搜索索引更新策略

如果搜索引擎需要处理的文档集合是静态集合，那么在索引建立好之后，就可以一直用建立好的索引响应用户查询需求。但是，在真实环境中，搜索引擎需要处理的文档集合往往是动态的。即在建好初始的索引后，后续不断有新文档进入系统，同时原有的文档集合内会被删除或者更改。

索引系统如何能够做到实时反映这种变化呢?动态索引就可以实现这种实时要求。其中有3个关键的索引结构：倒排索引，临时索引，已删除文档列表。

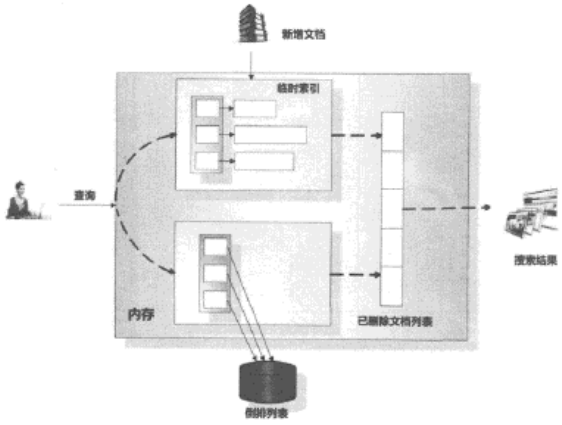


图 3-15 动态索引

倒排索引就是已经建立好的索引结构。一般单词词典存储在内存，对应的倒排列表存储在磁盘文件中。

临时索引是在内存中实时建立的倒排索引，其结构和前面的倒排索引时一样的。当有新文档进入系统，实时解析文档并将其追加到这个临时索引结构。

已删除文档列表，用来存储已被删除的文档的相应文档id，形成一个文档id列表。

当系统发现有新文档进入时，立即将其加入到临时索引中。有文档被删除时，则将其加入到删除文档队列。文档更改时，则将其原文档放入到删除队列，解析更改后的文档，加入到临时索引中，通过这种方式满足实时性的要求。

如果用户输入查询请求，则搜索引擎同时从**倒排索引**和**临时索引**中服务用户查询单词的倒排列表，找到包含用户查询的文档集合，并对两个结果进行**合并**，之后利用**删除文档列表进行过滤**。将搜索结果中那些已经被删除的文档从结果中过滤，形成最终的搜索的结果，并返回给用户，这样就能实现动态环境下的准实时搜索功能。

动态索引通过在内存中维护临时索引，可以实现对动态文件和实时搜索的支持。但是服务器内存总是有限的，随着新加入系统的文档越来越多，临时索引消耗的内存也会随之增加。当最初分配的内存将被使用完时，要考虑将临时索引的内容更新到磁盘索引中，以释放内存空间来容纳后续的新进文档。
下面介绍四种索引更新策略：完全重建策略、再合并策略、原地更新策略及混合策略。

1.1完全重建策略（Complete Re-Build）

当新文档增加到一定数量，将新文档和老文档进行合并，然后对所有文档重新建立索引。

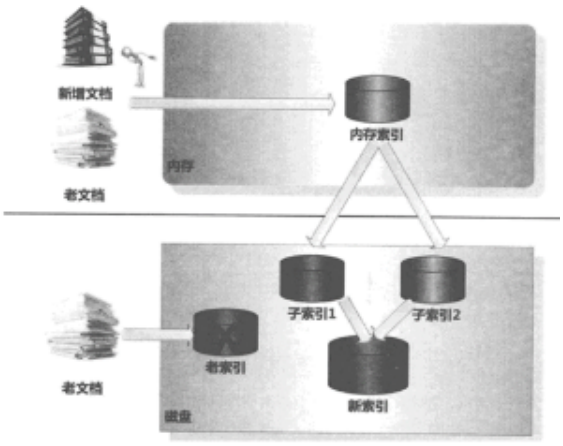
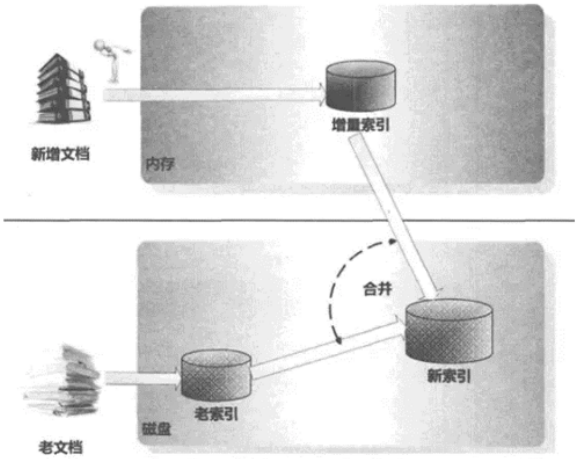


图 3-16 完全重建策略

因为重建索引需要较长时间，在进行索引重建的过程中，内存中仍然需要维护老索引来对用户的查询做出响应。只有当新索引完全建立完成后，才能释放旧的索引，将用户查询响应切换到新索引上。

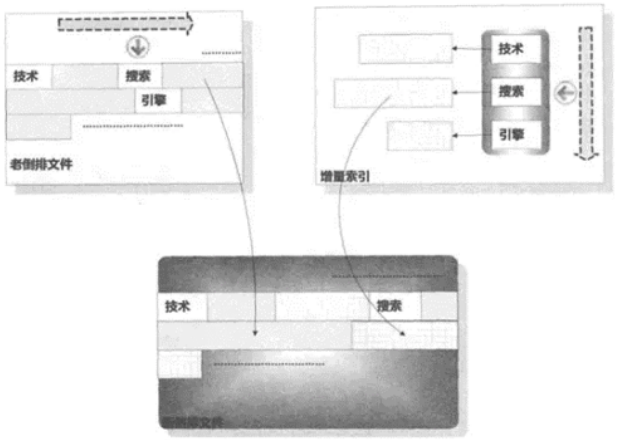
这种策略适合比较小的文档集合，但是目前主流商业搜索引擎也一般是采用这种方式维护索引更新，这跟互联网本身的特性有关。

1.2再合并策略（Re-Merge）



在合并过程中，需要依次遍历增量索引和老索引单词词典中包含的单词以及其对应的倒排列表，可以用两个指针分别指向增量索引和老索引目前需要合并的单词。

1.如果增量索引中指针指向的单词ID**小于**老索引中指针指向的单词ID，则说明这个单词在老索引中没有出现过，直接将这个单词对应的倒排列表写入新索引的倒排列表中，同时增量索引单词指针指向下一个单词。
2.如果两个单词ID**相等**，则先将老索引中这个单词对应的倒排列表加入新索引，然后在把增量索引这个单词对应的倒排列表追加到其后。如下图所示：



合并增量索引与老索引

3.如果新索引指向的单词ID**大于**老索引指针指向的单词ID，则直接将老索引中对应的倒排列表加入新索引的倒排文件中，老索引的单词指针指向下一个单词。

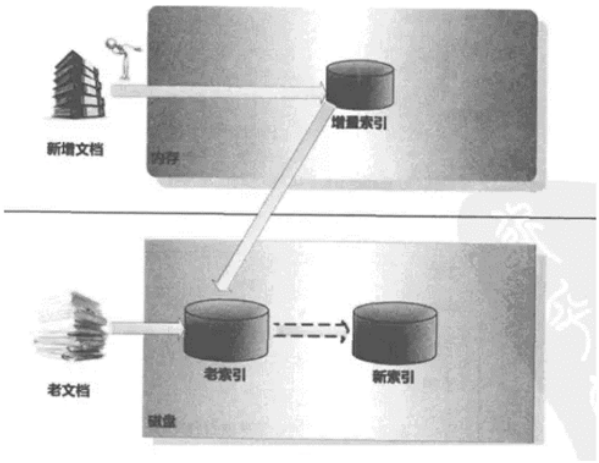
4.两个索引的所有单词都遍历完后，新索引建成。

优点：再合并策略是效率非常高的一种索引策略，主要是因为在对老索引进行遍历时，因为已经按照索引单词的词典顺序由低到高排好顺序，所以可以顺序读取文件内容，减少磁盘寻道时间。

缺点：对于老索引中的很多单词来说，尽管其倒排列表并为发生任何变化，但是也需要将其从老索引中读取出来并写入新索引中，这样就会造成很大的磁盘输入输出消耗。

1.3原地更新策略（In-Place）

在索引更新过程中，如果老索引的倒排列表没有变化，可以不需要读取这些信息，而只是对那些倒排列表变化的单词进行处理，或者是直接将发生变化的倒排列表追加到老索引的末尾，即只更新增量索引里出现的单词相关信息，这样就可以减少大量的磁盘读写操作，提升系统执行效率。



存在问题：对于倒排文件中的两个相邻单词，为了在查询时加快读写速度，其倒排列表一般是顺序存储的，这导致没有空余位置来追加新信息。为了能够支持追加操作，原地更新策略在初始建立的索引中，会在每一个单词的倒排列表末尾预留出一定的磁盘空间。当预留空间不足时，需要在磁盘中找到一块完整的连续存储区，将增量索引对应的倒排列表追加到其后，实现倒排列表的**“迁移”**工作。

原地更新策略出发点很好，但是实验数据证明其**索引更新效率比再合并策略低**，主要有两个原因：

1.在这种方法中，对倒排列表的“迁移”是比较常见的。这个策略需要对磁盘可用空间进行维护和管理，这种维护和查找成本非常高，这成为该方法效率的一个瓶颈。

2.由于存在数据迁移，某些单词及其对应的倒排列表会从老索引中溢出，这样就破坏了单词的连续性，导致在进行索引合并的时候不能进行顺序读取，必需维护一个单词到其倒排文件相应位置的映射表，这样不仅降低了磁盘读写速度，而且需要大量的内存来存储这种映射信息。

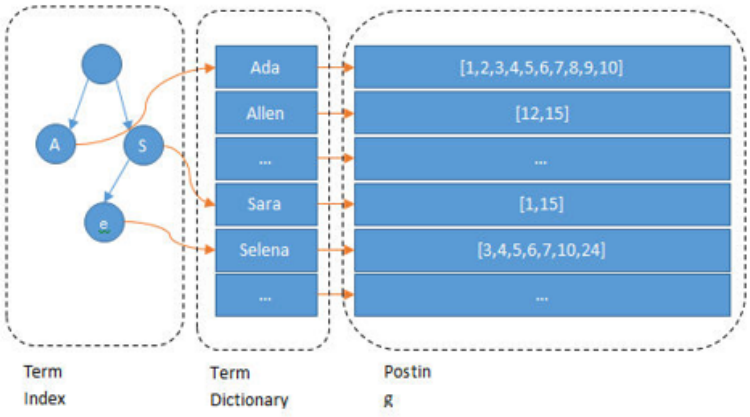
1.4混合策略（Hybrid）

混合策略一般会将单词根据其不同性质进行分类，不同类别的单词，对其索引采取不同的索引更新策略。常见做法：根据单词的倒排索引列表长度进行区分，因为有些单词经常在不同的文档中出现，所以其对应的倒排列表很长，而有些单词很少见，对应的倒排列表就较短。根据这一性质将单词划分为**长倒排列表单词**和**短倒排列表单词**。*长倒排列表单词采取原地更新策略，而短倒排列表单词采用再合并策略。*

因为原地更新策略能够节省磁盘读写次数，而长倒排列表单词的读写开销明显要比短倒排列表单词大很多，所以如果采用原地更新策略，效果体现比较显著。而大量的短倒排列表单词读写开销相对而言不算太大，利用再合并策略来处理，其顺序读写优势也能被充分利用。

写在后面：

上述所说的4中策略都是基于TermIndex，Term Dictionary，Posting List三种结构的基础之上。其中CURD操作涉及到Trie，Hashtable，skip table的相关变化。



ps 常见的搜索提示服务算法：

1.基于Redis的自动补全算法
https://allenwind.github.io/2017/11/26/%E5%9F%BA%E4%BA%8ERedis%E7%9A%84%E8%87%AA%E5%8A%A8%E8%A1%A5%E5%85%A8%E5%8A%9F%E8%83%BD/

2.双数组
https://blog.csdn.net/u013300579/article/details/78869742

3.三叉树
https://blog.csdn.net/ntc10095/article/details/52759489

4.Trie树+TOP K算法
https://www.kancloud.cn/wizardforcel/the-art-of-programming-by-july/97277

字典的几种结构：

数据结构	优缺点
排序列表Array/List	使用二分法查找，不平衡
HashMap/TreeMap	性能高，内存消耗大，几乎是原始数据的三倍
Skip List	跳跃表，可快速查找词语，在lucene、redis、Hbase等均有实现。相对于TreeMap等结构，特别适合高并发场景（Skip List介绍）
Trie	适合英文词典，如果系统中存在大量字符串且这些字符串基本没有公共前缀，则相应的trie树将非常消耗内存（数据结构之trie树）
Double Array Trie	适合做中文词典，内存占用小，很多分词工具均采用此种算法（深入双数组Trie）
Ternary Search Tree	三叉树，每一个node有3个节点，兼具省空间和查询快的优点（Ternary Search Tree）
Finite State Transducers (FST)	一种有限状态转移机，Lucene 4有开源实现，并大量使用