特别加餐 | 倒排检索加速（一）：工业界如何利用跳表、哈希表、位图进行加速？

2020-04-06 陈东 来自北京

《检索技术核心20讲》

你好，我是陈东。欢迎来到检索专栏的第一次加餐时间。

很多同学在留言区提问，说基础篇讲了这么多检索的基础数据结构和算法，那它们在工业界的实际系统中是怎么应用的呢？真正的检索系统和算法又是什么样的呢？

为了帮助你把这些基础的知识，更好地和实际应用结合。我特别准备了两篇加餐，来和你一起聊一聊，这些看似简单的基础技术是怎样在工业界的实际系统中发挥重要作用的。

在许多大型系统中，倒排索引是最常用的检索技术，搜索引擎、广告引擎、推荐引擎等都是基于倒排索引技术实现的。而在倒排索引的检索过程中，两个 posting list 求交集是一个最重要、最耗时的操作。

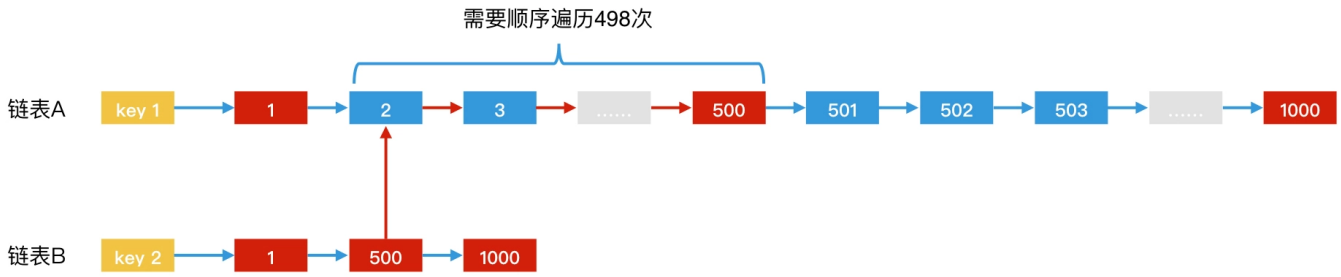
所以，今天我们就先来看一看，倒排索引在求交集的过程中，是如何借助跳表、哈希表和位图，这些基础数据结构进行加速的。

跳表法加速倒排索引

在第 5 讲中我们讲过，倒排索引中的 posting list 一般是用链表来实现的。当两个 posting list A 和 B 需要合并求交集时，如果我们用归并法来合并的话，时间代价是 O(m+n)。其中，m 为 posting list A 的长度，n 为 posting list B 的长度。

那对于这个归并过程，工业界是如何优化的呢？接下来，我们就通过一个例子来看一下。

假设 posting list A 中的元素为 <1,2,3,4,5,6,7,8……，1000>，这 1000 个元素是按照从 1 到 1000 的顺序递增的。而 posting list B 中的元素，只有 <1,500,1000>3 个。那按照我们之前讲过的归并方法，它们的合并过程就是，在找到相同元素 1 以后，还需要再遍历 498 次链表，才能找到第二个相同元素 500。

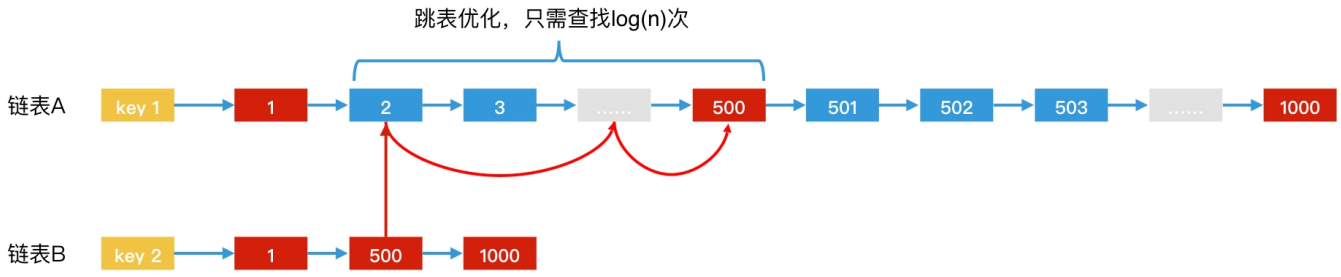


极客时间

链表遍历，时间代价高

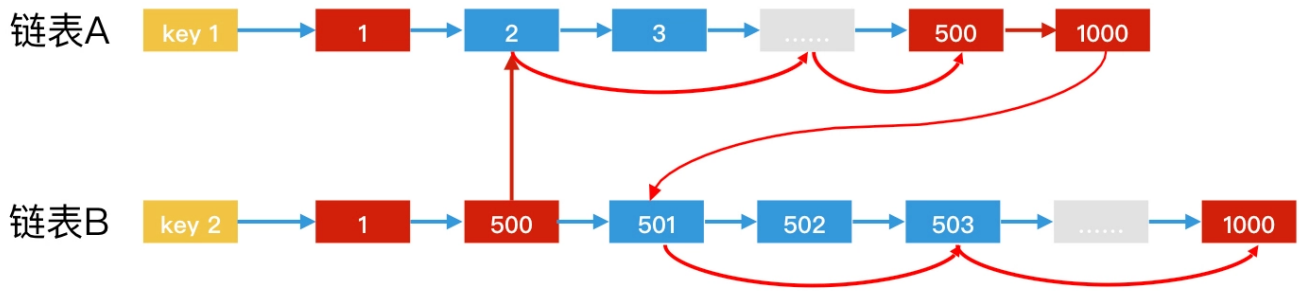
很显然，为了找到一个元素，遍历这么多次是很不划算的。那对于链表遍历，我们可以怎么优化呢？实际上，在许多工业界的实践中，比如搜索引擎，还有 Lucene 和 Elasticsearch 等应用中，都是使用跳表来实现 posting list 的。

在上面这个例子中，我们可以将链表改为跳表。这样，在 posting list A 中，我们从第 2 个元素遍历到第 500 个元素，只需要 log(498) 次的量级，会比链表快得多。



跳表查找，检索加速

这个时候你可能就会问了，我们只能用 B 中的每一个元素去 A 中二分查找吗？那在解答这个问题之前，我们先来看下图这个例子。



相互二分查找

你会发现，在寻找 500 这个公共元素的过程中，我们是拿着链表 B 中的 500 作为 key，在链表 A 中进行跳表二分查找的。但是，在查找 1000 这个公共元素的过程中，我们是拿着链表 A 中的元素 1000，在链表 B 中进行跳表二分查找的。

我们把这种方法定义为**相互二分查找**。那啥叫相互二分查找呢？

你可以这么理解：如果 A 中的当前元素小于 B 中的当前元素，我们就以 B 中的当前元素为 key，在 A 中快速往前跳；如果 B 中的当前元素小于 A 中的当前元素，我们就以 A 中的当前元素为 key，在 B 中快速往前跳。这样一来，整体的检索效率就提升了。

在实际的系统中，如果 posting list 可以都存储在内存中，并且变化不太频繁的话，那我们还可以利用**可变长数组**来代替链表。

可变长数组的数组的长度可以随着数据的增加而增加。一种简单的可变长数组实现方案就是当数组被写满时，我们直接重新申请一个 2 倍于原数组的新数组，将原数组数据直接导入新数组中。这样，我们就可以应对数据动态增长的场景。

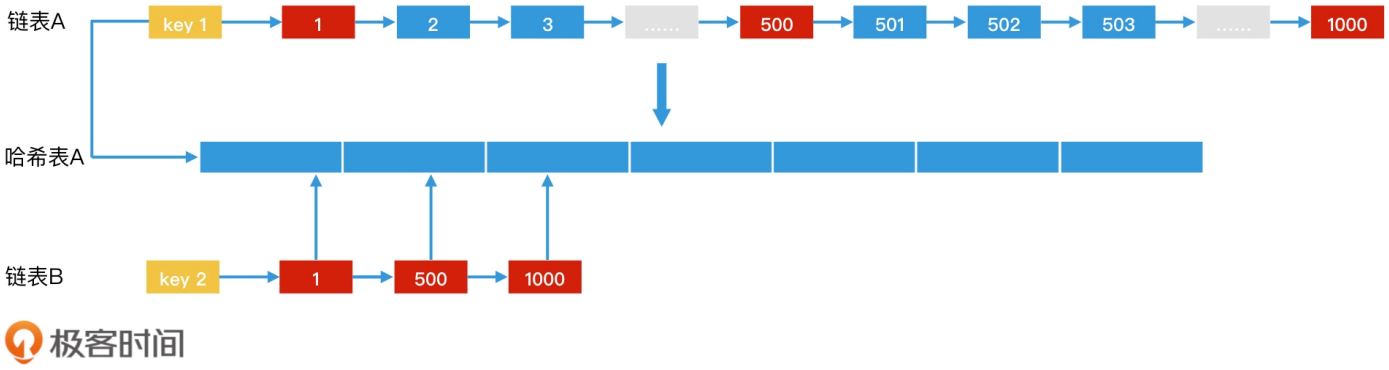
那对于两个 posting list 求交集，我们同样可以先使用可变长数组，再利用**相互二分查找**进行归并。而且，由于数组的数据在内存的物理空间中是紧凑的，因此 CPU 还可以利用内存的局部性原理来提高检索效率。

哈希表法加速倒排索引

说到高效查询，哈希表 O(1) 级别的查找能力令人印象深刻。那我们有没有能利用哈希表来加速的方法呢？别说，还真有。

哈希表加速的思路其实很简单，就是当两个集合要求交集时，如果一个集合特别大，另一个集合相对比较小，那我们就可以用哈希表来存储大集合。这样，我们就可以拿着小集合中的每一个元素去哈希表中对比：如果查找为存在，那查找的元素就是公共元素；否则，就放弃。

我们还是以前面说的 posting list A 和 B 为例，来进一步解释一下这个过程。由于 Posting list A 有 1000 个元素，而 B 中只有 3 个元素，因此，我们可以将 posting list A 中的元素提前存入哈希表。这样，我们利用 B 中的 3 个元素来查询的时候，每次查询代价都是 O(1)。如果 B 有 m 个元素，那查询代价就是 O(m)。



将 posting list A 中的元素提前存入哈希表

但是，使用哈希表法加速倒排索引有一个前提，就是我们要在查询发生之前，就把 posting list 转为哈希表。这就需要我们提前分析好，哪些 posting list 经常会被拿来求交集，针对这一批 posting list，我们将它们提前存入哈希表。这样，我们就能实现检索加速了。

这里还有一点需要你注意，原始的 posting list 我们也要保留。这是为什么呢？

我们假设有这样一种情况：当我们要给两个 posting list 求交集时，发现这两个 posting list 都已经转为哈希表了。这个时候，由于哈希表没有遍历能力，反而会导致我们无法合并这两个 posting list。因此，在哈希表法的最终改造中，一个 key 后面会有两个指针，一个指向 posting list，另一个指向哈希表（如果哈希表存在）。

除此之外，哈希表法还需要在有很多短 posting list 存在的前提下，才能更好地发挥作用。这是因为哈希表法的查询代价是 O(m)，如果 m 的值很大，那它的性能就不一定会比跳表法更优了。

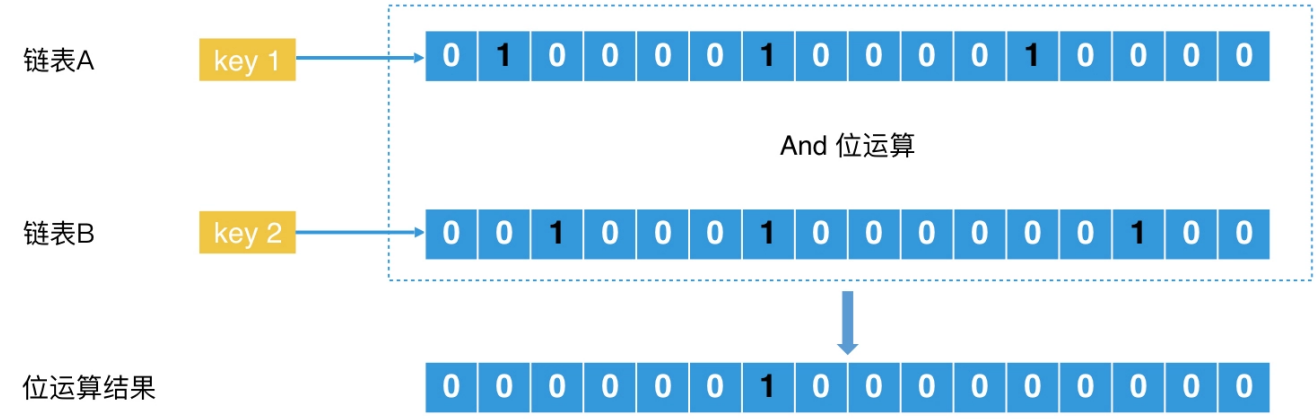
位图法加速倒排索引

我们知道，位图其实也可以看作是一种特殊的哈希，所以除了哈希表，我们还可以使用位图法来改造链表。如果我们使用位图法，就需要将所有的 posting list 全部改造为位图，这样才能使用位图的位运算来进行检索加速。那具体应该怎么做呢？我们一起来看一下。

首先，我们需要为每个 key 生成同样长度的位图，表示所有的对象空间。然后，如果一个元素出现在该 posting list 中，我们就将位图中该元素对应的位置置为 1。这样就完成了 posting list 的位图改造。

接下来，我们来看一下位图法的查询过程。

如果要查找 posting list A 和 B 的公共元素，我们可以将 A、B 两个位图中对应的位置直接做 and 位运算（复习一下 and 位运算：0 and 0 = 0； 0 and 1 = 0； 1 and 1 = 1）。由于位图的长度是固定的，因此两个位图的合并运算时间代价也是固定的。并且由于 CPU 执行位运算的效率非常快，因此，在位图总长度不是特别长的情况下，位图法的检索效率还是非常高的。



位运算

和哈希表法一样，位图法也有自己的局限性。我总结了以下 3 点，你可以感受一下。

1. 位图法仅适用于只存储 ID 的简单的 posting list。如果 posting list 中需要存储复杂的对象，就不适合用位图来表示 posting list 了。
2. 位图法仅适用于 posting list 中元素稠密的场景。对于 posting list 中元素稀疏的场景，使用位图的运算和存储开销反而会比使用链表更大。

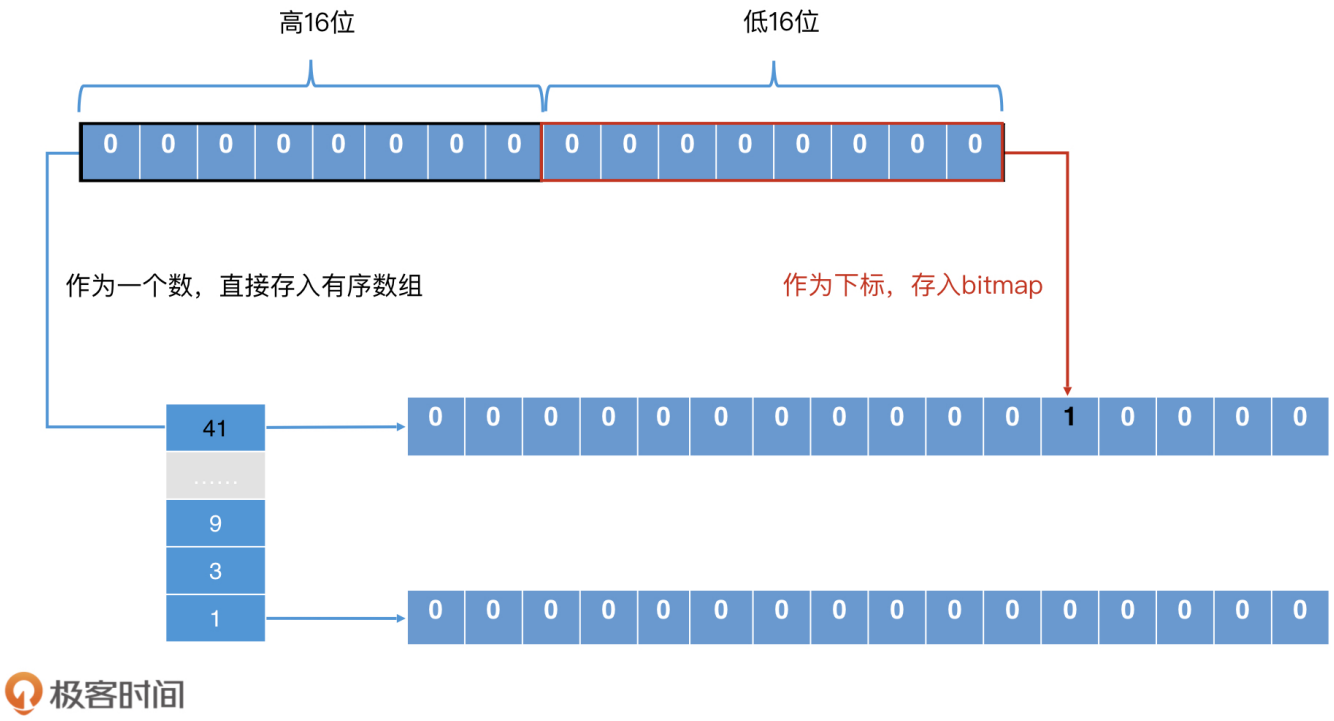
3. 位图法会占用大量的空间。尽管位图仅用 1 个 bit 就能表示元素是否存在，但每个 posting list 都需要表示完整的对象空间。如果 ID 范围是用 int32 类型的数组表示的，那一个位图的大小就约为 512M 字节。如果我们有 1 万个 key，每个 key 都存一个这样的位图，那就需要 5120G 的空间了。这是非常可怕的空间开销啊！

在很多成熟的工业界系统中，为了解决位图的空间消耗问题，我们经常会使用一种压缩位图的技术 Roaring Bitmap 来代替位图。在数据库、全文检索 Lucene、数据分析 Druid 等系统中，你都能看到 Roaring Bitmap 的身影。

升级版位图：Roaring Bitmap

下面我们来学习一下 Roaring Bitmap 的设计思想。

首先，Roaring Bitmap 将一个 32 位的整数分为两部分，一部分是高 16 位，另一部分是低 16 位。对于高 16 位，Roaring Bitmap 将它存储到一个有序数组中，这个有序数组中的每一个值都是一个“桶”；而对于低 16 位，Roaring Bitmap 则将它存储在一个 2^16 的位图中，将相应位置置为 1。这样，每个桶都会对应一个 2^16 的位图。



Roaring Bitmap 存储过程

接下来，如果我们要确认一个元素是否在 Roaring Bitmap 中存在，通过两次查找就能确认了。第一步是以高 16 位在有序数组中二分查找，看对应的桶是否存在。如果存在，第二步就是将桶中的位图取出，拿着低 16 位在位图中查找，判断相应位置是否为 1。第一步查找由于是数组二分查找，因此时间代价是 O（log n）；第二步是位图查找，因此时间代价是 O(1)。

所以你看，这种将**有序数组和位图用倒排索引结合起来的设计思路，是**能够保证高效检索的。那它到底是怎么节省存储空间的呢？

我们来看一个极端的例子。

如果一个 posting list 中，所有元素的高 16 位都是相同的，那在有序数组部分，我们只需要一个 2 个字节的桶（注：每个桶都是一个 short 型的整数，因此只有 2 个字节。如果数组提前分配好了 2^16 个桶，那就需要 128K 字节的空间，因此使用可变长数组更节省空间）。在低 16 位部分，因为位图长度是固定的，都是 2^16 个 bit，那所占空间就是 8K 个字节。

同样都是 32 位的整数，这样的空间消耗相比于我们在位图法中计算的 512M 字节来说，大大地节省了空间！

你会发现，相比于位图法，这种设计方案就是通过，**将不存在的桶的位图空间全部省去这样的方式，来节省存储空间的。**而代价就是将高 16 位的查找，从位图的 O(1) 的查找转为有序数组的 log(n) 查找。

那每个桶对应的位图空间，我们是否还能优化呢？

前面我们说过，当位图中的元素太稀疏时，其实我们还不如使用链表。这个时候，链表的计算更快速，存储空间也更节省。Roaring Bitmap 就基于这个思路，对低 16 位的位图部分进行了优化：如果一个桶中存储的数据少于 4096 个，我们就不使用位图，而是直接使用 short 型的有序数组存储数据。同时，我们使用可变长数组机制，让数组的初始化长度是 4，随着元素的增加再逐步调整数组长度，上限是 4096。这样一来，存储空间就会低于 8K，也就小于使用位图所占用的存储空间了。

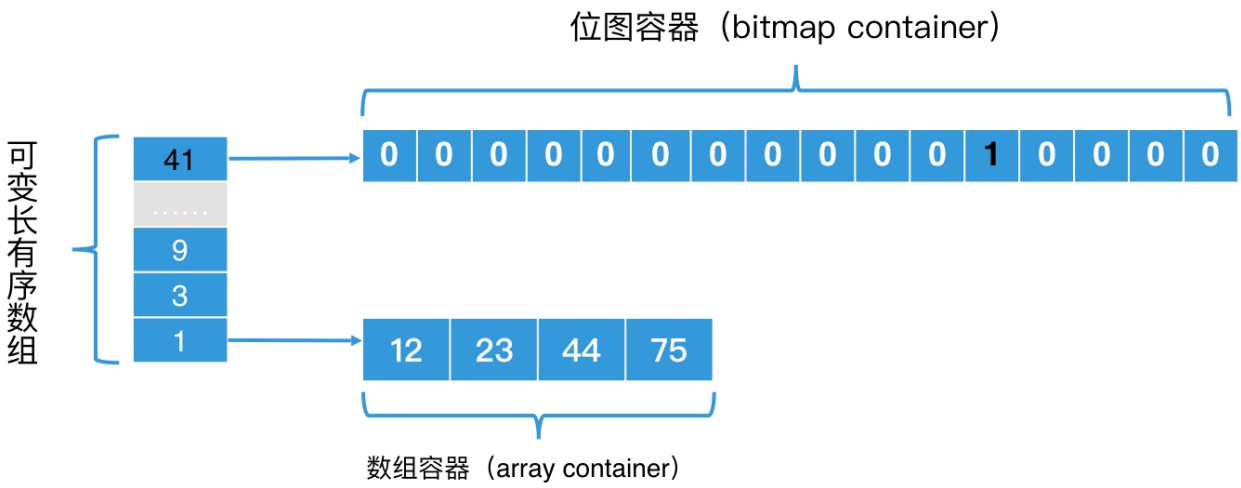
总结来说，一个桶对应的存储容器有两种，分别是数组容器和位图容器（其实还有一种压缩的 runContainer，它是对连续元素通过只记录初始元素和后续个数。由于它不是基础类型，需要手动调用 runOptimize() 函数才会启用，这里就不展开说了）。那在实际应用的过程中，数组容器和位图容器是如何转换的呢？这里有三种情况。

第一种，在一个桶中刚插入数据时，因为数据量少，所以我们就默认使用**数组容器**；

第二种，随着数据插入，桶中的数据不断增多，当数组容器中的元素个数大于 4096 个时，就从数组容器转为**位图容器**；

第三种，随着数据的删除，如果位图容器中的元素个数小于 4096 个，就退化回**数组容器**。

这个过程是不是很熟悉？没错，这很像第 3 节中的 Hashmap 的处理方法。



使用数组容器和位图容器并可以相互转换

好了，前面我们说了这么多 Roaring Bitmap 的压缩位图空间的设计思路。下面，我们回到两个集合 A 和 B 快速求交集的例子中，一起来看一看 Roaring Bitmap 是怎么做的。假设，这里有 Roaring Bitmap 表示的两个集合 A 和 B，那我们求它们交集的过程可以分为 2 步。

第 1 步，比较高 16 位的所有桶，也就是对这两个有序数组求交集，所有相同的桶会被留下来。

第 2 步，对相同的桶里面的元素求交集。这个时候会出现 3 种情况，分别是位图和位图求交集、数组和数组求交集、位图和数组求交集。

其中，位图和位图求交集，我们可以直接使用位运算；数组和数组求交集，我们可以使用相互二分查找（类似跳表法）；位图和数组求交集，我们可以通过遍历数组，在位图中查找数组中的每个元素是否存在（类似哈希表法）。这些方法我们前面都讲过了，那知道了方法，具体怎么操作就是很容易的事情了，你可以再自己尝试一下。

重点回顾

好了，今天的内容讲完了。我们来总结一下，你要掌握的重点内容。

在工业界，我们会利用跳表法、哈希表法和位图法，对倒排索引进行检索加速。

其中，跳表法是将实现倒排索引中的 posting list 的链表改为了跳表，并且使用相互二分查找来提升检索效率；哈希表法就是在有很多短 posting list 存在的前提下，将大的 posting list 转为哈希表，减少查询的时间代价；位图法是在位图总长度不是特别长的情况下，将所有的 posting list 都转为位图，它们进行合并运算的时间代价由位图的长度决定。

并且我们还介绍了位图的升级版本，Roaring Bitmap。很有趣的是，你会发现 Roaring Bitmap 求交集过程的设计实现，本身就是跳表法、哈希表法和位图法的一个综合应用案例。

最后呢，我还想再多说两句。实际上，我写这篇文章就是想告诉你，基础的数据结构和算法组合在一起，就能提供更强大的检索能力，而且这也是大量的工程系统中广泛使用的设计方案。因此，深入理解每一种基础数据结构和算法的特点和适用场景，并且能将它们灵活应用，这能帮助你更好地学习和理解复杂的数据结构和算法，以及更好地学会如何设计复杂的高性能检索系统。

课堂讨论

最后，我们还是来看一道讨论题。

在 Roaring Bitmap 的求交集过程中，有位图和位图求交集、数组和数组求交集、位图和数组求交集这 3 种场景。那它们求交集以后的结果，我们是应该用位图来存储，还是用数组来存储呢？

欢迎在留言区畅所欲言，说出你的思考过程和最终答案。如果有收获，也欢迎把这篇文章分享给你的朋友。

精选留言 (20)

每天晒白牙
2020-04-06

思考题
1.位图和位图求交集
要对两个位图的交集做预判，如果预判数据大于 4096 就用位图，如果小于 4096 就用数组，当然预判肯定会有误判率，不过没关系，即使误判错多做一次转换就行了

2.数组和数组求交集
数据和数据求交集结果肯定还是用数组存

3.位图和数组求交集
位图和数组求交集也是用数组

问题请教：
对 Roaring Bitmap 这儿看的不是很明白，也不知道自己哪里不明白，可能就是不明白吧，请问老师这块有啥好的学习方法吗？
还有就是对于 Lucene 采用 RoaringDocIdSet 实现的 Roaring Bitmap，要想学好这里，是不是还要学些相关的源码呢？如果要学 ES 和 Lucene 的源码，老师有啥好的建议吗？

作者回复: 讨论题思考得很清楚！
关于roaring bitmap（简写RBM），你其实可以把它看做是一个倒排索引。对于一个32位的数，RBM

将这个数的高16位当做key，然后将这个数存入对应的posting list中。posting list用位图表示（长度为2^16）。这样思考，会不会更好理解一些?

这也是我说的学习知识点要多对比，多拆解。

至于如何学好es和lucene的源码，一般来说有两种学习方式:一种是你先学好一个高效检索引擎的各种核心技术，然后这时候你去看es和Lucene的代码，你就会发现，其实这些代码就是为了实现这些设计而写的；另一种方式呢，就是从代码出发，遇到不明白的地方再去查资料，去弄明白为什么要这么实现。

这两种方式，我个人比较倾向先了解了大致原理，再去看代码。这样往往效率会更高。其实就像你自己写代码，先设计好以后再实现会更高效一样。

共 2 条评论 >　　15

一步 酥

2020-04-06

思考题：

数组和数组求交集、位图和数组求交集 这两种情况可以很容易的想到是使用数组

这里解释一下 位图与位图交集的预判的情况，一般是怎么进行预判的：

假设位图1有 n1 个值， 位图1 有 n2 个值，位图的空间位 2 ** 16 = 65536

这里进行预判的时候可以认为是均匀分布的：

那么对于位图1 可以认为间隔 65536 / n1 个位有个值，位图2 可以认为间隔 65536 / n2个位有个值，

那么同时存在 n1和n2 的间隔为 t = （ 65536 / n1 ） * （65536 / n2），那么交集出来的个数为

m = 65536 / t = n1 * n2 / 65536 , 载拿 m 和 4096 进行比较 预判即可

作者回复: 分析得很清晰！在系统面临多种后续情况时，一种常见的处理方式就是预判一种概率最大的情况，先按这种情况处理。这样系统的整体统计性能会最好。

共 5 条评论 >　　7

一步 酥

2020-04-06

仔细算了 一下 Roaring bitmap 压缩后使用的空间，发现压缩率非常大

在一个正常的 32位的 bitmap 占的空间位 2 ** 32 bit ---> 2 ** 29 byte ---> 2 ** 19k---> 2 ** 9 M 也就是512 M

在使用Roaring bitmap 后一个键位图占的空间位（不考虑高16位的数组空间动态申请，和底

16位使用数组存储）： 提前申请好高16位的空间为 2 ** 16 * 2 byte = 2 ** 17 byte --> 2 ** 7 k, 一个位图的空间为 2 ** 16 bit --> 2 ** 13 byte --> 2 ** 3 k， 所以需要的总空间位 2 ** 7 k * 2 ** 3 k = 1 M

从 512 M 降到 1M 这个效率，所以设计好的数据存储结构是写好程序的第一步

作者回复: 你很好地理解了roaring bitmap的思路。不过空间不会无缘无故变出来的，在极端情况下（高16位都有，低16位都是位图），那么roaring bitmap不会比原始位图小。你可以仔细算一下，每个位图是8k，如果高16位每个数都存在，那么就有2^16个位图，2^16*8k = 2^9 M = 512M

共 3 条评论 >

5

微思

2020-04-09

老师好，对于文章中说哈希表没有遍历能力我很疑惑，我的理解：哈希表底层基于数组实现，也是可以遍历的，比如以Java语言的HashMap为例，就可以遍历读取hash表中的元素的。还请老师解惑，谢谢

作者回复: 此“遍历”非彼“遍历”。我所说的遍历，严格意义上是指“元素接元素的，时间代价是o(n)级别的遍历”。
你可以想一想这样一个例子:
一个哈希表数组长度为10000，然后我们在里面存入a和b两个元素。
那我们要遍历哈希表时，需要去穷举整个长度为10000的数组的每个位置，才能找出里面有两个元素a和b。这个其实不是“o(n)级别的元素接元素的遍历”。
因此，尽管许多容器都有类似的函数接口，比如containsKey，next等，但是它们的效率差距非常大。这也是为什么我们要熟悉它们的底层实现的原因，这样才能保证你的程序是高效的。

共 11 条评论 >

4

牛牛

2020-05-19

原来roaring bitmap是这么设计出来的, 上次觉得太高大上就跳过去了~~~, 今天细细的看了下, 稍微写了点笔记:

1. 设计思想: 将32位整数分成高16位和低16位, 高16位、存储到一个有序数组中, 每一个值都是一个`桶`; 低16位, 存储在2^16的位图中, 相应位置1, 查找时, 先判断桶是否存在, 若存在、再check位图相应位
如何节省空间的

若所有元素高16位都是相同的, 在有序数组部分, 只需要一个2字节的桶(高16位-2字节), 若提前分配好了2^16个桶, 那就需要128k的空间. 低16位, 因为长度是固定的, 都是2^16个bit、即8字节. (若使用普通位图, id范围是int32的话、则需要512M的空间), 因此可以很大的节省空间.

其实核心思想就是: 将不存在的桶的位图空间全部省去

2. 优化: Roaring bitmap还对低16位的位图进行了优化: 若桶中存储的数据少于4096(容量的1/16), 就使用short型的有序数组来存储, 上限是4096(4k)个, short占用两个字节, 这样存储空间就小于位图占用的空间了.

3. 有序数组和位图转换时机(与hashmap类似)

a. 刚插入时、数量少, 默认数组容器

b. 桶中数据增多时, 大于4096个, 则转为位图容器

c. 随数据删除, 元素个数小于4096个, 则退化为数组容器

4. 怎么快速求交集

比较高16位, 将所有相同的桶保留

对相同桶里的元素求交集, 3种情况: 数组&数组: 相互二分查找; 位图&位图: 直接位运算; 位图&数组: 遍历数组、在位图中查找.

作者回复: 总结得很好。你会发现，看上去高大上的东西，其实拆开来看，也是由更基础的知识组成。相信经过这样的学习，你以后看到了高大上的新知识，会更有信心去拆解和研究了。

3

Roger宇

2020-08-07

各位，问一个可能很低级的数学问题呀，id范围是int32类型的bitmap，怎么算出大小约为512m呢？int32取值范围是-2^31 到2^31-1。为什么一个bitmap大小约为512m呢?

作者回复: 计算方如下:

int32能表示2^32个数，如果用bitmap表示，每个数需要占1个bit，因此共需要2^32个bit。

2^32个bit = 2^29个 byte (8个bit为一个byte)

而2^10 = 1024 = 1k，2^20 = 1024 k = 1m

故2^29个byte = 2^9 * 2^20 个byte = 512m byte。

共 2 条评论 >

2

峰

2020-04-06

今天竟然是两篇加餐，表示老师很努力，很nice！

讨论题： 主要考虑的点就是1. 怎么计算交之后该桶的元素个数 2. 新建continer，还是说原地计算。（先只考虑新建吧）
位图和位图 可能最后的结果是数组也可以是位图，可以根据两个位图本身的数量（n1 n2），并假设其均匀分布，n1 * n2/65536 大于等于 4096 则用位图，否则用数组，得到结果发现不是对应的continer，就要转换了。
数组和位图 ，数组和数组 这两个就相对简单了， 结果必然是数组。
问题：hash 表不能遍历这个问题，和链表结合不就可以了吗？为什么还要存一份原始的posting list。

作者回复: 讨论题:分析得很好！先做预判再操作，如果预判错误了，那么就需要多做一次转换。先做预判是计算机系统中经常会涉及的一种设计和实现思想。在第二篇加餐中，是否要做集合分配律拆分也是基于预判的。包括数据库查询时，对SQL语句的优化也是会基于预判。
问题:哈希表+链表的结构的确可以（类似LinkedHashMap这样的结构）。不过链表的遍历效率不高，在链表较长的情况下，链表和哈希表求交不如用跳表法。因此，保留原始posting list最大的好处，是原始posting list可以是用跳表实现，也可以是用链表实现。有更多的适用性。（当然，如果原posting list是链表实现，其实就是你说的哈希表+链表了）


2

null

2020-07-05

老师，问下，位图如何获取里面存的所有id，是需要遍历每一位吗？

作者回复: 位图你其实可以看做是特殊的哈希表，因此它和哈希表一样，如果想获得里面存的所有id，需要低效地遍历每一位。


1

青鸟飞鱼

2020-06-09

老师你好，这是第n次刷压缩位图了，总算看明白了。有个问题想请教一下，高16用数组存，低16用位图，为什么不是高16位用位图，低16位用数组实现压缩位图呢。我想到的原因是因为高16位相同概率高吗？还是有其他原因呢？

作者回复: 多问为什么是一种很好的学习方法。我们一起来分析一下，为什么不是高16位用位图，低16位用数组呢?

你可以先考虑这样一种情况:如果高16位用位图，低16位也是位图，那么这和原先一个完整的位图表示是否有区别?你会发现，这样的结构，空间一点都没有节省。高16位有2^16个桶，每个桶有一个位图(大小是2^16)，那么总空间还是2^16 * 2^16 = 2^32。如果再加上索引本身的开销，那么会比原始位图还更占空间。

那我们要怎么做才会省空间呢？你会发现，如果位图中的元素很稀疏，那么将位图转为数组，会更省空间。因此，在初始的时候，我们应该把高16位和低16位都变成数组，这就是为什么高16位是数组的原因。

那么，按照这个思路，如果高16位的元素越来越多，那么其实当它达到一定数量的时候(和低16位的处理方案相似，大于8k时数组转为位图)，我们是可以将它转为位图的，这样效率的确会更高。

当然，一般来说，高16位是比较稀疏的，因此默认用数组就好。

通过这样的分析，相信你也会更好地理解，roaring bitmap也并不是万能的银弹，在数据稠密的情况下，其实直接使用位图是更好的选择。

1

密码123456

2020-04-08

为什么2^16是8k？2^32是512M?

作者回复: 这里省略了单位byte（字节）。8个bit是一个byte。

2^16是8k byte。而2^32是512M byte。

你可以算一下。

共 2 条评论>

1

范闲

2020-04-07

看到了同学们的评论，感觉大家思考的还是很全面啊。

看到了很多同学提到的预判的m值。这里的m值是1假设均匀分布下求出来的，也就是说是65536**2/(n1*n2）但是还看到是n1*n2/65536这两种不等价啊。

作者回复: 65536^2/（n1*n2）= t，表示的含义是每隔t个0会有一个1。因此一个长度为65536的位图中，1的个数就是65536/t = （n1*n2）/65536。

我也可以换另一个方式推导，你可以看看:

对于位图a，第一个位置是1的概率是n1/65536（推导过程:如果只有一个1，落在第一位的概率是1/65536；如果有n1个1，那么第一位为1的概率就是n1/65536）。
对于位图b，第一个位置为1的概率是n2/65536。
因此，位图a和位图b第一个位置同时为1的概率就是（n1*n2）/65536^2。
同理，第二个位置同时为1的概率也是（n1*n2）/65536^2。以此类推，一共有65536个位置，因此1的个数就是65536 * （n1*n2）/65536^2，结果就是（n1*n2）/65536。

不过话说回来，具体怎么求概率不是我出这道题的期望，只要大家能理解"程序在面临分支选择时，会快速通过计算进行预判"这个设计思想就好了。

共 2 条评论 >　　1

桃梦依然
2020-04-06

不同数据规模，使用不同算法。熟悉业务场景和数据规模，是根本。

作者回复: 是的。所以我这一篇把几种方法放在一起，方便对比。此外，在roaring bitmap中，数组容器和位图容器会自动相互转换，这就是一个很好的例子，包括在hashmap中也是，它会在链表和红黑树之间进行转换。因此，我们能学到的一个高性能系统的设计思路，就是自动根据数据规模转换最合适的数据结构和算法。

1

范闲
2020-04-06

讨论题:
对于三种求交集的结果个人认为用数组存储比较好。

原因:
最终的交集集合相对来说比较短，这个时候直接用数组比较好，可以直接通过遍历数组拿到返回结果。如果存储的是位图还需要这做转化，得不偿失。

作者回复: 你的方法在大部分情况下是可行的。在你的方法中，隐含了一个假设:最终的交集集合比较短。如果假设成立，那么自然是使用数组最合适。
那么，你也可以想想，有没有可能最终的交集会比较长?如果我们一开始就发现结果会很长（可以评估概率），是不是一开始直接准备一个位图就好了?（举个例子，一个位图是满的，另一个位图半满，那么它们的交集个数是不是肯定超过4096了?）

 
1

Gum

2022-08-30 来自浙江

请问老师 对于多个 posting list 求交集，如果以可变长数组的形式存储，还能再利用相互二分查找进行归并吗?

kang

2021-06-29

如果用位图的话.docId 超过了int 32怎么办?
搞一个链表存储位图吗?
1,2,3,4,5(End)
1,2,3,4,5(n)
第二列视为第一个集合的最后一个元素(End)+n

> 作者回复: 直接使用int64版本的roaringbitmap就好了。
> 不过你这也是一个思路，就是自己实现了一个big-int的思路

Geek_d62be1

2021-04-16

求交集以后的数据，是需要还原成id然后获取文档返回的吧，那如果低16位交集结果用位图存储，怎么去获取位图中的值，遍历吗？还是有什么快速的方法可以获取?

> 作者回复: 要知道位图中有多少个位是1，常规的办法是遍历，如果只有低16位，遍历的代价是可接受的。但如果位图很大(比如有1024位)，而被置为1的位很少的话，我们可以用位运算的思想，按32位或者64位为单位去与位图当前的范围做and运算，如果结果是0就快速跳过往后继续查找，这样就可以加速了。

 
1

星期八

2020-07-31

位图局限性第三点，512M如何计算得来的，位图本来就是表示元素是否存在，用为什么要用一个位图存储一个对象?

作者回复: 位图不需要存储具体对象，只需要表示元素是否存在。局限性第三点的正确断句是“完整的/对象空间”，意思是位图下标需要能覆盖对象ID值的完整范围。
而如果每个对象的ID是int32类型的话，那么ID的范围就是0至2^32 - 1。我们要保证位图能覆盖这个范围空间，因此就需要构造一个2^32个bit的位图，那么这个位图的大小就是 2^32 bit = 512M byte。


Ray
2020-07-08

老师，我对于位图和数组求交集的情况下预判错误后的再次转换有疑问，预判错误有两种情况，一是实际交集元素数量大于4096，这种情况下结果存储方案由数组切换到位图可以理解：节省空间，也可能不具有连续存储空间；另一种是实际交集元素个数小于4096个，这种情况下为什么要将位图转换为数组，毕竟可能并不会节省空间。
如果上面的分析正确的话，我想问，这样的设计思路难道是为了保证设计思想的统一吗?

作者回复: 首先，如果数组元素小于4096的话，数组空间肯定是小于位图的。
然后考虑一下极端情况，如果每次都预判错误，交集元素其实很少，但需要生成更费位图来存储而不是数组。那么roaring bitmap的空间浪费就会很严重，这就无法提供高效的压缩服务了。所以在预判错误的情况下，将位图转为数组是保证roaring bitmap有高压缩率的一个设计。


Geek_4ccf01
2020-06-02

老师，请问所讲的数据结构，是否有具体的实现代码可以看和参考

作者回复: 其实许多开源软件都有相关的实现了。比如说roaring bitmap就是开源的，可以在GitHub上找到。还有Redis，elastic search，levelDB等。里面都有对应的代码实现。有兴趣的话，你可以基于这几个开源软件去看代码的具体实现。


扁舟
2020-06-01

讲得是真的好，透彻，有一次体会到了数据结构优越带来的提升，这么好的专栏，极客应该多推广推广呀

作者回复: 谢谢你的肯定。其实许多复杂的系统和算法，都是由简单的数据结构和算法组合得来的。因此数据结构和算法作为基本功，值得每一个工程师好好学习和消化。