时间序列数据库的秘密 (2)——索引

* 陶文

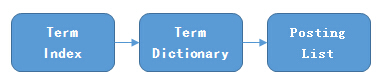
阅读数：239752015 年 8 月 20 日

话题：数据库语言 & 开发架构

如何快速检索？

Elasticsearch 是通过 Lucene 的倒排索引技术实现比关系型数据库更快的过滤。特别是它对多条件的过滤支持非常好，比如年龄在 18 和 30 之间，性别为女性这样的组合查询。倒排索引很多地方都有介绍，但是其比关系型数据库的 b-tree 索引快在哪里？到底为什么快呢？

笼统的来说，b-tree 索引是为写入优化的索引结构。当我们不需要支持快速的更新的时候，可以用预先排序等方式换取更小的存储空间，更快的检索速度等好处，其代价就是更新慢。要进一步深入的化，还是要看一下 Lucene 的倒排索引是怎么构成的。



这里有好几个概念。我们来看一个实际的例子，假设有如下的数据：

|  |  |  |
| --- | --- | --- |
| docid | 年龄 | 性别 |
| 1 | 18 | 女 |
| 2 | 20 | 女 |
| 3 | 18 | 男 |

这里每一行是一个 document。每个 document 都有一个 docid。那么给这些 document 建立的倒排索引就是：

年龄

性别

可以看到，倒排索引是 per field 的，一个字段由一个自己的倒排索引。18,20 这些叫做 term，而 [1,3] 就是 posting list。Posting list 就是一个 int 的数组，存储了所有符合某个 term 的文档 id。那么什么是 term dictionary 和 term index？

假设我们有很多个 term，比如：

**Carla,Sara,Elin,Ada,Patty,Kate,Selena**

如果按照这样的顺序排列，找出某个特定的 term 一定很慢，因为 term 没有排序，需要全部过滤一遍才能找出特定的 term。排序之后就变成了：

**Ada,Carla,Elin,Kate,Patty,Sara,Selena**

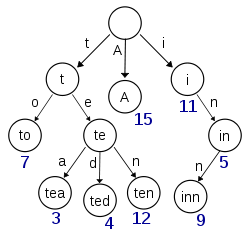
这样我们可以用二分查找的方式，比全遍历更快地找出目标的 term。这个就是 term dictionary。有了 term dictionary 之后，可以用 logN 次磁盘查找得到目标。但是磁盘的随机读操作仍然是非常昂贵的（一次 random access 大概需要 10ms 的时间）。所以尽量少的读磁盘，有必要把一些数据缓存到内存里。但是整个 term dictionary 本身又太大了，无法完整地放到内存里。于是就有了 term index。term index 有点像一本字典的大的章节表。比如：

A 开头的 term ……………. Xxx 页

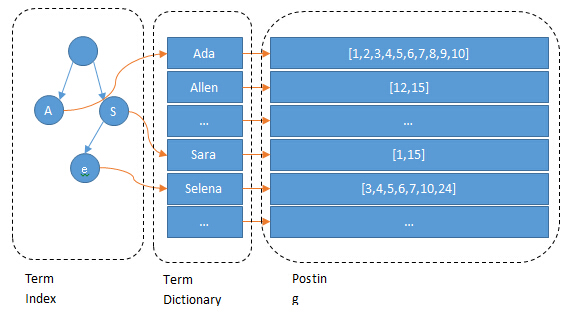
C 开头的 term ……………. Xxx 页

E 开头的 term ……………. Xxx 页

如果所有的 term 都是英文字符的话，可能这个 term index 就真的是 26 个英文字符表构成的了。但是实际的情况是，term 未必都是英文字符，term 可以是任意的 byte 数组。而且 26 个英文字符也未必是每一个字符都有均等的 term，比如 x 字符开头的 term 可能一个都没有，而 s 开头的 term 又特别多。实际的 term index 是一棵 trie 树：



例子是一个包含 "A", "to", "tea", "ted", "ten", "i", "in", 和 "inn" 的 trie 树。这棵树不会包含所有的 term，它包含的是 term 的一些前缀。通过 term index 可以快速地定位到 term dictionary 的某个 offset，然后从这个位置再往后顺序查找。再加上一些压缩技术（搜索 Lucene Finite State Transducers） term index 的尺寸可以只有所有 term 的尺寸的几十分之一，使得用内存缓存整个 term index 变成可能。整体上来说就是这样的效果。



现在我们可以回答“为什么 Elasticsearch/Lucene 检索可以比 mysql 快了。Mysql 只有 term dictionary 这一层，是以 b-tree 排序的方式存储在磁盘上的。检索一个 term 需要若干次的 random access 的磁盘操作。而 Lucene 在 term dictionary 的基础上添加了 term index 来加速检索，term index 以树的形式缓存在内存中。从 term index 查到对应的 term dictionary 的 block 位置之后，再去磁盘上找 term，大大减少了磁盘的 random access 次数。

额外值得一提的两点是：term index 在内存中是以 FST（finite state transducers）的形式保存的，其特点是非常节省内存。Term dictionary 在磁盘上是以分 block 的方式保存的，一个 block 内部利用公共前缀压缩，比如都是 Ab 开头的单词就可以把 Ab 省去。这样 term dictionary 可以比 b-tree 更节约磁盘空间。

如何联合索引查询？

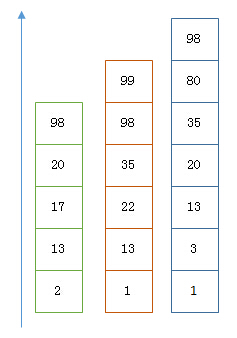
所以给定查询过滤条件 age=18 的过程就是先从 term index 找到 18 在 term dictionary 的大概位置，然后再从 term dictionary 里精确地找到 18 这个 term，然后得到一个 posting list 或者一个指向 posting list 位置的指针。然后再查询 gender= 女 的过程也是类似的。最后得出 age=18 AND gender= 女 就是把两个 posting list 做一个“与”的合并。

这个理论上的“与”合并的操作可不容易。对于 mysql 来说，如果你给 age 和 gender 两个字段都建立了索引，查询的时候只会选择其中最 selective 的来用，然后另外一个条件是在遍历行的过程中在内存中计算之后过滤掉。那么要如何才能联合使用两个索引呢？有两种办法：

* 使用 skip list 数据结构。同时遍历 gender 和 age 的 posting list，互相 skip；
* 使用 bitset 数据结构，对 gender 和 age 两个 filter 分别求出 bitset，对两个 bitset 做 AN 操作。

PostgreSQL 从 8.4 版本开始支持通过 bitmap 联合使用两个索引，就是利用了 bitset 数据结构来做到的。当然一些商业的关系型数据库也支持类似的联合索引的功能。Elasticsearch 支持以上两种的联合索引方式，如果查询的 filter 缓存到了内存中（以 bitset 的形式），那么合并就是两个 bitset 的 AND。如果查询的 filter 没有缓存，那么就用 skip list 的方式去遍历两个 on disk 的 posting list。

利用 Skip List 合并



以上是三个 posting list。我们现在需要把它们用 AND 的关系合并，得出 posting list 的交集。首先选择最短的 posting list，然后从小到大遍历。遍历的过程可以跳过一些元素，比如我们遍历到绿色的 13 的时候，就可以跳过蓝色的 3 了，因为 3 比 13 要小。

整个过程如下

Next -> 2

Advance(2) -> 13

Advance(13) -> 13

Already on 13

Advance(13) -> 13 MATCH!!!

Next -> 17

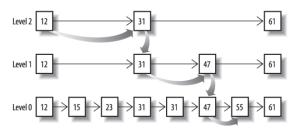
Advance(17) -> 22

Advance(22) -> 98

Advance(98) -> 98

Advance(98) -> 98 MATCH!!!

最后得出的交集是 [13,98]，所需的时间比完整遍历三个 posting list 要快得多。但是前提是每个 list 需要指出 Advance 这个操作，快速移动指向的位置。什么样的 list 可以这样 Advance 往前做蛙跳？skip list：



从概念上来说，对于一个很长的 posting list，比如：

[1,3,13,101,105,108,255,256,257]

我们可以把这个 list 分成三个 block：

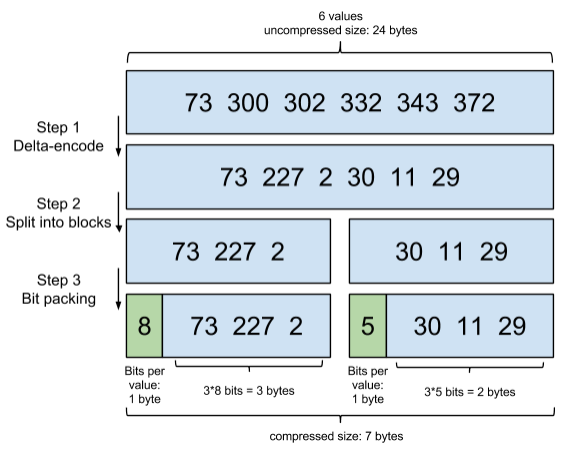
[1,3,13] [101,105,108] [255,256,257]

然后可以构建出 skip list 的第二层：

[1,101,255]

1,101,255 分别指向自己对应的 block。这样就可以很快地跨 block 的移动指向位置了。

Lucene 自然会对这个 block 再次进行压缩。其压缩方式叫做 Frame Of Reference 编码。示例如下：



考虑到频繁出现的 term（所谓 low cardinality 的值），比如 gender 里的男或者女。如果有 1 百万个文档，那么性别为男的 posting list 里就会有 50 万个 int 值。用 Frame of Reference 编码进行压缩可以极大减少磁盘占用。这个优化对于减少索引尺寸有非常重要的意义。当然 mysql b-tree 里也有一个类似的 posting list 的东西，是未经过这样压缩的。

因为这个 Frame of Reference 的编码是有解压缩成本的。利用 skip list，除了跳过了遍历的成本，也跳过了解压缩这些压缩过的 block 的过程，从而节省了 cpu。

利用 bitset 合并

Bitset 是一种很直观的数据结构，对应 posting list 如：

[1,3,4,7,10]

对应的 bitset 就是：

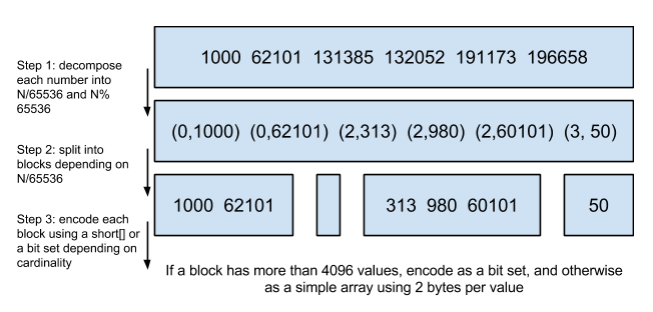
[1,0,1,1,0,0,1,0,0,1]

每个文档按照文档 id 排序对应其中的一个 bit。Bitset 自身就有压缩的特点，其用一个 byte 就可以代表 8 个文档。所以 100 万个文档只需要 12.5 万个 byte。但是考虑到文档可能有数十亿之多，在内存里保存 bitset 仍然是很奢侈的事情。而且对于个每一个 filter 都要消耗一个 bitset，比如 age=18 缓存起来的话是一个 bitset，18<=age<25 是另外一个 filter 缓存起来也要一个 bitset。

所以秘诀就在于需要有一个数据结构：

* 可以很压缩地保存上亿个 bit 代表对应的文档是否匹配 filter；
* 这个压缩的 bitset 仍然可以很快地进行 AND 和 OR 的逻辑操作。

Lucene 使用的这个数据结构叫做 Roaring Bitmap。



其压缩的思路其实很简单。与其保存 100 个 0，占用 100 个 bit。还不如保存 0 一次，然后声明这个 0 重复了 100 遍。

这两种合并使用索引的方式都有其用途。Elasticsearch 对其性能有详细的对比（[https://www.elastic.co/blog/frame-of-reference-and-roaring-bitmaps](https://www.elastic.co/blog/frame-of-reference-and-roaring-bitmaps" \t "_blank)）。简单的结论是：因为 Frame of Reference 编码是如此 高效，对于简单的相等条件的过滤缓存成纯内存的 bitset 还不如需要访问磁盘的 skip list 的方式要快。

如何减少文档数？

一种常见的压缩存储时间序列的方式是把多个数据点合并成一行。Opentsdb 支持海量数据的一个绝招就是定期把很多行数据合并成一行，这个过程叫 compaction。类似的 vivdcortext 使用 mysql 存储的时候，也把一分钟的很多数据点合并存储到 mysql 的一行里以减少行数。

这个过程可以示例如下：

|  |  |
| --- | --- |
| 12:05:00 | 10 |
| 12:05:01 | 15 |
| 12:05:02 | 14 |
| 12:05:03 | 16 |

合并之后就变成了：

可以看到，行变成了列了。每一列可以代表这一分钟内一秒的数据。

Elasticsearch 有一个功能可以实现类似的优化效果，那就是 Nested Document。我们可以把一段时间的很多个数据点打包存储到一个父文档里，变成其嵌套的子文档。示例如下：

{timestamp:12:05:01, idc:sz, value1:10,value2:11}

{timestamp:12:05:02, idc:sz, value1:9,value2:9}

{timestamp:12:05:02, idc:sz, value1:18,value:17}

可以打包成：

{

max\_timestamp:12:05:02, min\_timestamp: 1205:01, idc:sz,

records: [

{timestamp:12:05:01, value1:10,value2:11}

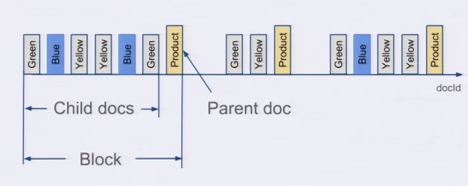
{timestamp:12:05:02, value1:9,value2:9}

{timestamp:12:05:02, value1:18,value:17}

]

}

这样可以把数据点公共的维度字段上移到父文档里，而不用在每个子文档里重复存储，从而减少索引的尺寸。



（图片来源：<https://www.youtube.com/watch?v=Su5SHc_uJw8>，Faceting with Lucene Block Join Query）

在存储的时候，无论父文档还是子文档，对于 Lucene 来说都是文档，都会有文档 Id。但是对于嵌套文档来说，可以保存起子文档和父文档的文档 id 是连续的，而且父文档总是最后一个。有这样一个排序性作为保障，那么有一个所有父文档的 posting list 就可以跟踪所有的父子关系。也可以很容易地在父子文档 id 之间做转换。把父子关系也理解为一个 filter，那么查询时检索的时候不过是又 AND 了另外一个 filter 而已。前面我们已经看到了 Elasticsearch 可以非常高效地处理多 filter 的情况，充分利用底层的索引。

使用了嵌套文档之后，对于 term 的 posting list 只需要保存父文档的 doc id 就可以了，可以比保存所有的数据点的 doc id 要少很多。如果我们可以在一个父文档里塞入 50 个嵌套文档，那么 posting list 可以变成之前的 1/50。

作者简介

**陶文**，曾就职于腾讯 IEG 的蓝鲸产品中心，负责过告警平台的架构设计与实现。2006 年从 ThoughtWorks 开始职业生涯，在大型遗留系统的重构，持续交付能力建设，高可用分布式系统构建方面积累了丰富的经验。

感谢[张凯峰](http://www.infoq.com/cn/author/%E5%BC%A0%E5%87%AF%E5%B3%B0)对本文的策划，[丁晓昀](http://www.infoq.com/cn/author/%E4%B8%81%E6%99%93%E6%98%80)对本文的审校。

给 InfoQ 中文站投稿或者参与内容翻译工作，请邮件至[editors@cn.infoq.com](mailto:editors@cn.infoq.com" \t "_blank)。也欢迎大家通过新浪微博（[@InfoQ](http://www.weibo.com/infoqchina" \t "_blank)，[@丁晓昀](http://weibo.com/u/1451714913" \t "_blank)），微信（微信号：[InfoQChina](http://weixin.sogou.com/gzh?openid=oIWsFt0HnZ93MfLi3pW2ggVJFRxY" \t "_blank)）关注我们，并与我们的编辑和其他读者朋友交流（欢迎加入 InfoQ 读者交流群