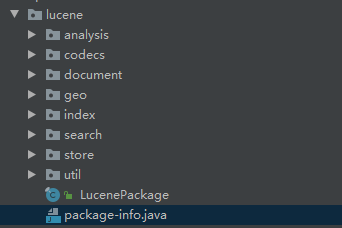
# 基于Lucene-8.6.2

## 概览

Lucene主要有三部分, 分别是: 索引, 搜索, 合并segment.

## core目录简介:



### analysis:

analysis的主要功能是分析field. 形成可索引可搜索的分词.

例如:



分析分词的开始位置和偏移量



分析位置增量,一般用于短语查询(Phrase Query)



分析每个分词的跨度,一般用于同义词过滤场景.

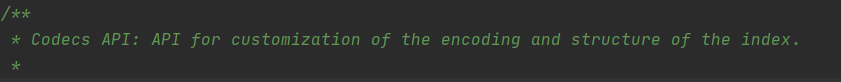


分析标准化因子,norm factor,一个影响评分的因素



分析分词类型

### codecs:



负责一些用到的数据结构和编解码算法的实现

### document:

The document package provides the user level logical representation of content to be indexed and searched. The  
\* package also provides utilities for working with {**@link** org.apache.lucene.document.Document}s and {**@link** org.apache.lucene.index.IndexableField}s.

定义document的field类型以及特性,store,term vector等

### geo:

地理空间工具,关于地理空间(经纬度)数据的计算,查询.

### index:

主要负责索引的增删改,flush和commit, merge工作, IndexWriter和IndexReader类所在的位置.

### search

负责对索引的搜索,评分(similarities),定义了常见的Query类

### store

主要负责对磁盘I/O的操作.Dictionary,IndexInput,IndexOutput类所在位置

### util

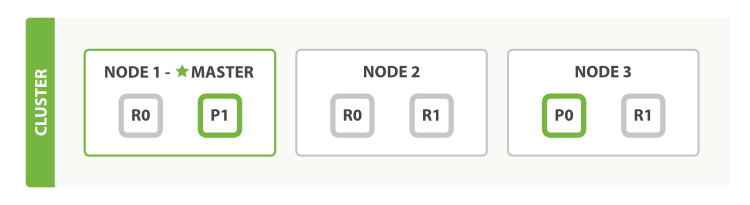
提供一些工具类,FST,LZ4,BKD,PackedInt,bitset和排序.

## index 索引

elasticsearch中的shard对应Lucene中的index. 每个shard都是独立的Lucene索引.

index索引有三种打开模式: create, append, create\_or\_append.

elasticsearch中:

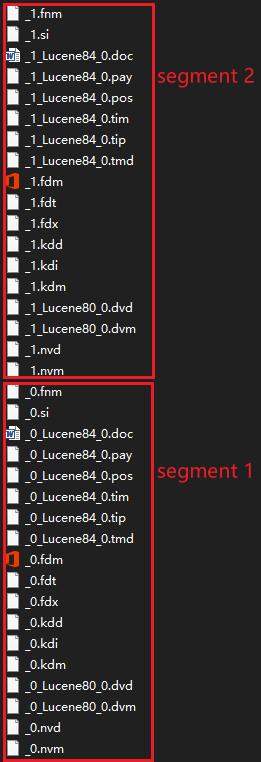


### 关于segment

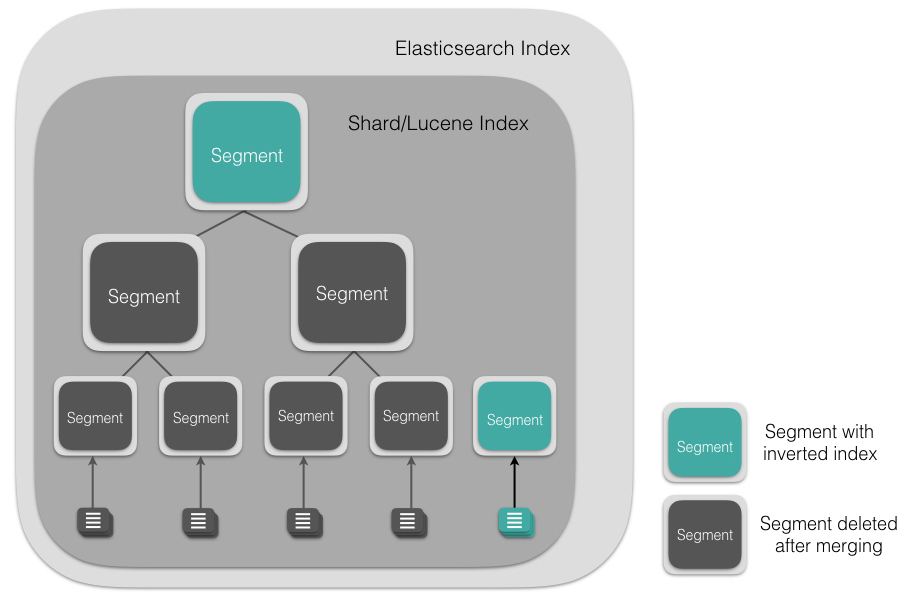
Lucene的文件结构有两种模式,一种是混合模式,一种是多文件模式

混合模式是将一个segment的不同类型数据都存放在.cfs文件中,Lucene默认.

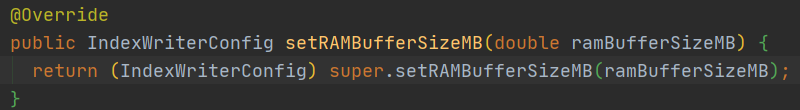
多文件模式将不同类型的数据放到不同的文件.

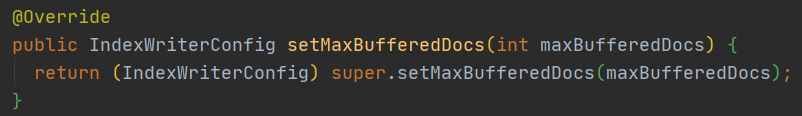


在Lucene中, 数值部分相同的文件共同组成一个segment. 搜索时会遍历每个segment, 每个segment都是独立的,共同组成index.

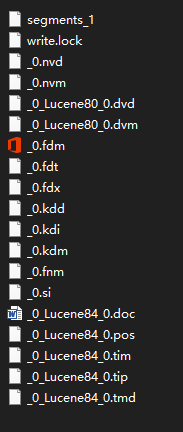


可以设置缓存最大值和缓存中最大文档数,内存中缓存的文档达到任意一个条件,都会触发生成segment.





### Lucene文件简介



### write.lock

一个索引同一时间只能有一个写操作,但写和读互不影响.

### segment\_N

记录index的所有segment信息,IndexWriter每commit一次, N的值会递增1.

仅仅通过flush的segment不能成为可搜索的segment.可以flush多次以后再commit一次.

一个索引中可能存在多个segment\_N文件,N值最大的代表最新的一次提交.

This {**@link** IndexDeletionPolicy} implementation that keeps only the most recent commit and immediately removes all prior commits after a new commit is done. This is the default deletion policy.

Lucene的默认策略是只保存最新的一次提交,之前的segment\_N在提交后立即删除.

### tim/tip

tim为term dictionary文件

tip 文件中存放的数据指向tim文件,看作tim的目录

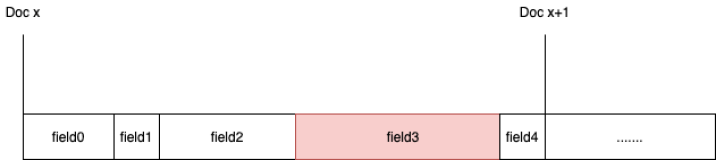
### fdx/fdt/fdm

fdm保存一些基本metadata,用于辅助读取fdx文件

fdx会先被读取到内存中,看做fdt的目录

fdt保存field原始信息(通过lz4压缩),目的是通过docId查出文档内容.

fdt:



具体步骤:

1. 在CompressingStoredFieldsIndexReader的构造函数中加载所有的"目录信息"
2. 确定docId所在的segment,由于starts数据记录了每个segment的docId的起始值,所以通过二分查找,很快可以找到对应的segment
3. 确定docId所在的block
4. 确定docId所在的chunk
5. 根据docId确定的chunk,找到chunk在fdt文件中的起始位置.
6. 读取fdt文件中的chunk信息,通过DocLengths和给定的docId确定整个chunk存储的所有doc的总长度totalLength和从baseDoc到docID的doc长度length。并用LZ4解压Chunk中的doc内容。当然，并不需要整个chunk的doc都解压，只需要解压到length的长度就可以了.
7. 解压还原文本内容,生成document.

### dvd/dvm

docvalues 相关文件,可以看做docId->field value

### kdd/kdi/kdm

points类型的索引文件,数据结构为bkd tree.

### nvd/nvm

norm factor相关文件

nvd 保存data

nvm保存metadata

### doc

保存term的倒排表信息,包括docId的倒排链,term在docId的term freq信息等

### pos

保存词汇在term中的位置信息

### liv

记录被删除文档信息(index级别,所有segment公用一个文件)

## 一些概念

### field

域,可以看作map中的key, 每个field都有name和type

#### StringField &TextField:

elasticsearch新版本中没有string类型,用keyword代替. String类型不可以被分词,Text类型可以分词, Text类型不支持doc value, String类型支持.

elasticsearch中的multi-field:

{

"mappings": {

"properties": {

"city": {

"type": "text",

"fields": {

"raw": {

"type": "keyword"

}

}

}

}

}

}

The city.raw field is a keyword version of the city field.

The city field can be used for full text search.

The city.raw field can be used for sorting and aggregations

#### Points类型:

点数据类型,可以表示多维数据

#### DocValues类型:

当我们需要对数据进行聚合或者排序时, 先根据FST找到关键词对应的docId列表, 然后提取所有document对应的field, 再进行聚合操作,如果数据量大容易造成内存溢出,并且性能很差.

示例:

{

"size": 0,

"aggs": {

"NAME": {

"terms": {

"field": "countryName.keyword",

"size": 300

}

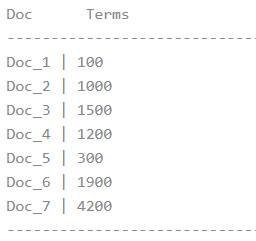
}

}

}

如果没有doc value, 会遍历所有document,然后提取countryName.keyword进行聚合.

doc values建立docId->field value,列式存储数据,可以看成用空间换时间,并且减少内存溢出.



Doc Values 不是由 JVM 来管理.

<https://www.elastic.co/guide/cn/elasticsearch/guide/current/heap-sizing.html>

elasticsearch默认对每个field(analyzed类型除外)开启doc value,不需要的可以禁用:

{

"mappings": {

"my\_type": {

"properties": {

"session\_id": {

"type": "string",

"index": "not\_analyzed",

"doc\_values": false

}

}

}

}

}

让一个字段可以被聚合，通过禁用倒排索引，使它不能被正常搜索:

{

"mappings": {

"my\_type": {

"properties": {

"customer\_token": {

"type": "string",

"index": "not\_analyzed",

"doc\_values": true,

"index": "no"

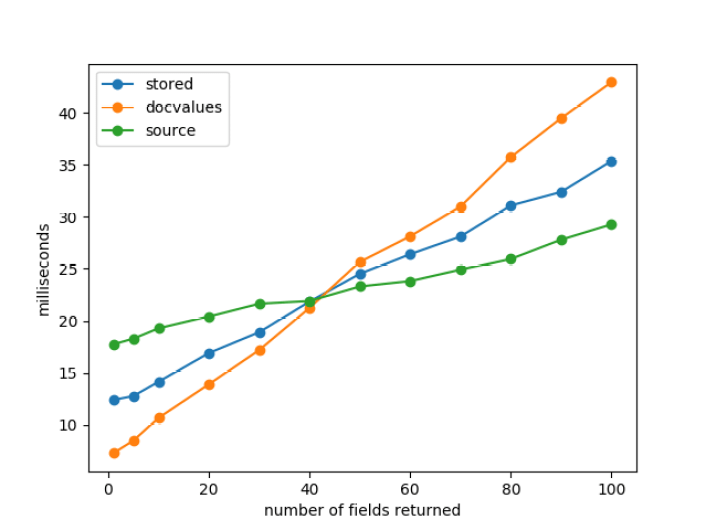
}

}

}

}

}



参考: <https://sease.io/2021/02/field-retrieval-performance-in-elasticsearch.html>

#### FieldData

doc values不支持Text这种分词字段, 如果要对分词字段进行聚合,elasticsearch提供了fielddata, 构建和管理都在内存中, 默认不使用fielddata.

如果要使用file data, 最好设置一个缓存大小限制,达到限制阈值,elasticsearch会通过least-recently-updated进行清理:

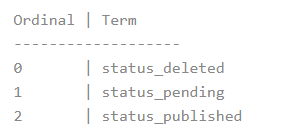
**indices.fielddata.cache.size**

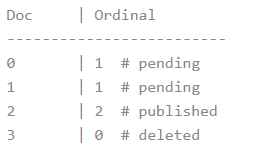
#### global originals

产生的原因:

Imagine that we have a billion documents, each of which has a status field. There are only three statuses: status\_pending, status\_published, status\_deleted. If we were to hold the full string status in memory for every document, we would use 14 to 16 bytes per document, or about 15 GB.

使用original:





This reduces memory usage from 15 GB to less than 1 GB!

使用场景:

* Certain bucket aggregations on keyword, ip, and flattened fields. This includes terms aggregations as mentioned above, as well as composite, diversified\_sampler, and significant\_terms.
* Bucket aggregations on text fields that require [fielddata](https://www.elastic.co/guide/en/elasticsearch/reference/master/fielddata.html) to be enabled.
* Operations on parent and child documents from a join field, including has\_child queries and parent aggregations.

注意:

The global ordinal mapping uses heap memory as part of the [field data cache](https://www.elastic.co/guide/en/elasticsearch/reference/master/modules-fielddata.html). Aggregations on high cardinality fields can use a lot of memory and trigger the [field data circuit breaker](https://www.elastic.co/guide/en/elasticsearch/reference/master/circuit-breaker.html#fielddata-circuit-breaker).

开启 global originals:

{

"properties": {

"tags": {

"type": "keyword",

"eager\_global\_ordinals": true

}

}

}

#### SuggestField:

该类field用来提供completion建议.供[CompletionQuery](#_CompletionQuery)(抽象类)使用.

Field is tokenized, not stored and stores documents, frequencies and positions.

suggest field会被分词,将数据编码到FST, 不会存储其他信息,用来做普通查询会查询不到.

#### Range field:

This field indexes dimensional ranges defined as min/max pairs. It supports up to a maximum of 4 dimensions (indexed as 8 numeric values). With 1 dimension representing a single integer range,  
\* 2 dimensions representing a bounding box, 3 dimensions a bounding cube, and 4 dimensions a tesseract.

此字段索引定义为最小/最大对的维度范围。它最多支持4个维度(索引为8个数值)。1维表示单个整数范围，2维表示边界框，3维表示边界立方体，4维表示超立方体。

### term

项,可以看作map的value,一个term是Lucene的基本搜索单元.

### term vector

项向量,通过分析器解析产生的信息, 关于term的词频,位置和偏移量(首字符和尾字符与原始字符串原点的偏移量)

Lucene在FileType中设置

作用: 高亮展示,辅助文档打分,文档相似性

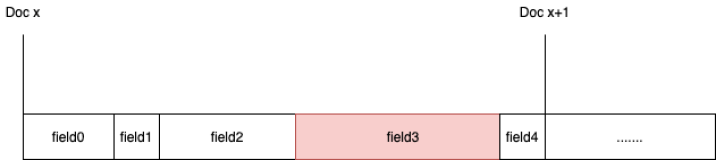
项向量会占用很多磁盘空间, Lucene也可以动态计算,在文档较少的时候可以这样做, 但是文档较多时, 这个过程是和索引阶段的analysis一样的,会消耗更多内存和CPU.

### Store

YES, 存储原始field值到index

NO, 不存储原始值到index

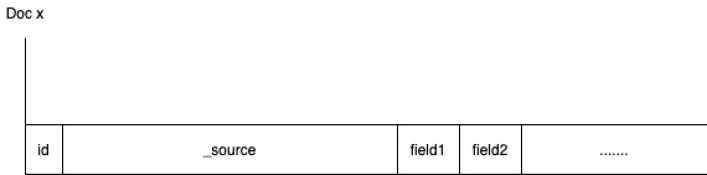
某些field只想用来搜索,不需要保存,则可以设置store为NO



elasticsearch默认field 的store为false,但是我们还是可以搜索出来,是因为elasticsearch在Lucene中添加了\_source字段,并默认设置为store为yes.

用\_source的原因:

Take as example the image above. In order to access the field3 of document x, we have to access the row of document x and skip all the fields that are stored before field3. Skipping a field require to get its length. Skipping fields is not as expansive as read them but this operation does not come for free.



#### 示例:

Lucene中只有Store设置为YES的field,在查询时才返回结果, 结果示例:

{"fields":[{"charSequenceValue":"has value"},{"charSequenceValue":"key310"},{"charSequenceValue":"text 410"}]}

elasticsearch中, 如果我们禁用\_source则查询结果中不会显示对应字段.禁用\_source,在查询时只会得到一个\_id

#### request:

{

"mappings": {

"\_source": {

"enabled": false

},

"properties" : {

"index\_store\_field" : {

"type" : "text",

"store" : true

},

"no\_index\_field" : {

"type" : "text",

"index" : false,

"store" : true

},

"no\_store\_field" : {

"type" : "text"

}

}

}

}

#### response:

{

"took" : 2,

"timed\_out" : false,

"\_shards" : {

"total" : 1,

"successful" : 1,

"skipped" : 0,

"failed" : 0

},

"hits" : {

"total" : {

"value" : 1,

"relation" : "eq"

},

"max\_score" : 1.0,

"hits" : [

{

"\_index" : "es-test",

"\_type" : "\_doc",

"\_id" : "1",

"\_score" : 1.0

}

]

}

}

如果某些字段只用来索引,不需要存储,则可以在\_source中排除,以节约空间:

"\_source": {

"excludes":[

"no\_store\_field"

]

}

### Index

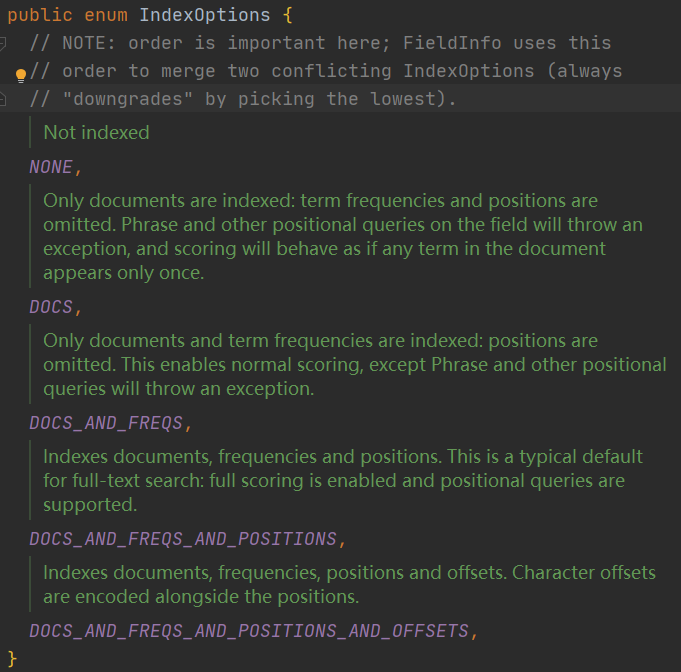
YES: 索引该field

NO: 不被索引

设置为no,则该字段不能被搜索

elasticsearch默认为true,如果设置为false,则用该字段查询时会报错.

### IndexOptions:



一共有5种IndexOptions

对应elasticsearch中的docs,freqs,positions,offsets. 没有NONE,index如果设置为false,则代表不索引.

IndexOptions和term vector比较像, 区别:

IndexOptions: field > term > doc > freq/pos/offset

term vector: doc > field > term > freq/pos > offset

### boost

boost可以影响文档的评分,比如我们想要title中含有a的文档要比content中含有a的更靠前

boost可以分为:索引时boost和查询时boost,新版本(5.0.0以后)Lucene已经废除index time boost 即setBoost方法,用BoostQuery代替,查询时boost相对更加灵活.

elasticsearch 查询示例:

{

"match": {

"title": {

"query": "xx",

"boost": 2

}

}

},

{

"match": {

"content": "xx"

}

}

### norm

Lucene中叫做Normalization Factor

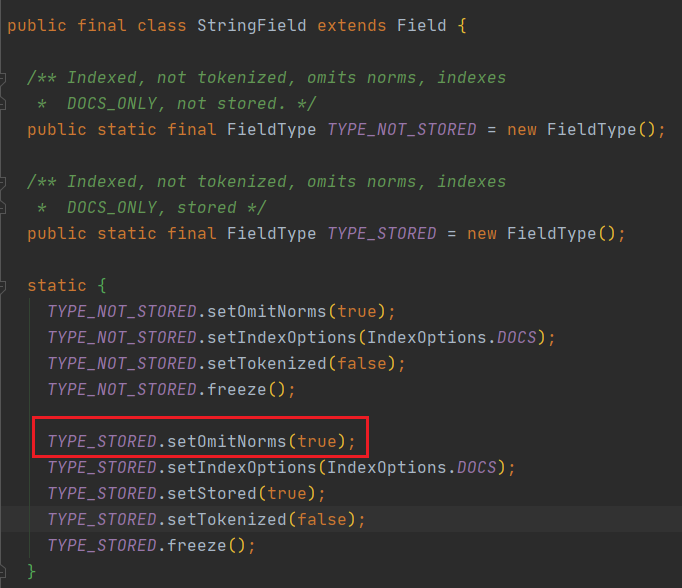
norm(t,d) is an index-time boost factor that solely depends on the number of tokens of this field in the document, so that shorter fields contribute more to the score.

一个创建索引时的boost因子,field越短,则评分会越高.

公式:

norm(d) = 1 / √numTerms 字段长度归一值（ norm ）是字段中词数平方根的倒数。

norm在StringField类型中被设置为忽略,表示field的长度不影响评分.



elasticsearch中进行设置:

{

"properties": {

"title": {

"type": "text",

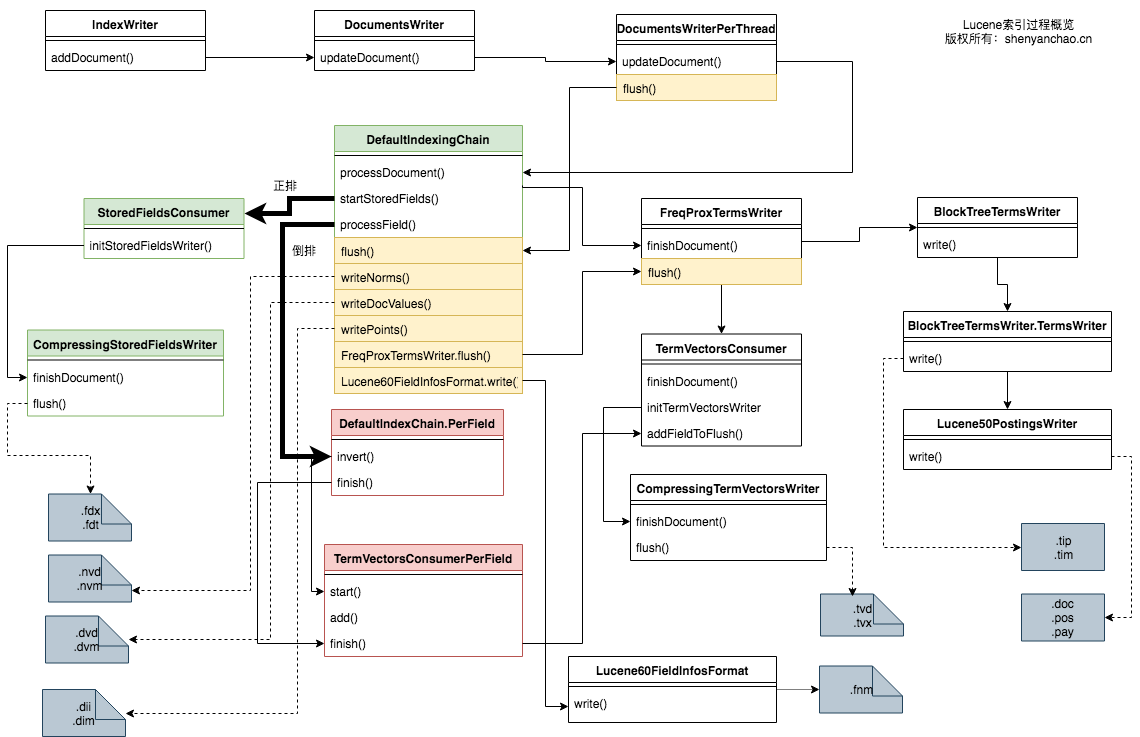
"norms": false

}

}

}

# Lucene索引过程



Lucene的添加操作通过update方法来实现,传入一个null的Term(field+value).

Lucene的update操作是通过先删除后增加来实现的,通过两阶段提交(prepareCommit, commit, rollback)来保证update的原子性.

delete操作,Lucene提供了by query和by term

当一个文档被删除或更新(delete+add),Lucene在bitset中标记文档被删除(生成liv文件),所有后续搜索会过滤这些被标记的文档.

删除文档并不会立即回收占用的磁盘空间, 直到段合并后，被删除的文档所消耗的字节才会被回收。这种方法减少更新Lucene的索引数据和每次删除文档时的聚合统计数据的成本.

Lucene的删除操作解释:

<https://www.elastic.co/cn/blog/lucenes-handling-of-deleted-documents>

## elasticsearch添加, 索引和删除document

elasticsearch中:



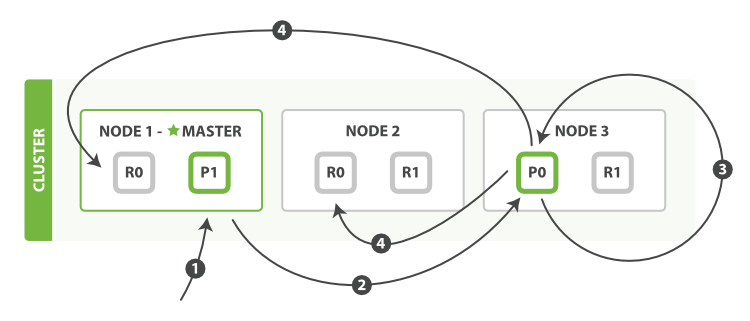
1. client发送一个 请求到node1(master)
2. node根据document的\_id,决定是否属于shard0, 然后将请求发送到node3(shard0的主分片所在)
3. node3在primary shard上执行请求, 如果成功,请求会并行的发送到node1 和node2 上的replica shards, 所有的replica shards返回成功后, node3返回成功给协调node,然后返回client.

### 批量添加document

如果有大量document要添加,使用bulk api 可以减少网络往返的耗时.

最佳批处理大小取决于许多因素:文档大小和复杂性、索引和搜索负载，以及集群可用的资源。从1000到5000个文档的批处理和5MB到15MB的总有效负载开始,是一个不错的选择。从那里，你可以尝试找到最佳位置。

## elasticsearch修改document



1. client发送请求到node1.
2. 转发请求到primary shard所在的node3
3. node3从primary shard中检索文档, 删除旧文档,添加新文档.

## flush

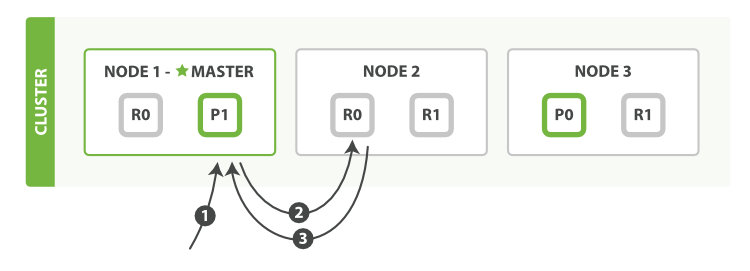
满足flush条件后, 通过flush方法把内存中的数据写入磁盘. 如果Lucene进程没有正常关闭,或者设备故障, 内存中的数据会丢失. elasticsearch通过写trans log的形式,保证数据的安全性.

# Lucene查询过程

核心类:IndexSearcher,也用到了IndexReader(作用是操作Directory,打开索引,提供底层文件读取的工作).

创建的IndexSearcher读取索引已有的快照,如果索引发生变更,如果没有新打开一个reader,搜索不会发生变化.

在elasticsearch中:



1. client发送数据到node1(master)
2. 节点使用文档的\_id来确定文档是否属于分片0。所有三个节点上都存在分片0的副本。在这种情况下，它将请求转发给node2
3. node2返回document给node1, node1返回文档给client

### 查询类:

#### TermQuery:

对特定项进行搜索,如果类型为Text,且进行分词操作,会返回包含的文档,如果为String类型或者没有进行分词,则匹配等于term的文档.

#### TermRangeQuery:

搜索开始字母范围从x到y的文档,true/false表示是否包含x或y

#### NumbericRangeQuery:

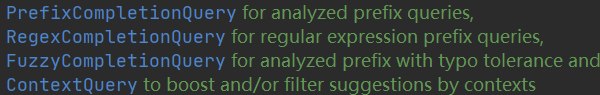
和TermRangeQuery类似,新版本已废除.

#### PrefixQuery:

前缀查询,FST结构

#### CompletionQuery

有4中类型:



completion query用到的数据结构是加权FST, 满足FST的查询特性

在elasticsearch中的使用:

{

"mappings": {

"properties": {

"suggest": {

"type": "completion"

},

"title": {

"type": "keyword"

}

}

}

}

#### BooleanQuery

private static int *maxClauseCount* = 1024;

嵌入和组合其他的query,最多允许1024个组合

MUST: 交集

FILTER: 交集

SHOULD: 并集.

MUST NOT: 不等于.

##### 一些区别

默认情况下，Elasticsearch按相关性得分对匹配的搜索结果进行排序，该得分衡量每个文档与查询的匹配程度。

在filter和must\_not上下文中，查询子句回答问题“此文档是否与此查询子句匹配？答案是简单的“是”或“否”-不计算分数。

常用的过滤器将由Elasticsearch自动缓存，以提高性能。

elasticsearch用must对比filter:

###### filter

GET /source-ip/\_search

{

"query": {

"bool": {

"filter": [

{

"term": {

"countryName.keyword": "South Korea"

}

}

]

}

},

"size": 10000

}

结果:



###### must



如果不计算相关性得分,最好使用filter,性能更好

#### BoostQuery

自定义评分因子,可以设置某个条件的评分更高

#### FuzzQuery:

基于levenshtein最短编辑距离查询,会枚举所有指定的编辑距离值的词汇,性能开销很大. 新版本改进用levenshtein automaton性能有一定提高,慎用.

#### PhraseQuery

对于词汇级别的编辑距离查询.

#### MatchALLDocsQuery:

不走倒排索引,默认不评分不排序,如果全量评分会导致性能很差.

#### DisjunctionMaxQuery

\* If the query is "albino elephant" this ensures that "albino" matching one field and "elephant" matching  
\* another gets a higher score than "albino" matching both fields.

同样影响评分, 如果我们查询的是albino elephant,则我们可以让匹配一个albino和一个elephant的分数高于只含有两个albino的文档

#### ConstantScoreQuery

A query that wraps another query and simply returns a constant score equal to  
\* 1 for every document that matches the query.  
\* It therefore simply strips of all scores and always returns 1.

包装其他查询, 使查询结果的每个document的score都为1,忽略文档评分,效率会有所提高

elasticsearch可以调整boost参数,但是每个document的score都是统一值:

{

"query": {

"constant\_score": {

"filter": {

"term": {

"index\_store\_field": "3"

}

},

"boost": 1.0

}

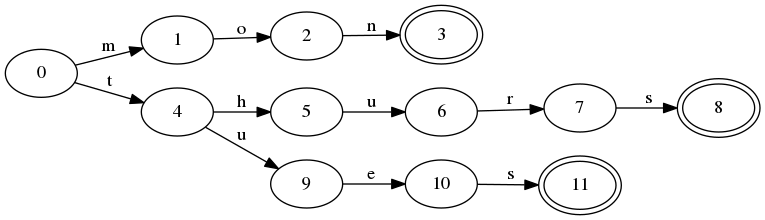
}

}

### FST索引

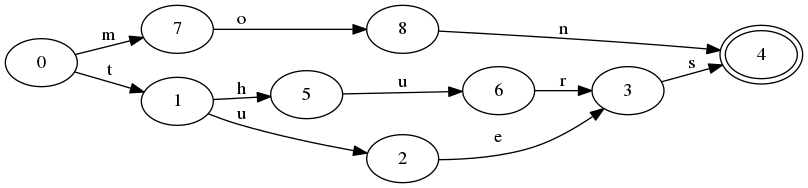
#### Trie结构

单词树结构,共享前缀



#### FSA结构

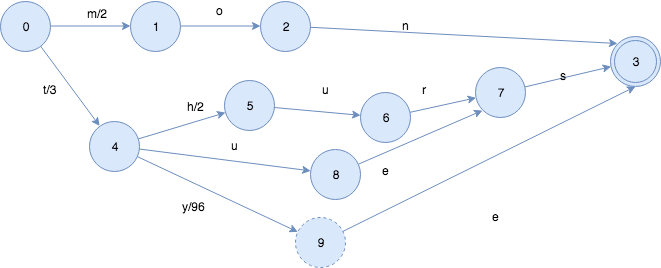
finite state acceptor(有限状态接收器),与Trie的区别在于FSA共享后缀,空间比Trie占用更少,构建也更复杂.



#### FST结构

FST(finite state transducer) 有限状态转换器

FST和FSA很相似,不同点在于FST给定一个key除了能知道是否存在,还能输出一个关联的值.



FST优点:

1. 不仅可以查询是否存在,还可以做值的转换. key-> value
2. 通过对词典中单词前缀和后缀的重复利用,压缩了存储空间
3. 可以方便实现范围,前缀,后缀查询,时间复杂度O(len(str))

缺点:

1. 构建较为复杂,单词必须先排序.

#### Lucene提供的FST工具

String inputValues[] 
long outputValues[] 
{"cat", 
"dog", 
12}; 
"dogs 
Positivelntoutputs outputs 
builder 
Positivelntoutputs . getSingleton(); 
new (INPUT_TYPE.BYTEI, 
outputs); 
BytesRef scratchBytes = new BytesRef(); 
IntsRefBuilder scratchlnts = new IntsRefBuilder(); 
@; i &lt; inputValues.length; i++) 
for (int i 
scratchBytes . 
builder.add(Util.toIntsRef(scratchBytes, scratchlnts), 
builder.finish(); 
fst 
outputValues[i]); 
Retrieval by key: 
<pre class="prettyprint"> 
Util.get(fst, 
Long value 
System . out .println(1'alue); 
Retrieval by value: 
<pre class="prettyprint"> 
new BytesRef("dog")); 
// Only works because outputs are also in sorted order 
Util.getBy0utput(fst, 12); 
IntsRef key 
System.out scratchBytes) .utf8ToString()); 
// dogs 
Iterate over key-value pairs in sorted order: 
<pre class="prettyprint"> 
// Like TermsEnum, this also supports seeking (advance) 
new BytesRefFSTEnum&lt;Long&gt; (fist); 
BytesRefFSTEnum&lt;Long&gt; iterator 
while (iterator.next() != null) { 
iterator. current(); 
mapEntry 
System . out .println(mapEntry . input . utf8ToString()) ; 
System . out .println(mapEntry . output) ; 

注意事项:

必须根据Unicode code排序(UTF8或UTF32). Java中的String.compareTo是根据UTF16排序,会导致结果不正确.

// Input values (keys). These must be provided to Builder in Unicode code point (UTF8 or UTF32) sorted order 
// Note that sorting by Java's String.compareTo, which is UTF16 sorted order, is not correct and can lead to 
exceptions while building the EST: 

#### term排序方式

##### Radix sort 基数排序

是一种非比较型整数排序算法，其原理是将整数按位数切割成不同的数字，然后按每个位数分别比较。由于整数也可以表达字符串（比如名字或日期）和特定格式的浮点数，所以基数排序也不是只能使用于整数。

 实现：将所有待比较数值（正整数）统一为同样的数位长度，数位较短的数前面补零。然后，从最低位开始，依次进行一次排序。这样从最低位排序一直到最高位排序完成以后，数列就变成一个有序序列。

 时间复杂度O(kn)

空间复杂度O(k+n)

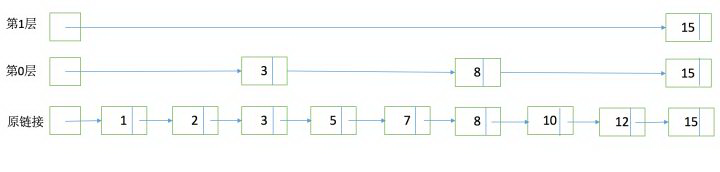
k为比特位数 n为要比较的数的个数.

##### Introsort内省排序

从快速排序开始，当递归深度超过一定深度（深度为排序元素数量的对数值）后转为堆排序,如果数据量较少,则使用二分排序.

### posting lists 倒排表结构:

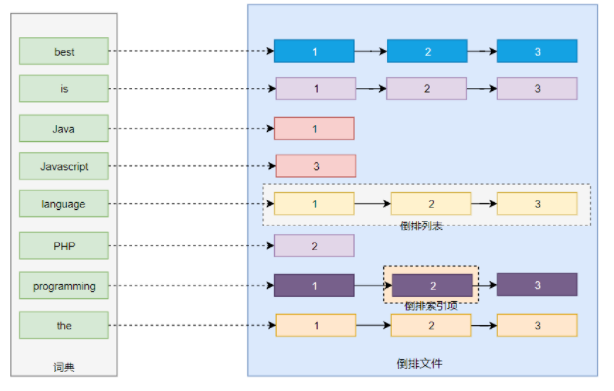
一般结构:



Lucene中的实现在MultiLevelSkipListWriter及其子类

\* For each skip point, we will record:   
\* 1. docID in former position, i.e. for position 12, record docID[11], etc.  
\* 2. its related file points(position, payload),   
\* 3. related numbers or uptos(position, payload).  
\* 4. start offset.

Lucene中的使用:



### BKD tree

#### Lucene数值类型演化:

参考:

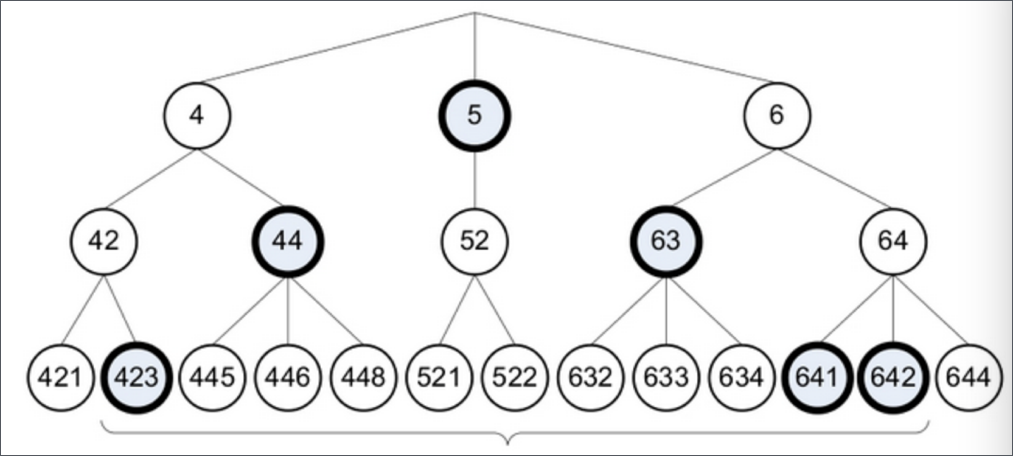
<https://www.elastic.co/cn/blog/apache-lucene-numeric-filters>

##### text时代:

在Lucene之前版本(2.9前),任何数据都是text类型. 例如索引: 17,2, 23, 要让2排序以后在17之前, 就统一位数,在2左边补0. 这样虽然能够满足数值和范围搜索,但是补0浪费了索引的空间,其次如果要索引一个10位数,但是之前的索引只有9位,则需要重新索引.在范围查询的时候会遍历大量的数据,会导致查询缓慢.

##### number tries:

2.9版本开始,使用一种number类型的trie来做数值范围搜索



当我们要查询符合[423,642]区间的数据时，我们只需要遍历number tries，匹配存在图中黑色加粗标识的几个数字即可。图中即为423，44，5，63，641，642, 这样查询的数量级大大的降低. 但是这样的number tries非常大, 另外number trie的分词是ASCII标记, 不兼容Lucene4.0的索引任何二进制标记的能力.

##### Geo-spatial:

Lucene6.0开始,为了改进地理空间搜索,使用了BKD(block K dimension) tree结构, 用于快速多维点过滤, Lucene最初的目标是使用bkd-tree来快速搜索2D和3D的地理空间, 后来发现这种数据结构也非常适合一维(一维的bkd-tree退化为二叉树)数字的情况.

##### bdk-tree论文:

<https://users.cs.duke.edu/~pankaj/publications/papers/bkd-sstd.pdf>

#### bkd-tree优点:

bkd-tree取代了弃用的数字字段和数字范围查询,因为它具有更好的总体性能和更通用的特性.

#### bkd-tree缺点:

构建比较复杂,并且一旦构建完成,再进行增删改会几乎改变整棵树的数据.

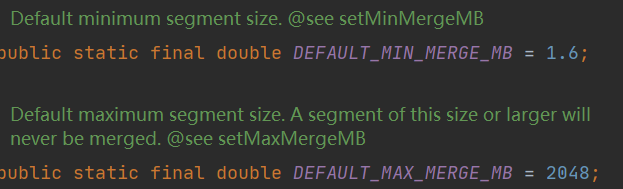
#### Lucene中的bkd tree:



一个points类型的field最多支持8个纬度.

# segments的合并过程

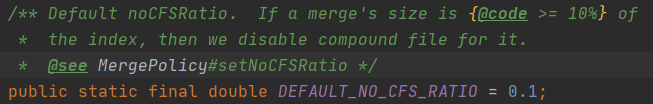
索引的合并目的是为了提高搜索速度, 在合并期间,会占用较多的磁盘空间,大约为segment初期的3倍, 所以merge前,需要保证有足够的磁盘空间, 当结束操作后,占用的空间会比初期少. 一个segment达一定规模后,就不再进行merge操作.



默认segment达到1.6M才会被merge. 达到2GB就不再merge(IndexWriter.forceMerge可能还会触发merge).

组成segment的文档分为混合型(多种文件放在一起)和多文件类型(一种类型一个文件).默认为cfs类型.

如果一次merge的大小达到索引的10%,则会使用多文件类型:



merge分为三个阶段,第一阶段和第三阶段需要获取锁,但是执行很快,第二阶段不用获取锁,但是比较耗时,有利于Lucene多线程merge操作.

1. 达到merge条件
2. 初始化merge设置,这一步很快,但是需要持有IW的同步锁.
3. 确保所有即将合并的segments的delete操作都被处理: 1. 将delGen小于等于当前maxDelGen的FrozenBufferedUpdates加入waitFor set(因为这个删除适用于正在合并的段,我们不需等待它被处理) 2. 将delete by term/query,或文档update操作冻结的包转换为它们在索引中的实际docIds，并应用更改。这是一个繁重的操作，由进入的索引线程并发地完成.
4. Writes all doc values updates to disk if there are any. 将挂起的文档值更新操作发送到磁盘,否则新合并的段将看不到它们
5. 计算并获取新的segment的name,主要是得到\_N.
6. 开始执行耗时的实际merge工作,但是不用获取IW的同步锁.
7. 包装Directory,使其具有限速功能
8. merge middle实现:

1. 获取文件reader

2. SegmentReader继承至CodecReader, 一个MergeReader包括一个SegmentReader和一个Bits(一个类似BitSet的结构)

3. 生成一个SegmentMerger(包括CodecReader集合,SegmentInfo,InfoStream,Directory,fieldNumbers,IOContext)

4. 调用SegmentMerger的merge方法(merge主要发生的地方,返回MergeState)

5. 合并所有segment的fieldInfos,合并为fnm文件

6. 合并所有segment的stored类型fields, 默认实现跳过deleted documents. 如果index有序,docId用堆结构存储,原因是堆部分排序,以便能在常数时间找到最小的元素.否则list存储.

7. 合并norms

8. 根据norms合并terms

9. 合并docValues,在磁盘上存储数据排序

10. 合并points类型索引,bkd tree实现.

11. 合并vectors

finally merge:

删除旧文件,更新segment信息.

### 合并的内容:

#### 正向信息:

.fnm文件的数据在mergeFieldInfos方法中操作,FieldInfos.Builder类的add方法调用addOrUpdateInternal,添加或修改FieldInfo.

FieldInfos的参数都是final类型,所以如果field一开始有norm factor,后面再取消norm factor合并后也会失效

private 
private 
private 
private 
private 
private 
private 
private 
private 
final 
final 
final 
final 
final 
final 
final 
final 
final 
boolean 
boolean 
boolean 
boolean 
boolean 
boolean 
boolean 
boolean 
hasFreq; 
hasProx; 
hasPayloads; 
hasOffsets; 
hasVectors; 
hasNorms; 
hasDocValues; 
hasPointVaIues; 
String softDeIetesFieId; 

 fdx/fdt文件的合并,在CompressingStoredFieldsWriter的merge方法中执行,如果两个segment的fieldinfo完全相同(field个数和name,如果排序索引,则field的顺序信息也相同),则可以把一个segment的fdt内容直接拷贝到另一个后面,如果不一样,则会对每篇文档进行添加,这样合并效率会变得很低,所以同一个索引不建议不同的文档有不同的field.

* If all readers are compressed and they have the same fieldinfos then we can merge the serialized document 
* directly. 
subs = 
new ArrayList<>(); 
for(int eState.storedFieldsReaders.len th •i++) { 
if (matching.matchingReaders[i] && 
mergeState. storedFieldsReaders [i] instanceof CompressingStoredFieIdsReader) { 
ompressxng ore le s ea er s ore xe s ea er = 
storedF ieIdsReader . checklntegrity ( ) ; 
ompressxng 
ore le s eader) mergeState.storedFieIdsReaders[i] ; 
mergeState.docMaps[i], mergeState.maxDocs[i])); 
subs.add(new CompressingStoredFieIdsMergeSub(storedFieIdsReader, 
} else { 
return super.merge(mergeState); 

 合并norm factor

对每一个域,用指向合并段的reader读出标准化因子,然后再写入新生成的段.

* 
* 
* 
* 
* 
Merges in the fields from the readers in 
<code>mergeState</code>. The default implementation 
calls {@link #mergeNormsField} for each field, 
filling segments with missing norms for the field with zeros. 
Implementations can override this method 
for more sophisticated merging (bulk-byte copying, etc). 
public void merge(MergeState mergeState) throws IOException { 
for(NormsProducer normsProducer . 
mergeState.normsProducers) { 
if (normsProducer null) -i 
normsProducer . checklntegrity() ; 
for (Fieldlnfo mergeFieIdInfo . 
mergeState.mergeFieIdInfos) { 
if (mergeFieIdInfo.hasNorms()) 4 
mergeNormsFieId(mergeFieIdInfo, mergeState) ; 

 合并term vector:

通常情况也是直接拷贝,如果要合并的段包含特殊的field,就需要合并每个文档.

#### 反向信息:

合并词典和倒排表:

在倒排表中,文档号要求从小到大顺数排序,在每个segment中,文档号都是从0开始编号.

反向信息的合并包含两部分

1. 对字典的合并,需要对词典中的term进行重新排序
2. 对于相同的term,对包含此term的文档号列表进行合并,需要对文档号重新编号.

对于文档重新编号,相对容易,两个段合并的时候,第二个编号都增加一个偏移量(前一个文档的个数)

对词典term的合并,需要排序, 需要找出两个段中相同的词,Lucene通过PriorityQueue来实现, queue按照字典顺序排序,DocIDMerger<T extends DocIDMerger.Sub>保存要合并的段的词典以及倒排表信息,在queue中用来排序的key是它代表的段中的第一个term.

* 
* 
Utility class to help merging documents from sub-readers according to either simple 
concatenated (unsorted) order, or by a specified index-time sort, skipping 
deleted documents and remapping non-deleted documents. 
public abstract class DocIDMerger<T extends DocIDMerger.Sub> { 
Represents one sub-reader being merged 
public static abstract class Sub { 
Mapped doc ID 
public int mappedDocID; 
final MergeState.DocMap docMap; 
Sole constructor 
public Sub(MergeState.DocMap docMap) { 
this.docMap = docMap; 
Returns the next document ID from this sub reader, 
public abstract int nextDoc() throws IOException; 
and {@link DocIdSetIterator#N0_MORE_DOCS} when done 

帮助根据简单的连接(未排序)顺序或指定的索引时排序从子阅读器合并文档，跳过已删除的文档并重新映射未删除的文档。

Maps document IDs from old segments to document IDs in the new segment 
public final DocMap[] docMaps; 

docMaps的作用是将旧segments中的文档ID映射到新的segment中.

子类SortedDocIDManager的实现

private static class SortedDocIDMerger<T extends DocIDMerger.Sub> extends DocIDMerger<T> { 
private final List<T> subs; 
private final PriorityQueue<T> qoeoe; 
private SortedDocIDMerger(List<T> subs, int maxCount) throws IOException { 
this.subs = 
subs; 
new PriorityQueue<T>(maxCount) 
Clljelje = 
@Override 
protected boolean lessThan(Sub a, Sub b) { 
assert a.mappedDocID b.mappedDocID; 
return a.mappedDocID < b.mappedDocID; 
reset(); 

for(T sub . 
subs) 
if (first) { 
// by setting mappedDocID 
-1, this entry is guaranteed to be the top of the Clljeoe 
// so the first call to next() will advance itl 
sub.mappedDocID = -1 • 
first = 
false; 
} else { 
int mappedDocID; 
while (true) { 
int docID = sub.nextDoc() 
if (dociD { 
mappedDocID - 
break; 
sub. docMap.get(docID); 
mappedDocID = 
if (mappedDocID ' - 
break; 
if (mappedDocID 
// all docs in this sub were deleted; 
continue; 
sub.mappedDocID = mappedDocID; 
add(sub) ; 
qoeoe. 
do not add it to the qoeoe! 

# 评分机制:

### TF-IDF

优点:

简单,快速.

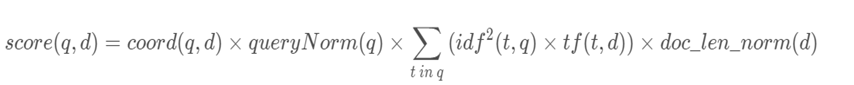
缺点:

依赖词频, 词频超过一定数量级, 相关度不够准确.

TF-IDF有两层意思，一层是"词频"（Term Frequency，缩写为TF），另一层是"逆文档频率"（Inverse Document Frequency，缩写为IDF）

TF-IDF的优点是简单快速，而且容易理解。缺点是有时候用词频来衡量文章中的一个词的重要性不够全面，有时候重要的词出现的可能不够多，而且这种计算无法体现位置信息，无法体现词在上下文的重要性。

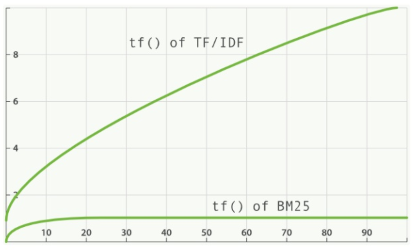
评分公式:

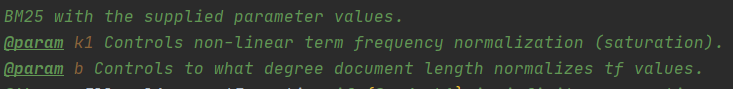


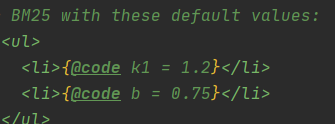
### BM25

BM25算法是对TF\_IDF算法的改进, 对于 TF-IDF 算法，TF(t) 部分的值越大，整个公式返回的值就会越大。

BM25 就针对这点进行来优化，随着TF(t) 的逐步加大，该算法的返回值会趋于一个数值。







BM25 有一个比较好的特性就是它提供了两个可调参数：

**k1**

这个参数控制着词频结果在词频饱和度中的上升速度。默认值为 1.2 。值越小饱和度变化越快，值越大饱和度变化越慢。

**b**

这个参数控制着字段长归一值所起的作用， 0.0 会禁用归一化， 1.0 会启用完全归一化。默认值为 0.75 。

### es中的配置

properties": {

"title": {

"type": "string",

"similarity": "BM25"

},

"body": {

"type": "string",

"similarity": "default"

}

}

default为TF-IDF算法评分规则.

#### 对bm25参数的调整

参考:

https://www.elastic.co/guide/cn/elasticsearch/guide/current/pluggable-similarites.html#bm25-tunability

"settings": {

"similarity": {

"bm25\_similarity": {

"type": "BM25",

"b": 0

}

}

}

#### 配置DFR:

"similarity" : {

"dfr\_similarity" : {

"type" : "DFR",

"basic\_model" : "g",

"after\_effect" : "l",

"normalization" : "h2",

"normalization.h2.c" : "2.0"

}

}

#### 配置IB:

"similarity" : {

"ib\_similarity" : {

"type" : "IB",

"distribution" : "ll",

"lambda" : "df",

"normalization" : "z",

"normalization.z.z" : "0.25"

}

}

# elasticsearch部分

GET /

查看elasticsearch的一些信息

{

"name" : "",

"cluster\_name" : "",

"cluster\_uuid" : "aXrSW96ZhxZw",

"version" : {

"number" : "7.5.0",

"build\_flavor" : "oss",

"build\_type" : "tar",

"build\_hash" : "e9ccaed468e2fac2275a3761849cbee64b39519f",

"build\_date" : "2019-11-26T01:06:52.518245Z",

"build\_snapshot" : false,

"lucene\_version" : "8.3.0",

"minimum\_wire\_compatibility\_version" : "6.8.0",

"minimum\_index\_compatibility\_version" : "6.0.0-beta1"

},

"tagline" : "You Know, for Search"

}

## 节点类型:

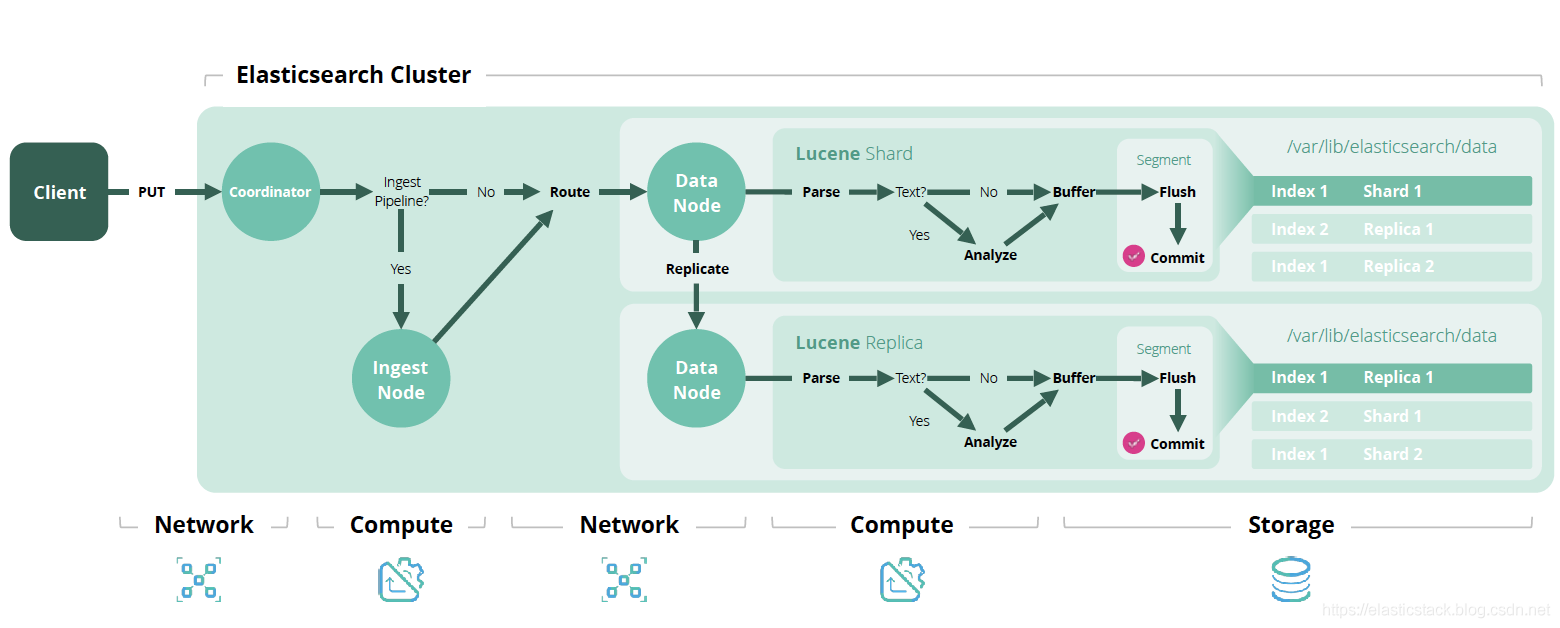
如果在elasticsearch.yml中指定了节点的角色,则该节点知会被指定为该角色, 如果没有设置,则可能被动态分配

node.roles: [ data, master, voting\_only ]

注意: Only nodes with the master role can be marked as having the voting\_only role.

参考:

<https://www.elastic.co/guide/en/elasticsearch/reference/master/modules-node.html#modules-node>



### master(主节点)

负责管理elasticsearch集群, 负责索引的创建/删除, 管理其他节点上的shard.

### data node(数据节点)

存放数据,负责数据的增删改查,聚合排序. 对机器配置要求较高.

### ingest node(提取节点)

在数据写入data node之前对数据进行一些处理.

示例:

PUT \_ingest/pipeline/test\_ingtest

{

"description": "test ingtest",

"processors": [

{

"uppercase": {

"field": "name"

}

}

]

}

使用:

PUT /es-test\_3/\_doc/10?pipeline=test\_ingtest

{

"name": "the letters are all lowercase"

}

### Remote-eligible node

连接其他集群,用于跨集群搜索,跨集群同步数据.

node.roles: [ remote\_cluster\_client ]

### coordinating node(协调节点)

每一个节点都是一个潜在的协调节点, 如果一个节点设置的node.roles是一个空list, 它将只会充当coordinating node, 且它不能被禁用. 协调节点最大的作用就是转发请求,将各个shard里面的数据汇集起来一并返回给客户端, 因此coordinating node需要有足够的CPU和内存去处理gather阶段.

node.roles: [ ]

### remote\_cluster\_client

用于跨集群搜索和跨集群请求

## heap的设置

By default, Elasticsearch automatically sizes JVM heap based on a node’s roles and total memory. We recommend this default sizing for most production environments. If needed, you can override default sizing by manually setting the heap size.

如果要自定义heap:

* Set Xmx and Xms to no more than 50% of your total system memory. Elasticsearch requires memory for purposes other than the JVM heap and it is important to leave space for this. For instance, Elasticsearch uses off-heap buffers for efficient network communication, relies on the operating system’s filesystem cache for efficient access to files, and the JVM itself requires some memory too. It is normal to observe the Elasticsearch process using more memory than the limit configured with the Xmx setting.
* Set Xmx and Xms to no more than the threshold that the JVM uses for compressed object pointers (compressed oops). The exact threshold varies but is near 32 GB. You can verify that you are under the threshold by looking for a line in the logs like the following:

heap size [1.9gb], compressed ordinary object pointers [true]

* Set Xmx and Xms to no more than the threshold for zero-based compressed oops. The exact threshold varies but 26GB is safe on most systems and can be as large as 30GB on some systems. You can verify that you are under this threshold by starting Elasticsearch with the JVM options -XX:+UnlockDiagnosticVMOptions -XX:+PrintCompressedOopsMode and looking for a line like the following:

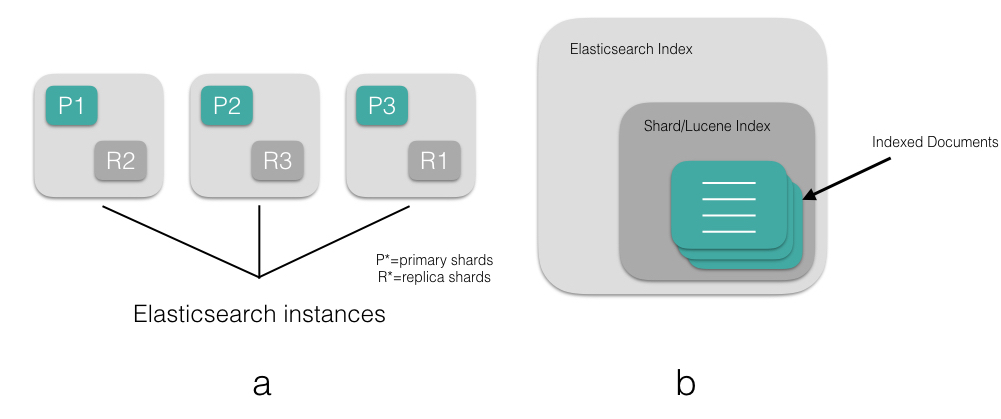
heap address: 0x000000011be00000, size: 27648 MB, zero based Compressed Oops

This line shows that zero-based compressed oops are enabled. If zero-based compressed oops are not enabled, you’ll see a line like the following instead:

heap address: 0x0000000118400000, size: 28672 MB, Compressed Oops with base: 0x00000001183ff000

The more heap available to Elasticsearch, the more memory it can use for its internal caches. This leaves less memory for the operating system to use for the filesystem cache. Larger heaps can also cause longer garbage collection pauses.

## shard



elasticsearch的shard分为primary shard和replica shard. 一个shard是一个Lucene的index.

一个index可以分成多个shard, 有利于提高性能, 可以想象成关系型数据库中的分表,

replica shard是primary shard的数据备份. 既可以提高数据可用性,也可以提高查询吞吐量.

elasticsearch的primary shard数量在创建的时候指定,后面不能修改, replica shard数量可以修改, 一个document被放在哪个shard,elasticsearch通过shard = hash(routing) % number\_of\_primary\_shards来确定, 默认情况routing是document的\_id, 可以指定.

越多的shard查询起来越快, 但是维护更耗性能,所以一个准则是每GB空间不超过20个shard.

### shard 查询

Nodes and shards used for the search. By default, Elasticsearch selects from eligible nodes and shards using [adaptive replica selection](https://www.elastic.co/guide/en/elasticsearch/reference/current/search-shard-routing.html#search-adaptive-replica), accounting for [allocation awareness](https://www.elastic.co/guide/en/elasticsearch/reference/current/modules-cluster.html#shard-allocation-awareness).

**\_only\_local**

Run the search only on shards on the local node.

**\_local**

If possible, run the search on shards on the local node. If not, select shards using the default method.

**\_only\_nodes:<node-id>,<node-id>**

Run the search on only the specified nodes IDs. If suitable shards exist on more than one selected nodes, use shards on those nodes using the default method. If none of the specified nodes are available, select shards from any available node using the default method.

**\_prefer\_nodes:<node-id>,<node-id>**

If possible, run the search on the specified nodes IDs. If not, select shards using the default method.

**\_shards:<shard>,<shard>**

Run the search only on the specified shards. This value can be combined with other preference values, but this value must come first. For example: \_shards:2,3|\_local

**<custom-string>**

Any string that does not start with \_. If the cluster state and selected shards do not change, searches using the same <custom-string> value are routed to the same shards in the same order.

示例: 指定为local:

GET /my-index-000001/\_search?preference=\_local

{

"query": {

"match": {

"user.id": "kimchy"

}

}

}

## Translog

可以类比MySQL的binlog, 目的是保证数据安全性.

## elasticsearch as a time series data store

refer to: <https://www.elastic.co/cn/blog/elasticsearch-as-a-time-series-data-store>

# 附录:

## 构建一棵kd-tree:

1、初始化分割轴：对每个维度的数据进行方差的计算，取最大方差的维度作为分割轴，标记为r；  
2、确定节点：对当前数据按分割轴维度进行检索，找到中位数数据，并将其放入到当前节点上；  
3、划分双支：  
 划分左支：在当前分割轴维度，所有小于中位数的值划分到左支中；  
 划分右支：在当前分割轴维度，所有大于等于中位数的值划分到右支中。  
4、更新分割轴：r = (r + 1) % k;  
5、确定子节点：  
 确定左节点：在左支的数据中进行步骤2；  
 确定右节点：在右支的数据中进行步骤2；

### 二维kd-tree构建示例：

{（2,3），（5,4），（9,6），（4,7），（8,1），（7,2）}

#### 1、初始化分割轴：

 计算结果x轴的方差较大，所以，最开始的分割轴为x轴。

#### 2、确定当前节点：

 对{2，5，9，4，8，7}找中位数，发现{5,7}都可以，这里我们选择7，也就是(7,2);

#### 3、划分双支数据：

 在x轴维度上，比较和7的大小，进行划分：

 左支：{(2,3)，(5,4)，(4,7)}

 右支：{(9,6)，(8,1)}

#### 4、更新分割轴：

 一共就两个维度，所以，下一个维度是y轴。

#### 5、确定子节点：

左节点：在左支中找到y轴的中位数(5,4)，左支数据更新为{(2,3)}，右支数据更新为{(4,7)}

 右节点：在右支中找到y轴的中位数(9,6)，左支数据更新为{(8,1)}，右支数据为null。

#### 6、更新分割轴：

 下一个维度为x轴。

#### 7、确定(5,4)的子节点：

 左节点：由于只有一个数据，所以，左节点为(2,3)

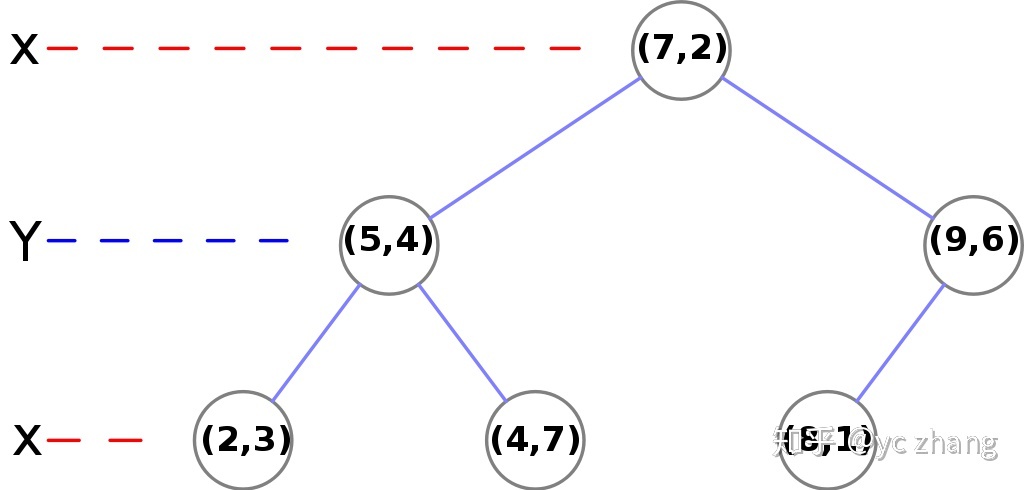
 右节点：由于只有一个数据，所以，右节点为(4,7)

#### 8、确定(9,6)的子节点：

 左节点：由于只有一个数据，所以，左节点为(8,1)

 右节点：右节点为空。

 最终，就可以构建整个的kd-tree。



### kd-tree的最近邻检索:

我们只需要定位到对应的分支上，找到最接近的点就可以了。

示例：查找(2.1,3.1)的最近邻。

1. 计算当前节点(7,2)的距离，为6.23，并且暂定为(7,2)，根据当前分割轴的维度（2.1 < 7），选取左支。
2. 计算当前节点(5,4)的距离，为3.03，由于3.03 < 6.23，暂定为(5,4)，根据当前分割轴维度（3.1 < 4），选取左支。
3. 计算当前节点(2,3)的距离，为0.14，由于0.14 < 3.03，暂定为(2,3)，根据当前分割轴维度（2.1 > 2），选取右支，而右支为空，回溯上一个节点。
4. 计算(2.1,3.1)与(5,4)的分割轴{y = 4}的距离，如果0.14小于距离值，说明就是最近值。如果大于距离值，说明，还有可能存在值与(2.1,3.1)最近，需要往右支检索。

由于0.14 < 0.9，我们找到了最近邻的值为(2,3)，最近距离为0.14。

1. 多个近邻其实和一个最近邻类似，不过是存储区间变为了多个，判定方法还是完全一样。

## elasticsearch中的Zen Discovery & Gossip 算法