 是位数组中某一位仍是 0 的概率，所以 p = 1/2 对应着位数组中 0 和 1 各一半。换句话说，要想保持错误率低，最好让位数组有一半还空着。

需要强调的一点是，p = 1/2 时错误率最小这个结果并不依赖于近似值 p 和 f。同样对于 f' = exp(k ln(1 − (1 − 1/m)^{kn}))，g' = k ln(1 − (1 − 1/m)^{kn})，p' = (1 − 1/m)^{kn}，我们可以将 g'写成

$$g' = \frac{1}{n\ln(1-1/m)}\ln(p')\ln(1-p')$$

同样根据对称性法则可以得到当 p' = 1/2 时，g'取得最小值。

4. 位数组的大小

下面我们来看看，在不超过一定错误率的情况下，布隆过滤器至少需要多少位才能表示全集中任意 n 个元素的集合。假设全集中共有 u 个元素，允许的最大错误率为ϵ，下面我们来求位数组的位数 m。

假设 X 为全集中任取 n 个元素的集合，F(X)是表示 X 的位数组。那么对于集合 X 中任意一个元素 x，在 s = F(X)中查询 x 都能得到肯定的结果，即 s 能够接受 x。显然，由于布隆过滤器引入了错误，s 能够接受的不仅仅是 X 中的元素，它还能够ϵ (u - n)个 false positive。因此，对于一个确定的位数组来说，它能够接受总共 n + ϵ (u - n)个元素。在 n + ϵ (u - n)个元素中，s 真正表示的只有其中 n 个，所以一个确定的位数组可以表示$C_{n+\epsilon(u-n)}^{n}$个集合。m 位的位数组共有 2^m 个不同的组合，进而可以推出，m 位的位数组可以表示$2^m C_{n+\epsilon(u-n)}^{n}$个集合。全集中 n 个元素的集合总共有$C_u^n$个，因此要让 m 位的位数组能够表示所有 n 个元素的集合，必须有

$$2^m C_{n+\epsilon(u-n)}^{n} \geq C_u^n$$

即：

$$m \geq \log_2 \frac{C_u^n}{C_{n+\epsilon(u-n)}^{n}} \approx \log_2 \frac{C_u^n}{C_{\epsilon u}^{n}} \geq \log_2 \epsilon^{-n} = n\log_2(1/\epsilon)$$

上式中的近似前提是 n 和ϵu 相比很小，这也是实际情况中常常发生的。根据上式，我们得出结论：在错误率不大于ϵ的情况下，m 至少要等于 n $\log_2(1/\epsilon)$才能表示任意 n 个元素的集合。

上一小节中我们曾算出当 k = ln2 • (m/n)时错误率 f 最小，这时 f = $(1/2)^k$= $(1/2)^{m\ln 2/n}$。现在令 f≤ϵ，可以推出

$$m \geq n\frac{\log_2(1/\epsilon)}{\ln 2} = n\log_2\log_2(1/\epsilon)$$

这个结果比前面我们算得的下界 n $\log_2(1/\epsilon)$大了 $\log_2 e \approx 1.44$ 倍。这说明在哈希函数的个数取到最优时，要让错误率不超过ϵ，m 至少需要取到最小值的 1.44 倍。

布隆过滤器可以用来实现数据字典，进行数据的判重，或者集合求交集。

问题实例

1. 寻找通过 URL

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。

分析：如果允许有一定的错误率，可以使用布隆过滤器，4G 内存大概可以表示 340 亿 bit。将其中一个文件中的 url 使用布隆过滤器映射为这 340 亿 bit，然后挨个读取另外一个文件的 url，检查是否与布隆过滤器，如果是，那么该 url 应该是共同的 url。

此外，如果是三个乃至 n 个文件呢？欢迎读者继续思考。

2. 垃圾邮件过滤

用过邮箱的朋友都知道，经常会受到各种垃圾邮件，可能是广告，可能是病毒，所以邮件提供商每天都需要过滤数以几十亿的垃圾邮件，请想一个办法过滤这些垃圾邮件。

分析：比较直观的想法是把常见的垃圾邮件地址存到一个巨大的集合中，然后遇到某个新的邮件，将它的地址和集合中的全部垃圾邮件地址一一进行比较，如果有元素与之匹配，则判定新邮件为垃圾邮件。

虽然本节开头，我们提到集合可以用 hashtable 实现，但它太占空间。比如存储一亿个 Email 地址，就得需要 1.6G 内存，从而存储几十亿个 Email 地址，则可能需要上百 G 的内存，虽然现在有的机器内存达到了上百 G，但终究是少数机器。

事实上，如果允许一定的误判率的话，我们可以使用布隆过滤器。解决了存储的问题后，可以利用贝叶斯分类鉴别一份邮件是否为垃圾邮件，减少误判率。

6.9 Trie 树

方法介绍

1. 什么是 Trie 树

Trie 树，即字典树，又称单词查找树或键树，是一种树形结构。典型应用是用于统计和排序大量的字符串（但不仅限于字符串），所以经常被搜索引擎系统用于文本词频统计。它的优点是最大限度地减少无谓的字符串比较，查询效率比较高。

Trie 的核心思想是空间换时间，利用字符串的公共前缀来降低查询时间的开销以达到提高效率的目的。

它有 3 个基本性质：

1. 根节点不包含字符，除根节点外每一个节点都只包含一个字符。
2. 从根节点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串。
3. 每个节点的所有子节点包含的字符都不相同。

2. 树的构建

咱们先来看一个问题：假如现在给你 10 万个长度不超过 10 的单词，对于每一个单词，我们要判断它出没出现过，如果出现了，求第一次出现在第几个位置。对于这个问题，我们该怎么解决呢？

如果我们用最笨拙的方法，对每一个单词，都去查找它前面的单词中是否有它。那么这个算法的复杂度就是 $O(n^2)$。显然对于 10 万的范围难以接受。

换个思路想：假设我要查询的单词是 abcd，那么在它前面的单词中，以 b，c，d，f 之类开头的显然不必考虑，而只要找以 a 开头的单词中是否存在 abcd 就可以了。

同样的，在以 a 开头的单词中，我们只要考虑以 b 作为第二个字母的，一次次缩小范围和提高针对性，这样一个树的模型就渐渐清晰了。

即如果现在有 b，abc，abd，bcd，abcd，efg，hii 这 6 个单词，我们可以构建一棵如图 6-12 所示的树：

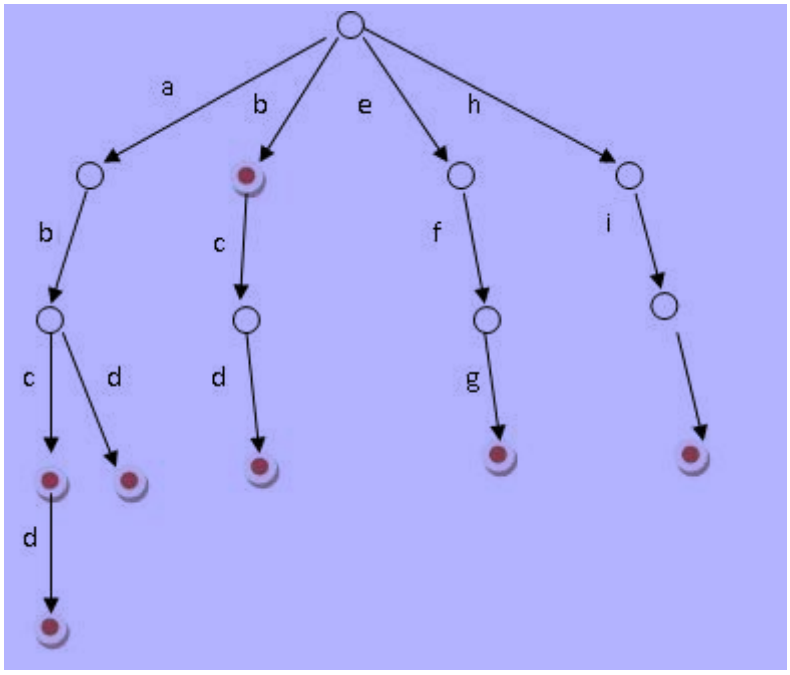

图 6-12

如上图所示，对于每一个节点，从根遍历到它的过程就是一个单词，如果这个节点被标记为红色，就表示这个单词存在，否则不存在。

那么，对于一个单词，只要顺着他从根走到对应的节点，再看这个节点是否被标记为红色就可以知道它是否出现过了。把这个节点标记为红色，就相当于插入了这个单词。

这样一来我们查询和插入可以一起完成，所用时间仅仅为单词长度（在这个例子中，便是 10）。这就是一棵 Trie 树。

我们可以看到，Trie 树每一层的节点数是 26^i 级别的。所以为了节省空间，我们还可以用动态链表，或者用数组来模拟动态。而空间的花费，不会超过单词数×单词长度。

3. 查询

Trie 树是简单且实用的数据结构，通常用于实现字典查询。我们做即时响应用户输入的 AJAX 搜索框时，就是以 Trie 开始。本质上，Trie 是一颗存储多个字符串的树。相邻节点间的边代表一个字符，这样树的每条分支代表一则子串，而树的叶节点则代表完整的字符串。和普通树不同的地方是，相同的字符串前缀共享同一条分支。

下面，再举一个例子。给出一组单词，inn, int, ate, age, adv, ant, 我们可以得到如图 6-13 所示的 Trie 树：

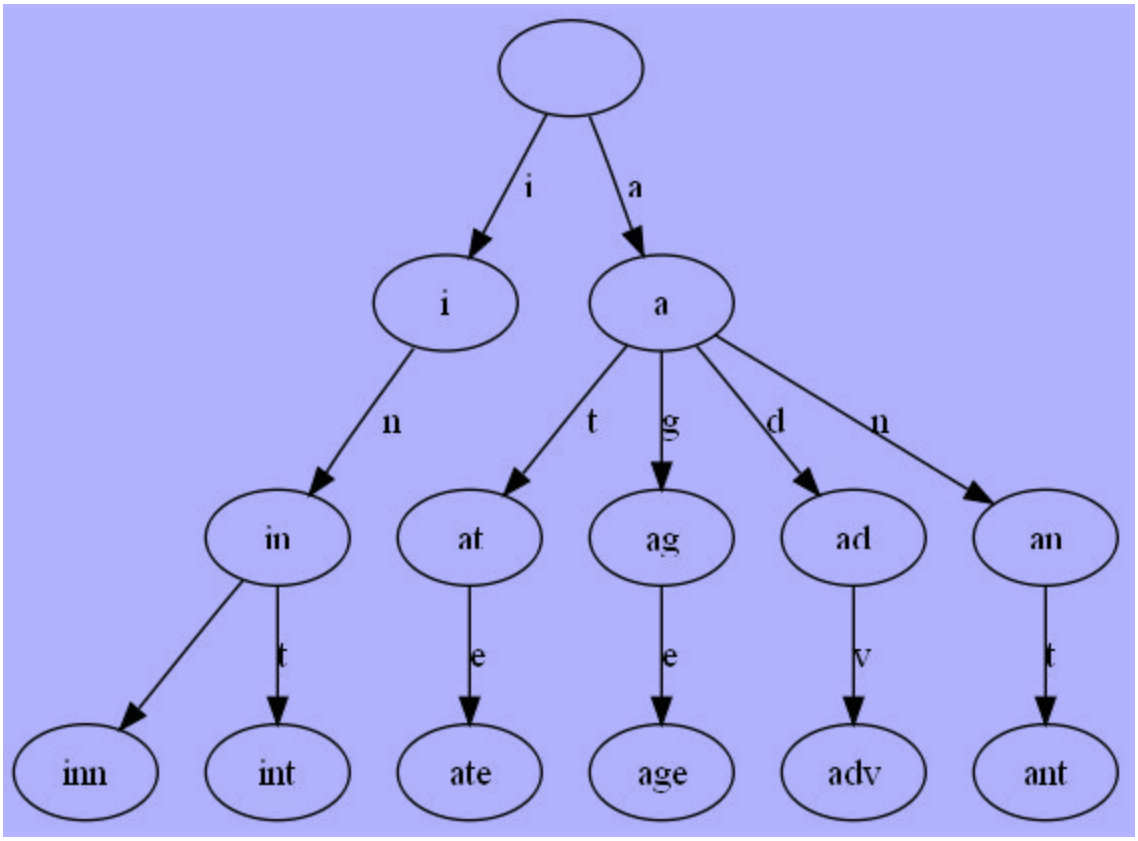

图 6-13

可以看出：

- 每条边对应一个字母。
- 每个节点对应一项前缀。叶节点对应最长前缀，即单词本身。
- 单词 inn 与单词 int 有共同的前缀“in”，因此他们共享左边的一条分支，root->i->in。同理，ate, age, adv, 和 ant 共享前缀"a"，所以他们共享从根节点到节点"a"的边。

查询操纵非常简单。比如要查找 int，顺着路径 i -> in -> int 就找到了。

搭建 Trie 的基本算法也很简单，无非是逐一把每则单词的每个字母插入 Trie。插入前先看前缀是否存在。如果存在，就共享，否则创建对应的节点和边。比如要插入单词 add，就有下面几步：

1. 考察前缀"a"，发现边 a 已经存在。于是顺着边 a 走到节点 a。
2. 考察剩下的字符串"dd"的前缀"d"，发现从节点 a 出发，已经有边 d 存在。于是顺着边 d 走到节点 ad
3. 考察最后一个字符"d"，这下从节点 ad 出发没有边 d 了，于是创建节点 ad 的子节点 add，并把边 ad->add 标记为 d。

问题实例

1. 10 个频繁出现的词

一个文本文件，大约有一万行，每行一个词，要求统计出其中出现次数最频繁的 10 个词，请给出思路和时间复杂度的分析。

分析：用 trie 树统计每个词出现的次数，时间复杂度是 O(n×le)（le 表示单词的平均长度），然后是找出出现最频繁的前 10 个词。当然，也可以用堆来实现，时间复杂度是 O(n×lg10)。所以总的时间复杂度，是 O(n×le)与 O(n×lg10)中较大的哪一个。

2. 寻找热门查询

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，但是如果去除重复和，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。

分析：利用 Trie 树，观察关键字域存该查询串出现的次数，若没有出现则为 0。最后用 10 个元素的最小推来对出现频率进行排序。

6.10 数据库

方法介绍

当遇到大数据量的增删改查时，一般把数据装进数据库中，从而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。

而数据库索引的建立则对查询速度起着至关重要的作用。

Hash 索引，实际上就是通过一定的 Hash 算法，将须要索引的键值进行 Hash 运算，然后将得到的 Hash 值存入 Hash 表中。检索时，根据 Hash 表中的 Hash 值逆 Hash 运算反馈原键值。Hash 索引在 MySQL 中使用并不多，目前在 Memory 和 NDB Cluster 存储引擎使用。

在本书的第 3 章，我们介绍了 B 树、B+树等索引，事实上，InnoDB 存储引擎的 B-Tree 索引使用的存储结构就是 B+树。B+树在 B 树的基础上做了很小的改造，在每一个 LeafNod 上除了存放索引键的相关信息，还存储了指向与该 LeafNode 相邻的后一个 LeafNode 的指针，此举是为了加快检索多个相邻 LeafNode 的效率；

换言之，B+树的叶子节点中除了跟 B 树一样包含了关键字的信息之外，还包含了指向相邻叶子节点的指针，如此，叶子节点之间就有了联系、有序了。而 B*树则更进一筹，增加了兄弟节点之间的指针。

无处不透露着数据结构与算法思想，数据库也不例外。尤其当涉及到数据库性能优化，则更是如此。

问题实例

索引的选择

我们知道，Hash 索引的效率比 B-Tree 高很多，但为什么大家都不用 Hash 索引而要使用 B-Tree 索引呢？你能说出几个原因呢？

6.11 倒排索引

方法介绍

倒排索引（(Inverted index)）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，常被应用于搜索引擎和关键字查询的问题中。

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于存储每个文档的单词的列表。正向索引的查询能够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。

问题实例

文档检索系统

请设计一个文档检索系统，用于查询哪些文件包含了某单词，比如常见的学术论文的关键字搜索。

提示：建倒排索引。