可以作为一个大型分布式集群（数百台服务器）技术,处理PB级数据,服务大公司;也可以运行在单机上,服务小公司。

Apache Lucene是一个用Java编写的高性能，功能齐全信息检索库。Elasticsearch内部利用Lucene来构建的分布式和分析功能。提供了 REST API 的操作接口，开箱即用。

Elasticsearch在后台使用Lucene来提供最强大的全文检索，提供任何开源产品的能力。搜索自带的多语言支持，强大的查询语言，地理位置支持，上下文感知的建议，自动完成和搜索片段。

Elasticsearch允许你快速上手。简单的指定一个JSON文档将自动检测数据的结构和类型，创建一个索引，并使你的数据检索。还拥有完全控制，以自定义数据是如何被索引。

一.基本概念

1.1 Node

Elastic是一个分布式数据库，可以运行在多个节点上，多个节点可以组成一个集群。

1.2 Index

Elastic 会索引所有字段，经过处理后写入反向索引（Inverted Index），当查找时会直接查找这个索引。

索引，Elastic数据管理的顶层就是Index(索引)，它是单个数据库的同义词。每个Index的名称必须是小写。

1.3 Document

Index 里面单条的记录成为Document(文档)，许多个Document 构成了一个Index

Document 使用JSON格式表示。同一个Index中的Document不要求使用相同的结构（Schema）,但是最好使用相同的，这样有利于提高查询效率。

1.4 Type

用于将Document分组，但是不同的Type 需要有相同的数据结构，如果有不同的数据结构，应该存在不同的Index中。由于Type可以分组但是却要求有相同的数据结构，这样不利于查询的效率。

在6.X版中一个Index只可以有一个Type,在7.X版本中可以不指定Type.

1.5 Shard

每个Index上包含多个Shard,默认是5个，分散在不同节点上，不会存在两个相同的Shard存在同一个Node上。Shard是最小的Lucene索引单元。

当Document 存储时Elastic会通过doc id进行hash来确定存储到哪一个Shard上，然后再Shard上面进行索引存储。

1.6 Segments

一个Index由许多独立的Segments 组成，而 Segments 则包含了文档中的Term Dictionary ,Term Index的倒排索引以及Document的字段数据。

Segments 直接提供了搜索功能，ES的一个Shard(Lucence Index)中是由大量的Segments构成的，且每一次fresh 都会产生新的Segments文件。但是这样产生的Segments会有大有小，相当碎片化。因此在ES内部会开启一个线程，将小的Segments 合并(Merge)为一个大的Segments,减少碎片化，降低文件打开数，提升IO性能。

二.Elastic索引深入探究

Elasticsearch索引的精髓是

> 一切设计都是为了提高搜索的性能

在插入数据的同时，会为每一个字段建立索引——倒排索引。

有如下要存的数据

> 1. The quick brown fox jumped over the lazy dog
> 2. Quick brown foxes leap over lazy dogs in summer

为了建立倒排索引首先将每个文档的Content分成一个个单独的词（可以称之为词条或者是Tokens）创建一个不重复的词条的排序列表。

得到如下的结构

Term	Doc_1	Doc_2
brown	X	X
dog	X	X
fox	X	X
in		X
jump	X	X
lazy	X	X
over	X	X
quick	X	X
summer		X
the	X	X

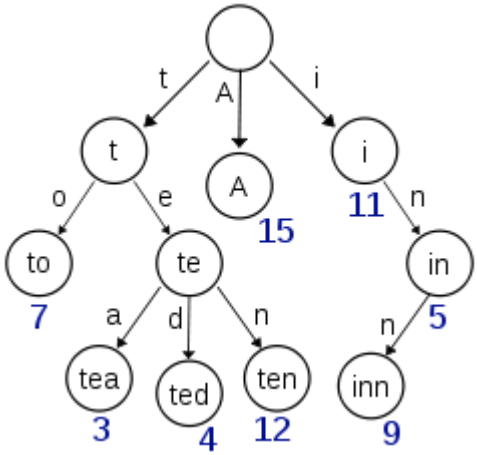
假设doc1的id为1，doc2的id为2，这个ID是Elasticsearch自建的文档ID,那么经过倒排索引之后我们得到如下的对应关系

> **Term Posting List**
>
> The 【1】
>
> quick 【1,2】
>
> brown 【1,2】
>
> fox 【1】
>
> …… ……

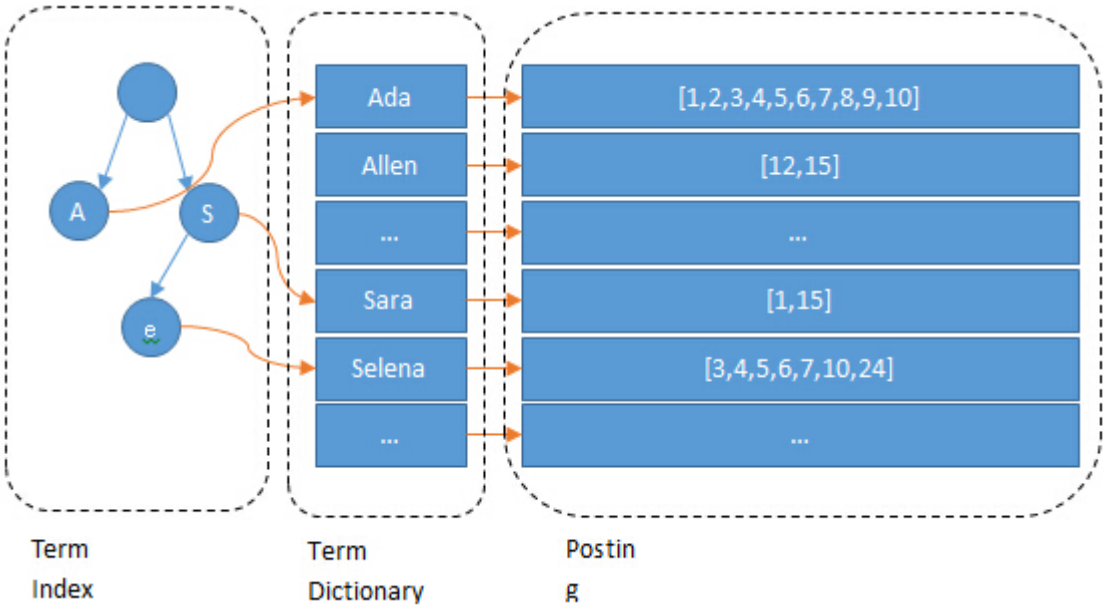
Elasticsearch为了能快速查找到某一个term，将所有的term排个序，然后使用二分法查找term,logN的查找效率，就像通过字典查找一样，这就是**Term Dictionary**

Term Index

如果数据量大的话使用**Term Dictionary**依然会开销过大，放内存中不现实，因此有了Term Index，就像字典里面的索引一样。比如存储A开头的有哪些term 存在哪一页等。可以将Term Index看作是一棵树。



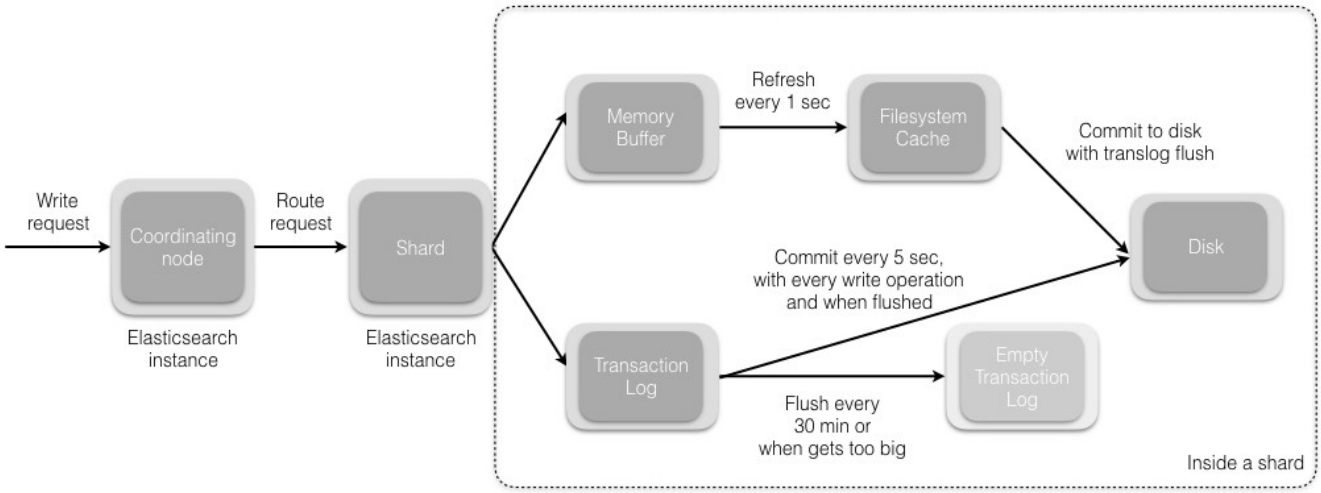
这棵树不会包含所有的term,包含的是term的一些前缀。通过Term Index可以快速的定位到Term Dictionary的某个offset，然后从这个位置继续进行查找。



因此Term Index不需要存储所有的Term,存储的是Term 的前缀与Term Dictionary的block之间的关系，再结合FST的压缩技术可以将Term Index 缓存到内存中。

从term index查到对应的term dictionary的block位置之后，再去磁盘上找term，大大减少了磁盘随机读的次数。

三.Elastic 存储过程



3.1索引创建

1.新document首先写入内存buffer缓存中，

2.每隔一段时间，执行commit point操作 buffer写入新Segment中

3.新segment写入文件系统缓存filesystem cache中

4.文件系统缓存中的index Segment 被fsync强制刷到磁盘上，确保物理写入。此时新Segment被打开供search操作。

5.清空内存buffer,可以接收新的文档。

6.以上是传统的写入步骤，实际上Elasticsearh为保证实时性，会进行refresh操作。

7.在新的文档写入后，写入index buffer的同时会写入translog

8.reflush操作使得写入文档搜索可见

9.flush操作使得filesystem cache 写入磁盘，以达到持久化的目的。

3.2 索引更新/删除

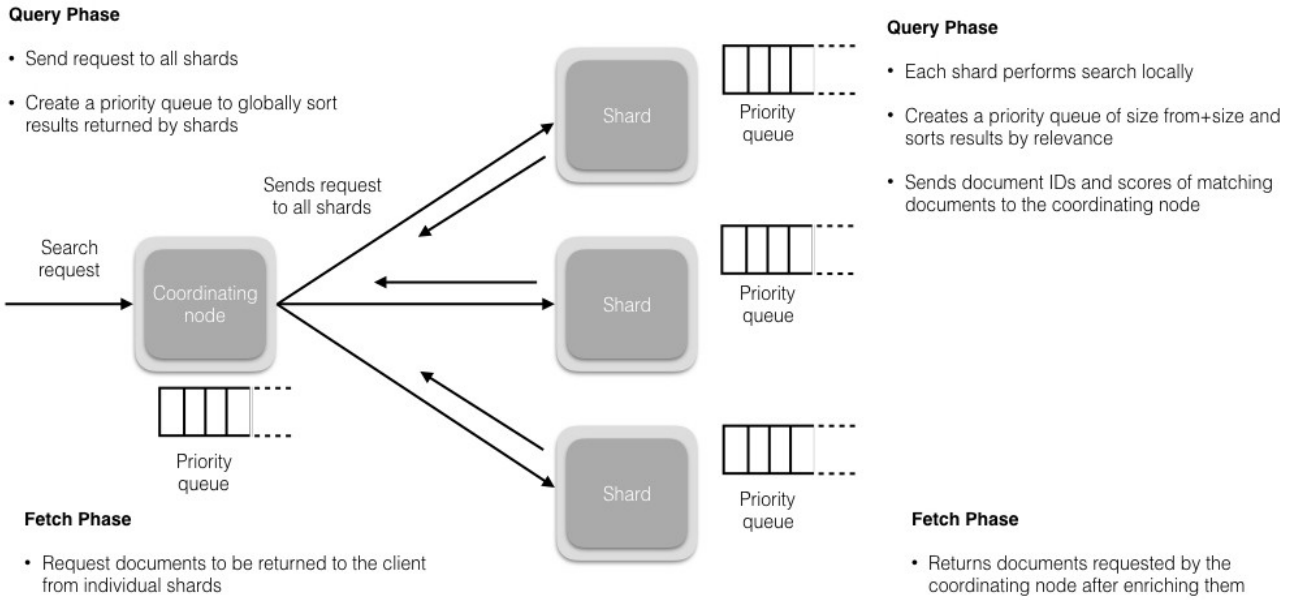
删除和更新也是写操作，但是在Elasticsearch中Document是不可变的。因此不能直接进行更新和删除。

1.在每个segment中存在一个.del文件，这个文件记录的segment的删除状态。当发送删除请求时，并未被实际删除，而是在.del中记录状态为删除。此文档仍然可以被搜索到，但是在结果中会被过滤掉。

2.创建新文档的时候会为文档分配一个版本号。文档每次更新都能产生一个新的版本号。当执行更新操作时，旧版本会在.del文件中标记删除，并且新版本在新的Segments中编入索引。旧版本仍然能被查询匹配，但是会在结果中被过滤掉。

当执行归并（Merge）的时候，老的segment 文件将会被删除，合并成新的segment 文件，这个时候也就是物理删除了。

四.Elastic 读取过程



4.1查询（Query Phase）阶段

1.在这个阶段首先协调节点将搜索请求路由到Index下的所有Shard(包括主要和副本)。

2.各个Shard独立执行搜索，根据相关性分数创建一个优先级排序结果。

3.各个Shard将匹配的Document和Document相关性分数ID返回给协调节点。

4.协调节点创建一个新的优先级队列，对全局结果进行排序。

5.查询可以由许多个结果，但是默认每个Shard会获取前10个结果，发送给协调节点，协调节点则返回全局的前10个结果。

4.2 获取（Fetch Phase）阶段

在协调节点对所有结果进行排序，生成全局排序的列表之后，他将会向各个分片请求原始文档。各个分片则会将文档补充完整后返回给协调节点。

五.Elastic 搜索相关性

The relevance is determined by a score that Elasticsearch gives to each document returned in the search result. The default algorithm used for scoring is tf/idf (term frequency/inverse document frequency). The term frequency measures how many times a term appears in a document (higher frequency == higher relevance) and inverse document frequency measures how often the term appears in the entire index as a percentage of the total number of documents in the index (higher frequency == less relevance). The final score is a combination of the tf-idf score with other factors like term proximity (for phrase queries), term similarity (for fuzzy queries), etc.