Elasticsearch-深入理解索引原理

最近开始大面积使用ES，很多地方都是知其然不知其所以然，特地翻看了很多资料和大牛的文档，简单汇总一篇。内容多为摘抄，说是深入其实也是一点浅尝辄止的理解。希望大家领会精神。

首先学习要从官方开始地址如下。

es官网原文：https://www.elastic.co/guide/en/elasticsearch/reference/current/docs-index_.html#index-refresh

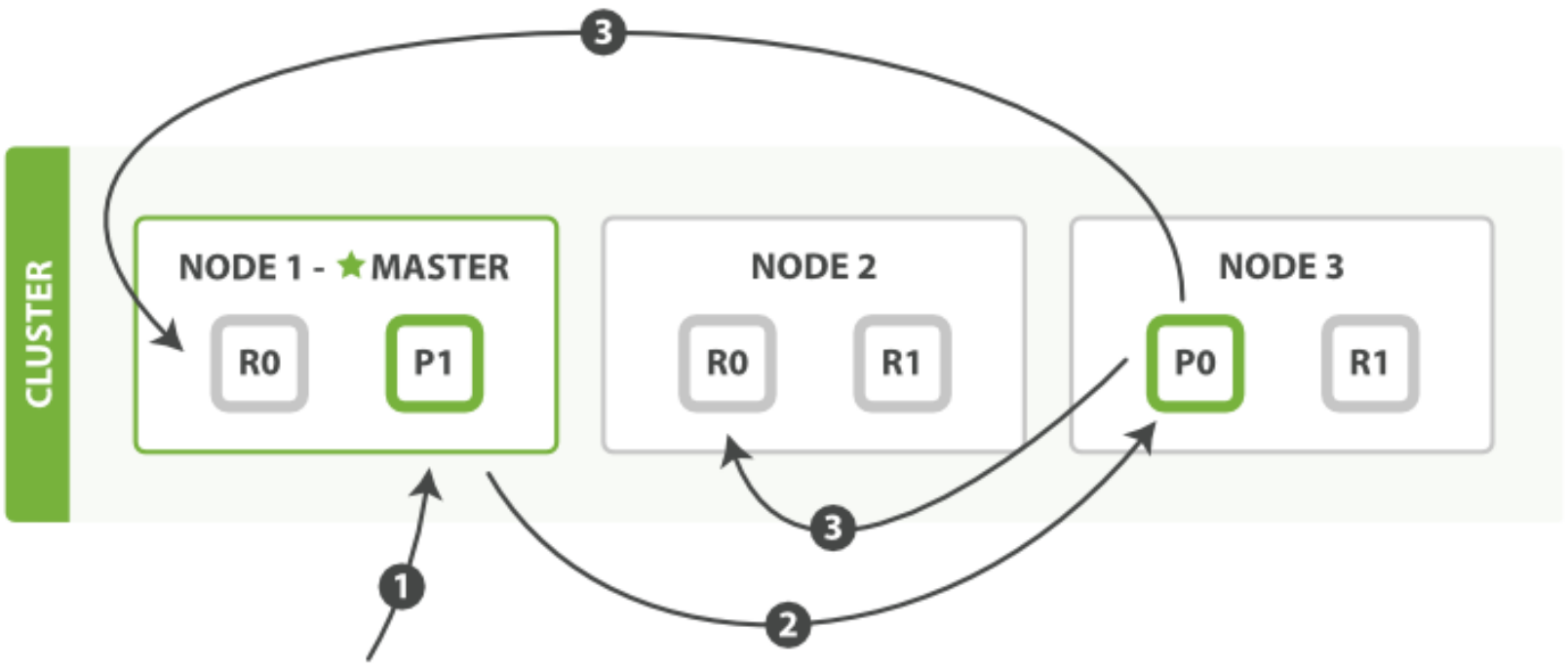
索引(Index)

ES将数据存储于一个或多个索引中，索引是具有类似特性的文档的集合。类比传统的关系型数据库领域来说，索引相当于SQL中的一个数据库，或者一个数据存储方案(schema)。索引由其名称(必须为全小写字符)进行标识，并通过引用此名称完成文档的创建、搜索、更新及删除操作。一个ES集群中可以按需创建任意数目的索引。

如果不懂这块可以看我的写的上一篇入门的内容
http://www.cnblogs.com/wenBlog/p/8482326.html

我们了解索引的写操作后可知，更新、索引、删除文档都是写操作，这些操作必须在primary shard完全成功后才能拷贝至其对应的replicas上，默认情况下主分片等待所有备份完成索引后才返回客户端。

Figure 9. Creating, indexing, or deleting a single document



步骤:

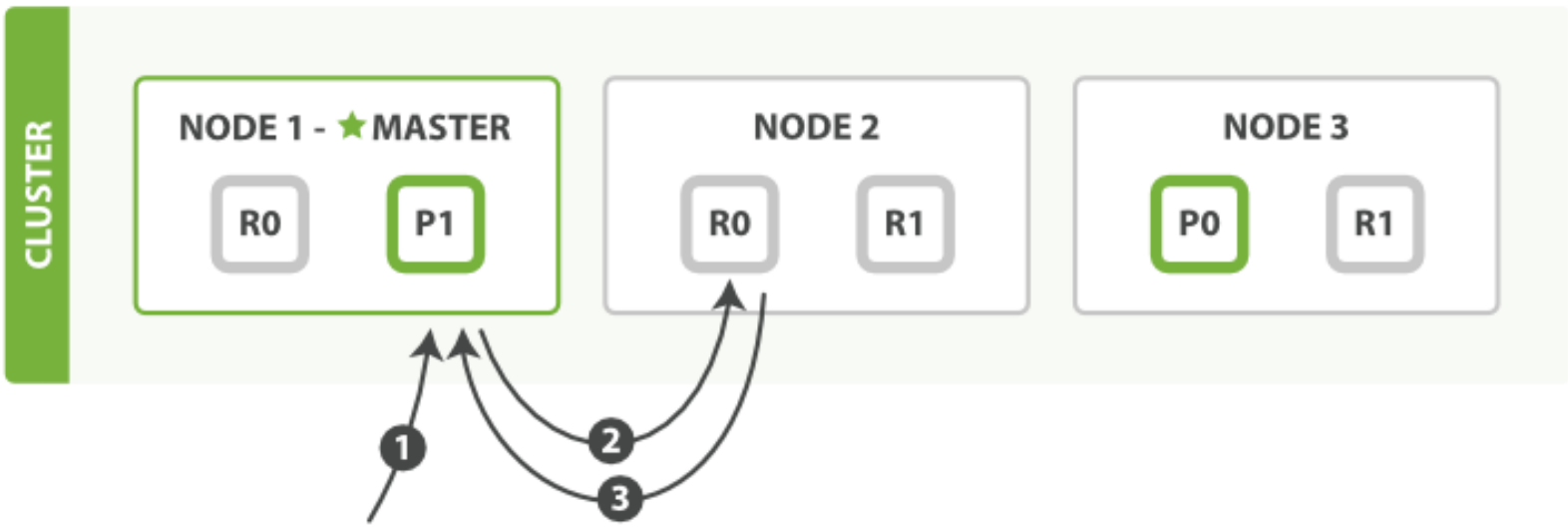
1. 客户端向Node1 发送索引文档请求
2. Node1 根据文档ID(_id字段)计算出该文档应该属于shard0，然后请求路由到Node3的P0分片上
3. Node3在P0上执行了请求。如果请求成功，则将请求并行的路由至Node1，Node2的R0上。当所有的Replicas报告成功后，Node3向请求的Node(Node1)发送成功报告，Node1再报告至Client。

当客户端收到执行成功后，操作已经在Primary shard和所有的replica shards上执行成功了

读操作

一个文档可以在primary shard和所有的replica shard上读取。见Figure10

Figure 10. Retrieving a single document



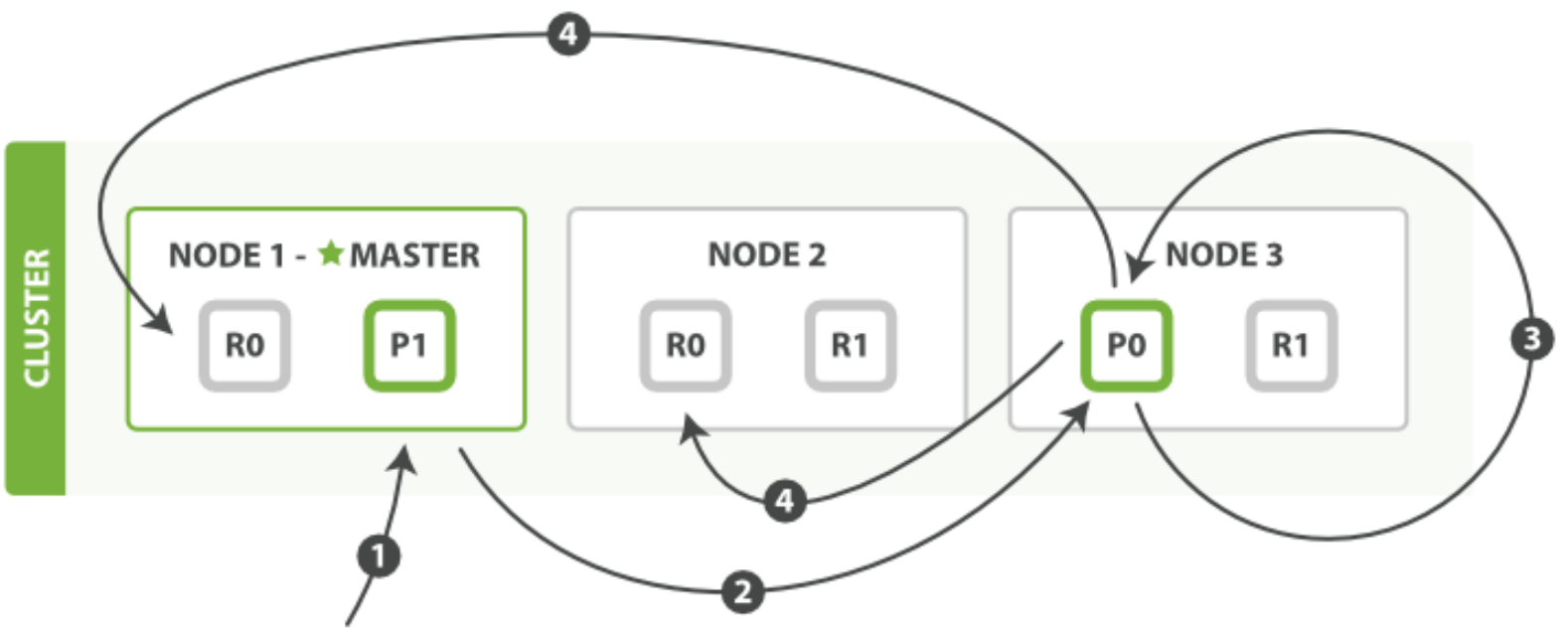
读操作步骤：

> 1.客户端发送Get请求到NODE1。
> 2.NODE1使用文档的_id决定文档属于shard 0.shard 0的所有拷贝存在于所有3个节点上。这次，它将请求路由至NODE2。
> 3.NODE2将文档返回给NODE1，NODE1将文档返回给客户端。 对于读请求，请求节点(NODE1)将在每次请求到来时都选择一个不同的replica。
> shard来达到负载均衡。使用轮询策略轮询所有的replica shards。

更新操作

更新操作，结合了以上的两个操作：读、写。见Figure11

Figure 11. Partial updates to a document



步骤：

> 1.客户端发送更新操作请求至NODE1
> 2.NODE1将请求路由至NODE3，Primary shard所在的位置
> 3.NODE3从P0读取文档，改变*source字段的JSON内容，然后试图重新对修改后的数据在P0做索引。如果此时这个文档已经被其他的进程修改了，那么它将重新执行3步骤，这个过程如果超过了retry*on_conflict设置的次数，就放弃。
> 4.如果NODE3成功更新了文档，它将并行的将新版本的文档同步到NODE1和NODE2的replica shards重新建立索引。一旦所有的replica shards报告成功，NODE3向被请求的节点(NODE1)返回成功，然后NODE1向客户端返回成功。

2.6 SHARD

本节将解决以下问题：

- 为什么搜索是实时的
- 为什么文档的CRUD操作是实时的
- ES怎么保障你的更新在宕机的时候不会丢失
- 为什么删除文档不会立即释放空间

2.6.1不变性

写到磁盘的倒序索引是不变的：自从写到磁盘就再也不变。 这会有很多好处：

> 不需要添加锁。如果你从来不用更新索引，那么你就不用担心多个进程在同一时间改变索引。
> 一旦索引被内核的文件系统做了Cache，它就会待在那因为它不会改变。只要内核有足够的缓冲空间，绝大多数的读操作会直接从内存而不需要经过磁盘。这大大提升了性能。
> 其他的缓存(例如fiter cache)在索引的生命周期内保持有效，它们不需要每次数据修改时重构，因为数据不变。
> 写一个单一的大的倒序索引可以让数据压缩，减少了磁盘I/O的消耗以及缓存索引所需的RAM。

当然，索引的不变性也有缺点。如果你想让新修改过的文档可以被搜索到，你必须重新构建整个索引。这在一个index可以容纳的数据量和一个索引可以更新的频率上都是一个限制。

2.6.2动态更新索引

如何在不丢失不变形的好处下让倒序索引可以更改？答案是：使用不只一个的索引。 新添额外的索引来反映新的更改来替代重写所有倒序索引的方案。 Lucene引进了per-segment搜索的概念。一个segment是一个完整的倒序索引的子集，所以现在index在Lucene中的含义就是一个segments的集合，每个segment都包含一些提交点(commit point)。见Figure16。新的文档建立时首先在内存建立索引buffer，见Figure17。然后再被写入到磁盘的segment，见Figure18。

Figure 17. A Lucene index with new documents in the in-memory buffer, ready to commit

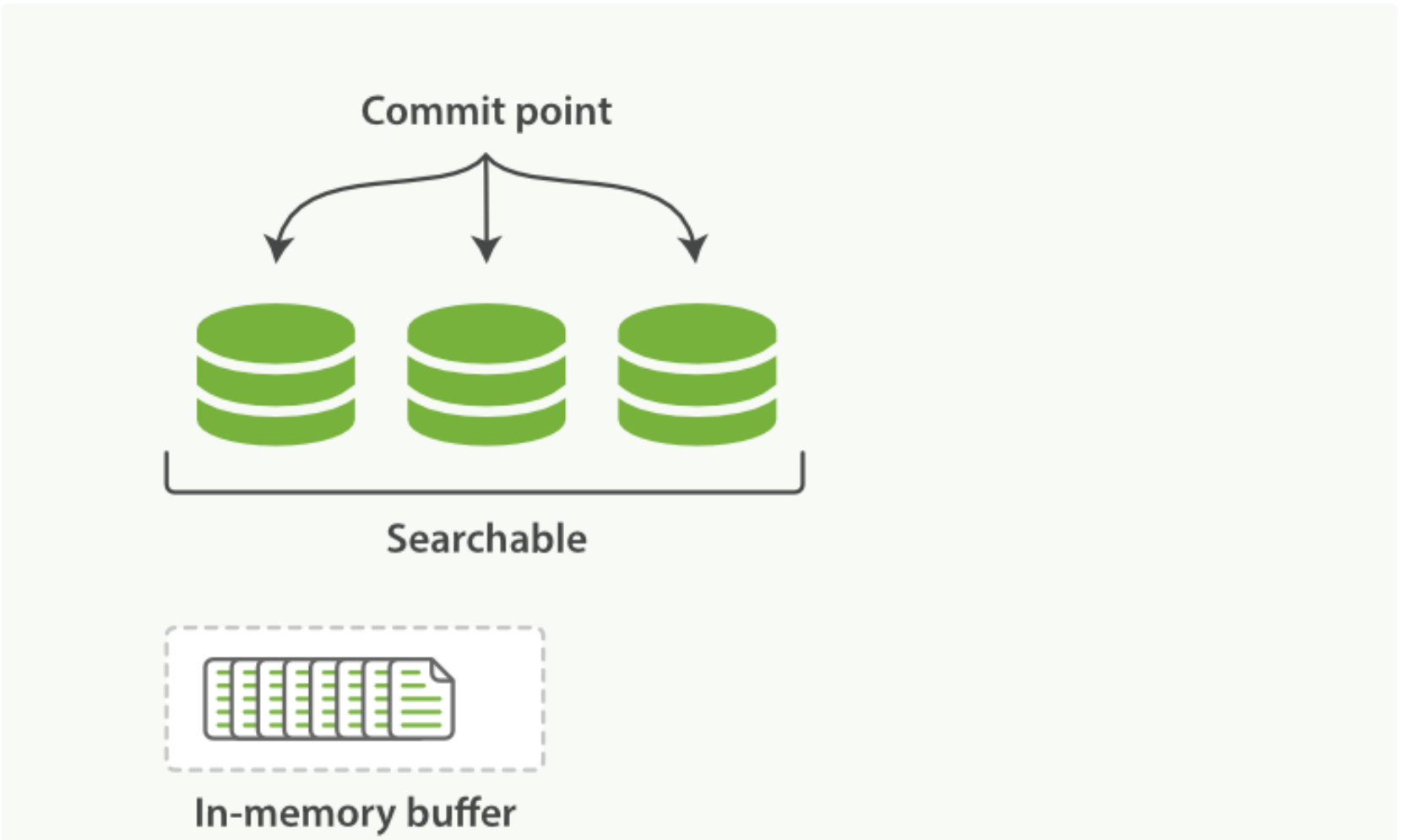
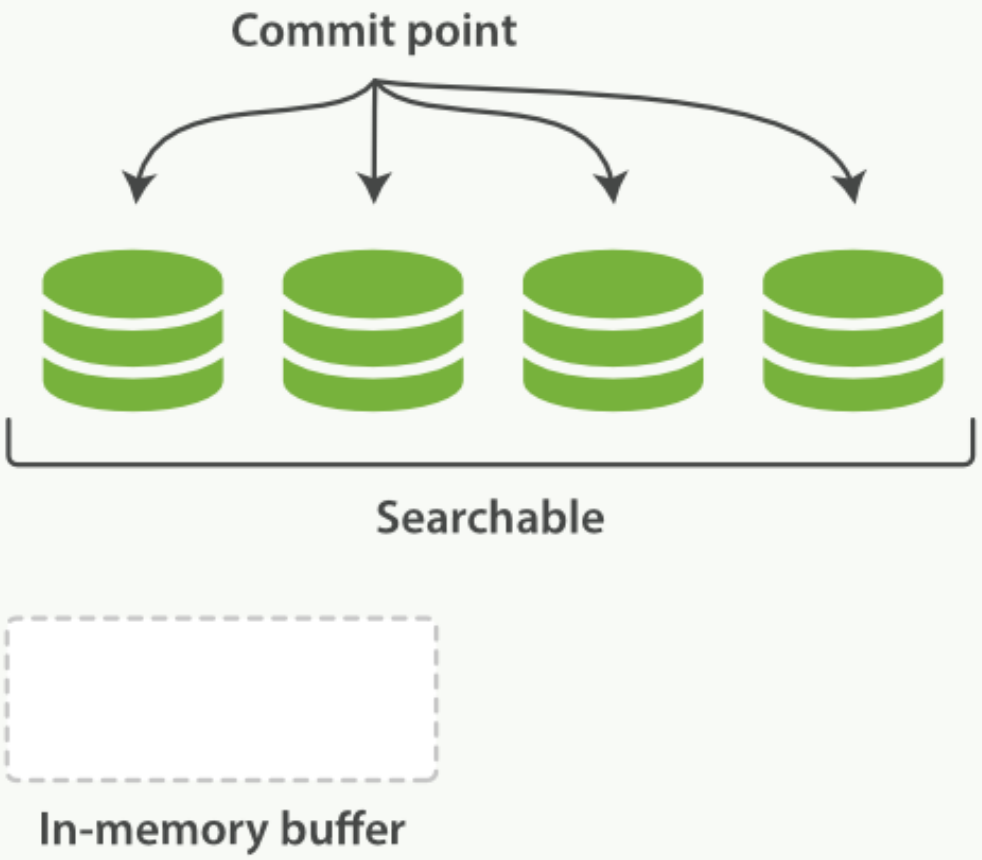


Figure 18. After a commit, a new segment is added to the commit point and the buffer is cleared



一个per-segment的工作流程如下：

> 1.新的文档在内存中组织，见Figure17。
> 2.每隔一段时间，buffer将会被提交： 一个新的segment(一个额外的新的倒序索引)将被写到磁盘 一个新的提交点(commit point)被写入磁盘，将包含新的segment的名称。 磁盘fsync，所有在内核文件系统中的数据等待被写入到磁盘，来保障它们被物理写入。
> 3.新的segment被打开，使它包含的文档可以被索引。
> 4.内存中的buffer将被清理，准备接收新的文档。

当一个新的请求来时，会遍历所有的segments。词条分析程序会聚合所有的segments来保障每个文档和词条相关性的准确。通过这种方式，新的文档轻量的可以被添加到对应的索引中。

删除和更新

segments是不变的，所以文档不能从旧的segments中删除，也不能在旧的segments中更新来映射一个新的文档版本。取之的是，每一个提交点都会包含一个.del文件，列举了哪一个segmen的哪一个文档已经被删除了。 当一个文档被”删除”了，它仅仅是在.del文件里被标记了一下。被”删除”的文档依旧

可以被索引到，但是它将会在最终结果返回时被移除掉。

文档的更新同理：当文档更新时，旧版本的文档将会被标记为删除，新版本的文档在新的segment中建立索引。也许新旧版本的文档都会本检索到，但是旧版本的文档会在最终结果返回时被移除。

2.6.3实时索引

在上述的per-segment搜索的机制下，新的文档会在分钟级内被索引，但是还不够快。 瓶颈在磁盘。将新的segment提交到磁盘需要fsync来保障物理写入。但是fsync是很耗时的。它不能在每次文档更新时就被调用，否则性能会很低。 现在需要一种轻便的方式能使新的文档可以被索引，这就意味着不能使用fsync来保障。 在ES和物理磁盘之间是内核的文件系统缓存。之前的描述中,Figure19,Figure20，在内存中索引的文档会被写入到一个新的segment。但是现在我们将segment首先写入到内核的文件系统缓存，这个过程很轻量，然后再flush到磁盘，这个过程很耗时。但是一旦一个segment文件在内核的缓存中，它可以被打开被读取。

Figure 19. A Lucene index with new documents in the in-memory buffer

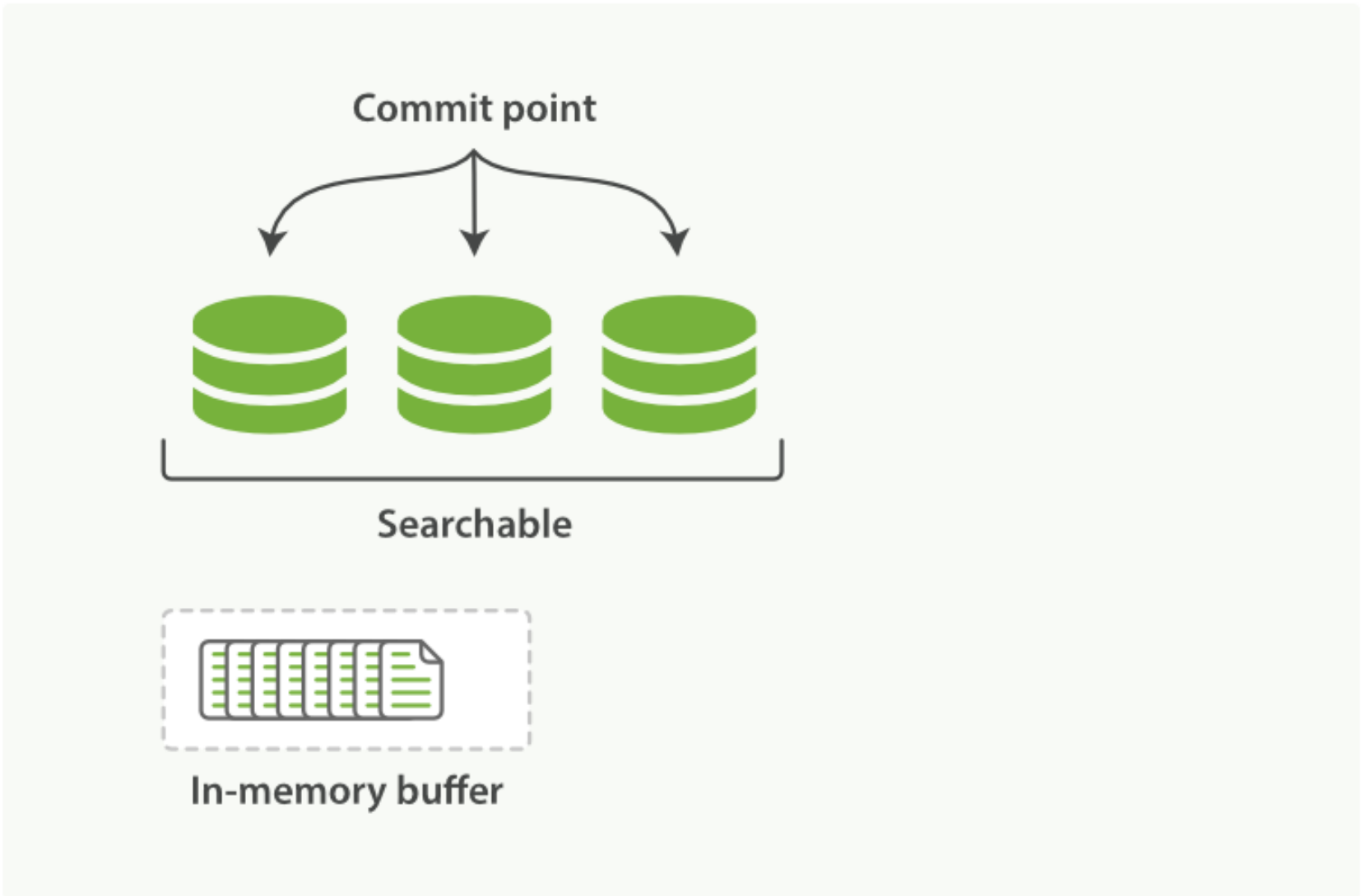
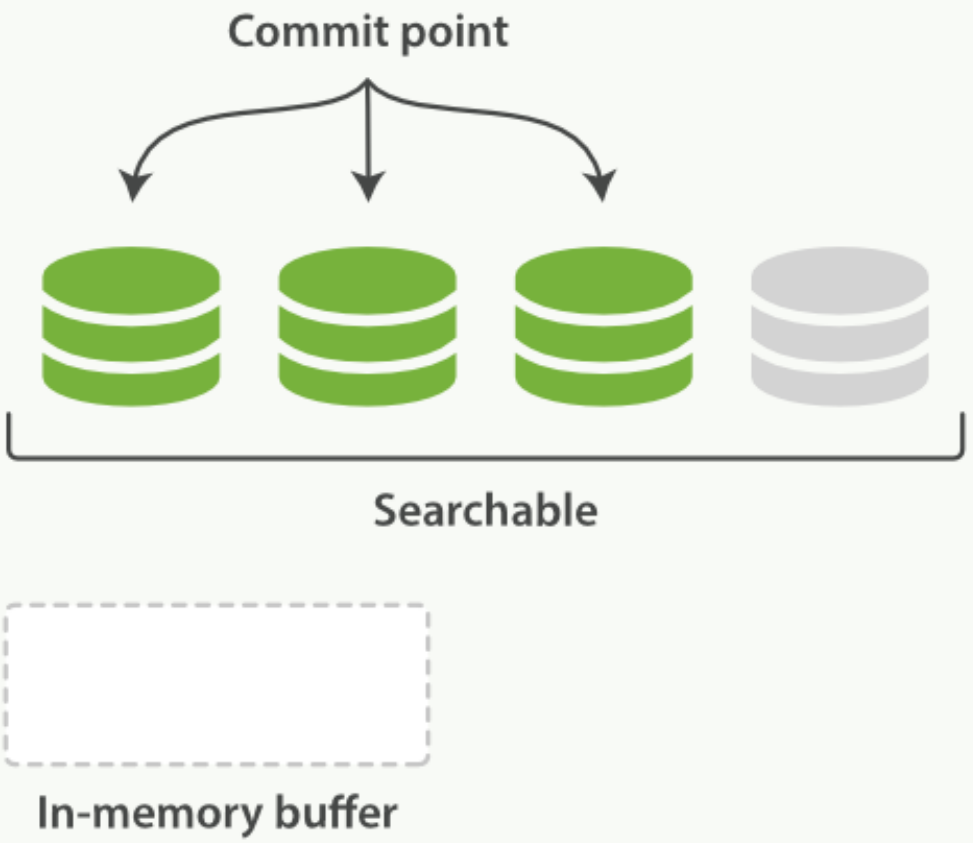


Figure 20. The buffer contents have been written to a segment, which is searchable, but is not yet commited

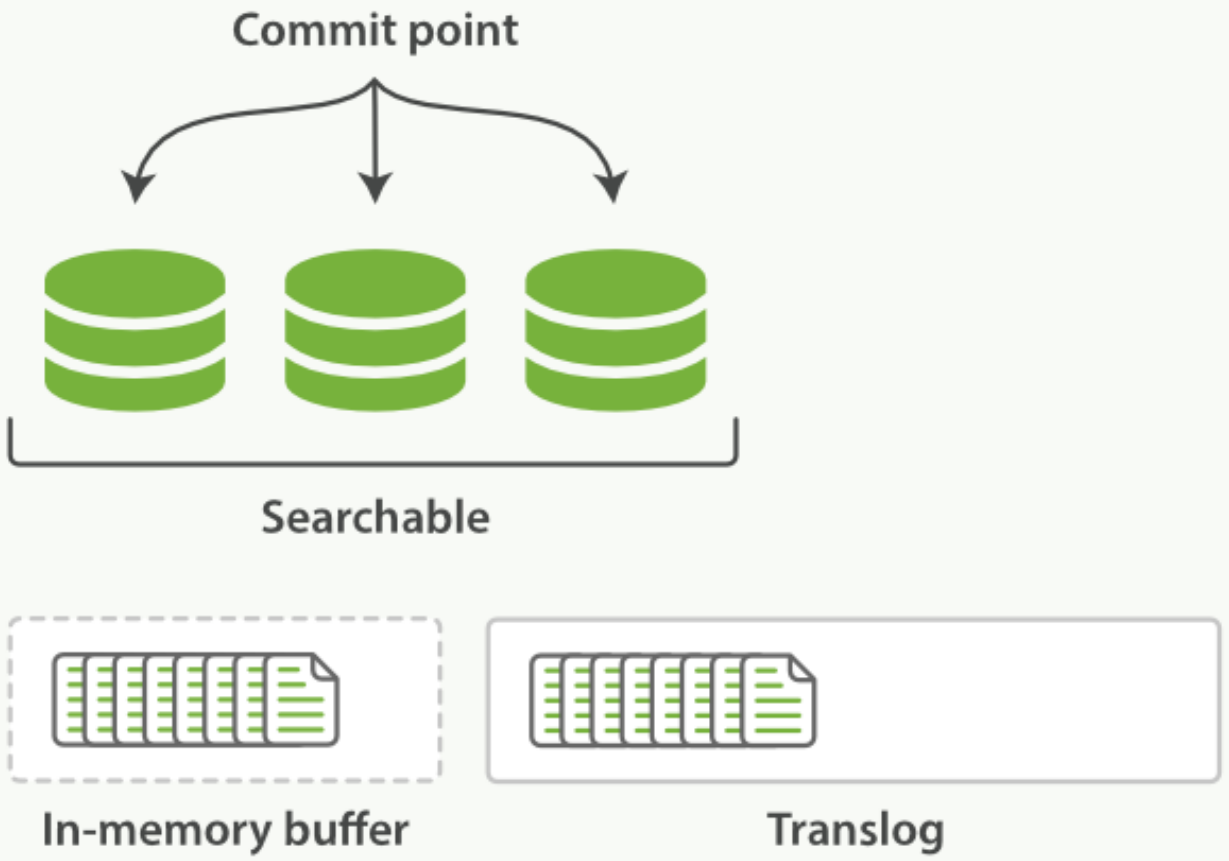


2.6.4更新持久化

不使用fsync将数据flush到磁盘，我们不能保障在断电后或者进程死掉后数据不丢失。ES是可靠的，它可以保障数据被持久化到磁盘。 在2.6.2中，一个完全的提交会将segments写入到磁盘，并且写一个提交点，列出所有已知的segments。当ES启动或者重新打开一个index时，它会利用这个提交点来决定哪些segments属于当前的shard。 如果在提交点时，文档被修改会怎么样? 不希望丢失这些修改:

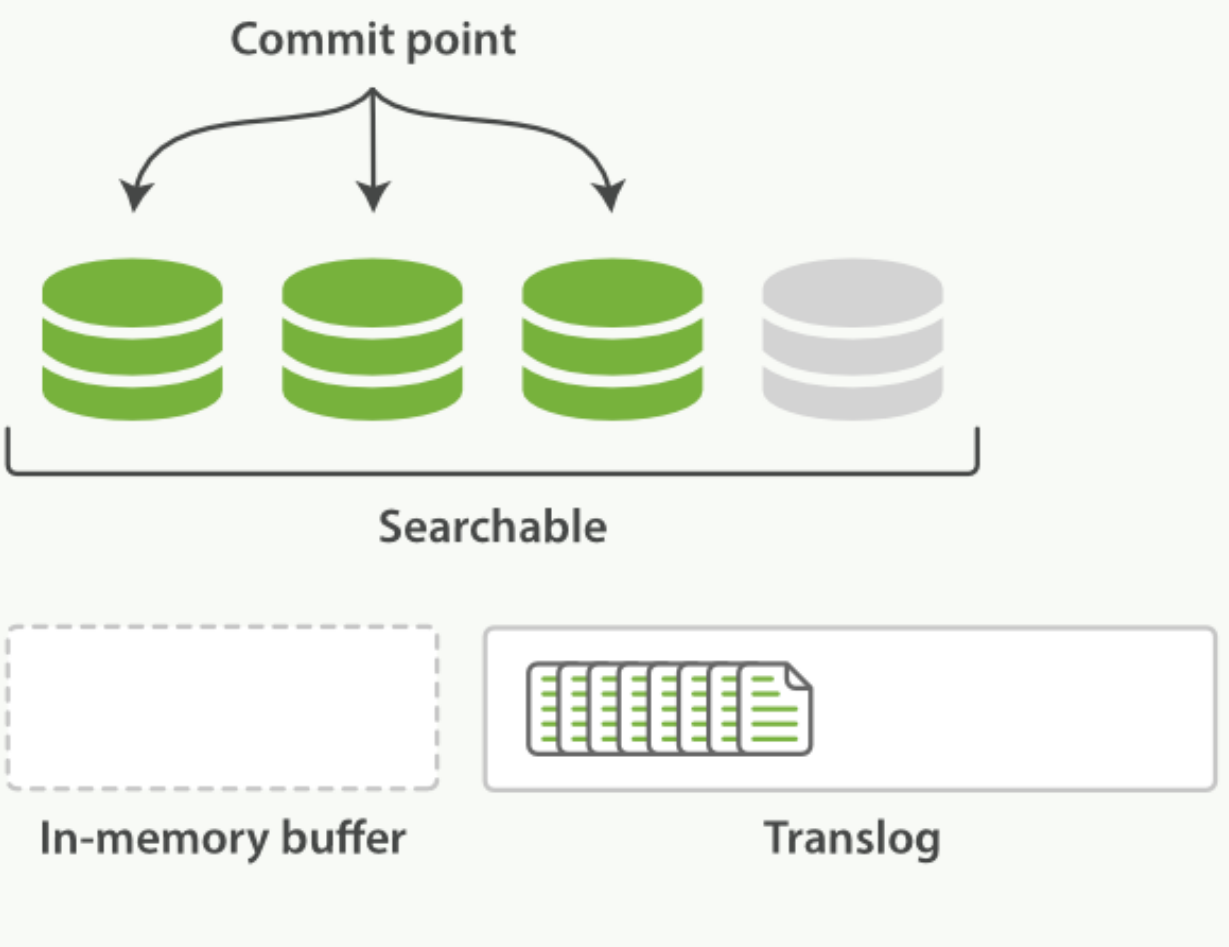
> 1.当一个文档被索引时，它会被添加到in-memory buffer，并且添加到Translog日志中，见Figure21.

Figure 21. New documents are added to the in-memory buffer and appended to the transaction log



2.refresh操作会让shard处于Figure22的状态：每秒中，shard都会被refreshed：

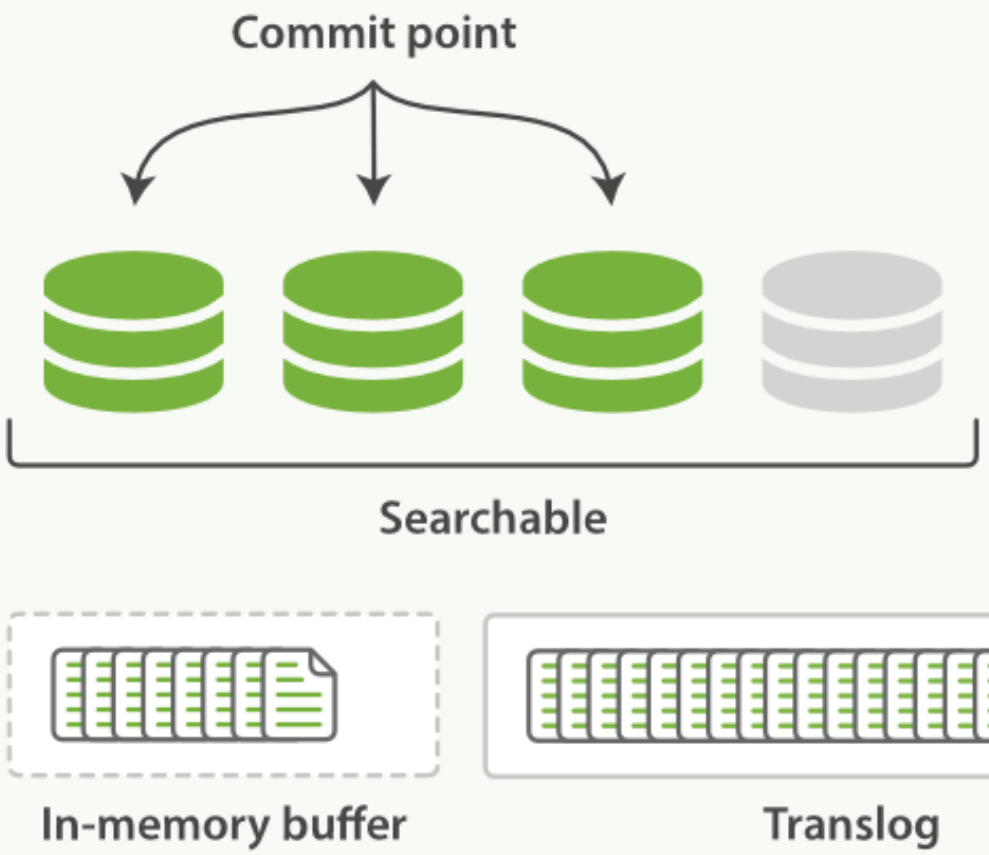
Figure 22. After a refresh, the buffer is cleared but the transaction log is not



- 在in-memory buffer中的文档会被写入到一个新的segment，但没有fsync。
- in-memory buffer被清空

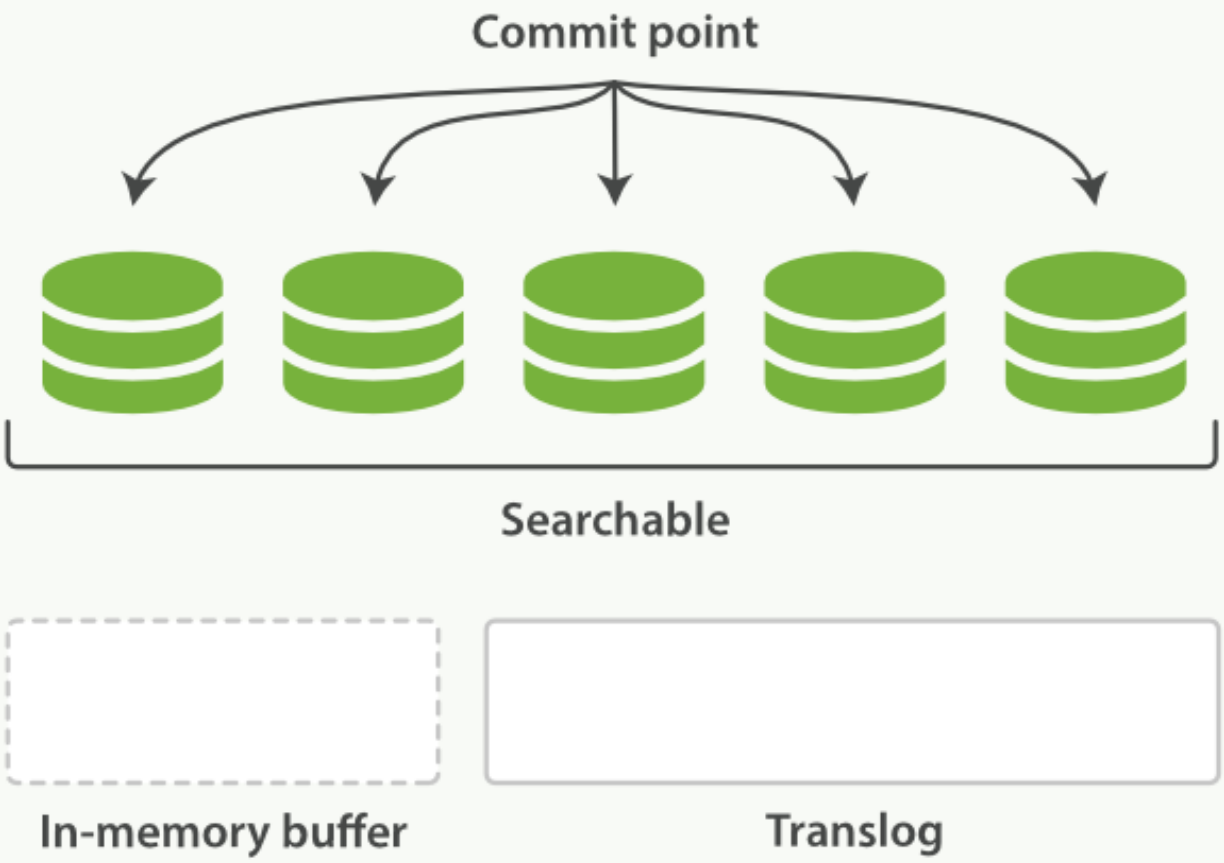
3.这个过程将会持续进行：新的文档将被添加到in-memory buffer和translog日志中，见Figure23

Figure 23. The transaction log keeps accumulating documents



4.一段时间后，当translog变得非常大时，索引将会被flush，新的translog将会建立，一个完全的提交进行完毕。见Figure24

Figure 24. After a flush, the segments are fully commited and the transaction log is cleared



- 在in-memory中的所有文档将被写入到新的segment
- 内核文件系统会被fsync到磁盘。
- 旧的translog日志被删除

translog日志提供了一个所有还未被flush到磁盘的操作的持久化记录。当ES启动的时候，它会使用最新的commit point从磁盘恢复所有已有的segments，然后将重现所有在translog里面的操作来添加更新，这些更新发生在最新的一次commit的记录之后还未被fsync。

translog日志也可以用来提供实时的CRUD。当你试图通过文档ID来读取、更新、删除一个文档时，它会首先检查translog日志看看有没有最新的更新，然后再从响应的segment中获得文档。这意味着它每次都会对最新版本的文档做操作，并且是实时的。

2.6.5 Segment合并

通过每隔一秒的自动刷新机制会创建一个新的segment，用不了多久就会有很多的segment。segment会消耗系统的文件句柄，内存，CPU时钟。最重要的是，每一次请求都会依次检查所有的segment。segment越多，检索就会越

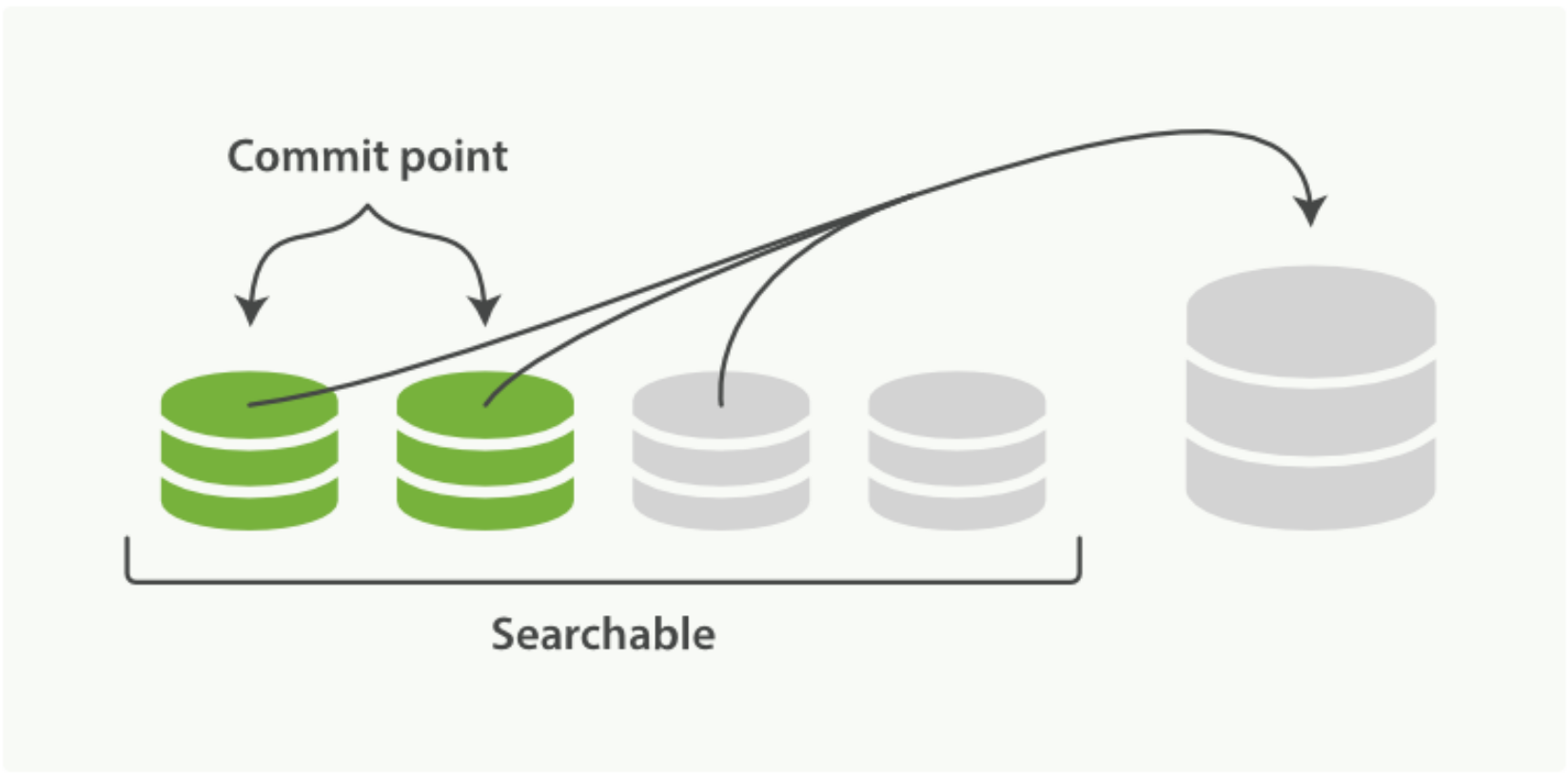
慢。

ES通过在后台merge这些segment的方式解决这个问题。小的segment merge到大的，大的merge到更大的。。。

这个过程也是那些被”删除”的文档真正被清除出文件系统的过程，因为被标记为删除的文档不会被拷贝到大的segment中。

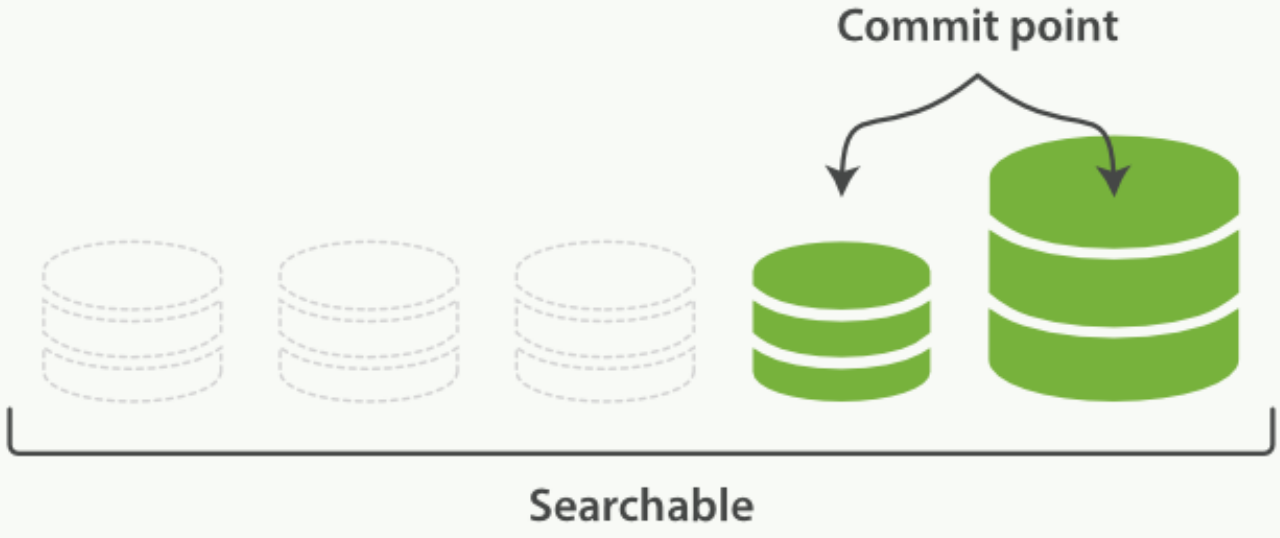
合并过程如Figure25：

Figure 25. Two commited segments and one uncommited segment in the process of being merged into a bigger segment



1.当在建立索引过程中，refresh进程会创建新的segments然后打开他们以供索引。
2.merge进程会选择一些小的segments然后merge到一个大的segment中。这个过程不会打断检索和创建索引。

Figure 26. Once merging has finished, the old segments are deleted



3.Figure26，一旦merge完成，旧的segments将被删除

- 新的segment被flush到磁盘
- 一个新的提交点被写入，包括新的segment，排除旧的小的segments
- 新的segment打开以供索引
- 旧的segments被删除

merge大的segments会消耗大量的I/O和CPU，严重影响索引性能。默认，ES会节制merge过程来给留下足够多的系统资源。

近实时搜索，段数据刷新，数据可见性更新和事务日志

理想的搜索解决方案是这样的：新的数据一添加到索引中立马就能搜索到。第一眼看上去，这不正是ElasticSearch的工作方式吗，即使是多服务器环境也是如此。但是真实情况不是这样的(至少现在不是)，后面会讲到为什么它是似是而非。首先，我们往新创建的索引中添加一个新的文档，命令如下：

```
curl -XPOST localhost:9200/test/test/1 -d '{ "title": "test" }'
```

接下来，我们在替换文档的同时查找该文档。我们用如下的链式命令来实现这一点：

```
curl -XPOST localhost:9200/test/test/1 -d '{ "title": "test2" }' ; curl
localhost:9200/test/test/_search?pretty
```

上面命令的结果类似如下：

```
{"ok":true,"_index":"test","_type":"test","_id":"1","_version":2}
{
    "took" : 1,
    "timed_out" : false,
        "_shards" : {
        "total" : 5,
        "successful" : 5,
        "failed" : 0
    },
    "hits" : {
        "total" : 1,
        "max_score" : 1.0,
        "hits" : [ {
            "_index" : "test",
            "_type" : "test",
            "_id" : "1",
            "_score" : 1.0, "_source" : { "title": "test" }
        } ]
    }
}
```

第一行是第一个命令，即索引命令的返回结果。可以看到，数据更新成功。因此，第二个命令，即查询命令查询到的文档title域值应该为test2。但是，可以看到结果并不如人所愿。这背后发生了什么呢?

在揭开前一个问题的答案之前，我们先退一步，来了解底层的Apache Lucene工具包是如何让新添加的文档对搜索可见的。

更新索引并且将改动提交

从 第1章 介绍ElasticSearch 的 介绍Apache Lucene一节中，我们已经了解到，在索引过程中，新添加的文档都是写入到段(segments)中。每个段都是有着独立的索引结构，这意味着查询与索引两个过程是可以并行存在的，索引过程中，系统会不定期创建新的段。Apache Lucene通过在索引目录中创建新的segments_N文件来标识新的段。段创建的过程就称为索引的提交。

Lucene可以一种安全的方式实现索引的提交——我们可以确定段文件要么全部创建成功，要么失败。如果错误发生，我们可以确保索引状态的一致性。

回到我们的例子中，第一条命令添加文档到索引中，但是没有提交。这就是它的工作方式。然而，索引数据的提交也不能保证数据是搜索可见的。Lucene工具包使用一个名为Searcher的抽象类来读取索引。索引提交操作完成后，Searcher对象需要重新打开才能加载到新创建的索引段。这整个过程称为更新。出于性能的考虑，ElasticSearch会将推迟开销巨大的更新操作，默认情况下，单个文档的添加并不会触发搜索器的更新，Searcher对象会每秒更新一次。这个频率已经比较高了，但是在一些应用程序中，需要更频繁的更新。对面这个需求，我们可以考虑使用其它的解决方案或者再次核实我们是否真的需要这样做。如果确实需要，那么可以使用ElasticSearch API强制更新。比如，上面的例子中，我们可以执行如下的命令强制更新：

```
curl –XGET localhost:9200/test/_refresh
```

如果在搜索前执行了上面的命令，那么ElasticSearch就可以搜索到修改后的文档。

修改Searcher对象默认的更新时间

Searcher对象的默认更新时间可以通过使用`index.refresh_interval`参数来修改，该参数无论是添加到ElasticSearch的配置文件中或者使用update settings API都可以生效。例如：

```
curl -XPUT localhost:9200/test/_settings -d '{
    "index" : {
        "refresh_interval" : "5m"
    }
}'
```

上面的命令将使Searcher每5秒钟自动更新一次。请记住在更新两个时间点之间添加到索引的数据对查询是不可见的。

们已经提到，更新操作的时间开销和内存开销都很大。更新的时间段设置越长，索引速度越快。如果在整个索引过程中数据都不必对索可见，那么可以考虑关闭更新操作来换取高效的索引过程，设置index.refresh_interval 参数值为-1即可，记得在索引完成后改回来的值。

事务日志的配置

l果事务日志的默认配置无法满足业务需求，ElasticSearch允许用户在事务日志的处理上自己配置参数。如下l参数可以控制系统的事务日志行为，参数可以设置在elasticsearch.yml文件中，也可以用索引设置更新API:设置：

- `index.translog.flush_threshold_period`:默认值为30秒(30m)。该属性用于控制自动刷新的时间，即使期间没有数据写入，也会强制刷新。
- `index.translog.flush_threshold_ops`:用来指定刷新操作执行的最多事务次数。默认值是5000。
- `index.translog.flush_threshold_size`:用来指定事务日志的最大容量。如果事务日志的大小等于或者超过参数值，就会执行刷新操作。默认值是200M.
- `index.translog.disable_flush`:该属性用来关闭自动刷新。默认情况下日志的自动刷新是开启的，但是有时需要暂时关闭日志的自动刷新。比如需要添加大量数据到集群中时，关闭日志的自动刷新有助于系统性能的提升。

面的所有参数都定义于某个索引，但是作用于索引的每个分片上。

然，除了预先在elasticsearch.yml文件中设置这些参数外，它们还可以用Settings Update API来设置,比l：

```
curl -XPUT localhost:9200/test/_settings -d '{
    "index" : {
        "translog.disable_flush" : true
    }
}'
```

.面的命令一般用于导入大量数据到索引前执行，这样会提升索引阶段的性能。但是要记住，数据索引完毕后·开启事务日志的自动刷新。

近实时GET

·务日志带来的副产品就是实时GET操作，即提供了返回早期版本文档，包括未提交文档的功能。实时GET操·从索引中获取数据，但事先会从事务日志中查看是否有更新的版本。如果存在没有刷新的文档，索引中的数i就会被忽略，返回更新版本的文档——事务日志中的文档。想弄清楚该如何使用该功能，可以用户如下的命·代替搜索操作：

```
curl -XGET localhost:9200/test/test/1?pretty
```

asticSearch返回的结果如下：

```
{
    "_index" : "test",
    "_type" : "test",
    "_id" : "1",
    "_version" : 2,
    "exists" : true, "_source" : { "title": "test2" }
}
```

l果看到结果，你肯定会以为出现了幻觉会再多看一眼，因为返回的文档是最新的，没有用到前面提到的更新earcher的小窍门也得到想要的结果。

总结

本篇从索引的创建操作和原理等方面介绍了ES索引的一些内容，很多都来自各位大神的总结。经过使用ES越来越多开始作为数据库的辅助。希望大家多多指点。