分片内部原理

在 [distributed-cluster]，我 介 了 分片，并将它 描述成最小的 工作 元。但是究竟什 是一个分片，它是如何工作的？在 个章 ，我 回答以下 ：

- 什 搜索是 近 的？
- 什 文 的 CRUD (建- 取-更新- 除) 操作是 的?
- Elasticsearch 是 保 更新被持久化在断 也不 失数据?
- 什 除文 不会立刻 放空 ？
- `refresh`, `flush`, 和 `optimize` API 都做了什 ， 什 情况下 是用他 ？

最 的理解一个分片如何工作的方式是上一堂 史 。 我 将要 提供一个 近 搜索和分析的分布式持久化数据存 需要解决的 。

> **内容警告**
>
> 本章展示的 些信息 供 趣 。 了使用 Elasticsearch 并不需要理解和 所有的 。 个章 是 了了解工作机制，并且 了将来 需要 些信息 ，知道 些信息在 里。但是不要被 些 所累。

使文本可被搜索

必 解决的第一个挑 是如何使文本可被搜索。 的数据 个字段存 个 ，但 全文 索并不 。文本字段中的 个 需要被搜索， 数据 意味着需要 个字段有索引多 (里指)的能力。

最好的支持 一个字段多个 需求的数据 是我 在 [inverted-index] 章 中介 的 倒排索引 。倒排索引包含一个有序列表，列表包含所有文 出 的不重 个体，或称 ， 于 一个 ，包含了它所有曾出 文 的列表。

```
Term  | Doc 1 | Doc 2 | Doc 3 | ...
-----------------------------------
brown |   X   |       |  X    | ...
fox   |   X   |   X   |  X    | ...
quick |   X   |   X   |       | ...
the   |   X   |       |  X    | ...
```

NOTE

当 倒排索引 ，我 会 到 文 引，因 史原因，倒排索引被用来 整个非 化文本文 行 引。 Elasticsearch 中的 文 是有字段和 的 化 JSON 文 。事 上，在 JSON 文 中， 个被索引的字段都有自己的倒排索引。

个倒排索引相比特定 出 的文 列表，会包含更多其它信息。它会保存 一个 出 的文 数， 在 的文 中一个具体 出 的 次数， 在文 中的 序， 个文 的 度，所有文 的平均 度，等等。 些 信息允 Elasticsearch 决定 些 比其它 更重要， 些文 比其它文 更重要， 些内容在 [relevance-intro] 中有描述。

了能 期功能，倒排索引需要知道集合中的 所有 文 ， 是需要 到的 。

早期的全文 索会 整个文 集合建立一个很大的倒排索引并将其写入到磁 。 一旦新的索引就 ，旧的就会被其替 ， 最近的 化便可以被 索到。

不 性

倒排索引被写入磁 后是 不可改 的:它永 不会修改。 不 性有重要的 :

- 不需要 。如果 从来不更新索引， 就不需要担心多 程同 修改数据的 。
- 一旦索引被 入内核的文件系 存，便会留在 里，由于其不 性。只要文件系 存中 有足 的空 ，那 大部分 求会直接 求内存，而不会命中磁 。 提供了很大的性能提升。
- 其它 存(像filter 存)，在索引的生命周期内始 有效。它 不需要在 次数据改 被重建，因 数据不会 化。
- 写入 个大的倒排索引允 数据被 ， 少磁 I/O 和 需要被 存到内存的索引的使用量。

当然，一个不 的索引也有不好的地方。主要事 是它是不可 的! 不能修改它。如果 需要 一个新的文 可被搜索， 需要重建整个索引。 要 一个索引所能包含的数据量造成了很大的限制，要 索引可被更新的 率造成了很大的限制。

更新索引

下一个需要被解决的 是 在保留不 性的前提下 倒排索引的更新？答案是: 用更多的索引。

通 加新的 充索引来反映新近的修改，而不是直接重写整个倒排索引。 一个倒排索引都会被 流到—从最早的 始— 完后再 果 行合并。

Elasticsearch 基于 Lucene, 个 java 引入了 按段搜索 的概念。 一 段 本身都是一个倒排索引， 但 索引 在 Lucene 中除表示所有 段 的集合外， 加了 提交点 的概念 — 一个列出了所有已知段的文件，就像在 一个 Lucene 索引包含一个提交点和三个段 中描 的那 。如 一个在内存 存中包含新文 的 Lucene 索引 所示，新的文 首先被添加到内存索引 存中，然后写入到一个基于磁 的段，如 在一次提交后，一个新的段被添加到提交点而且 存被清空。 所示。

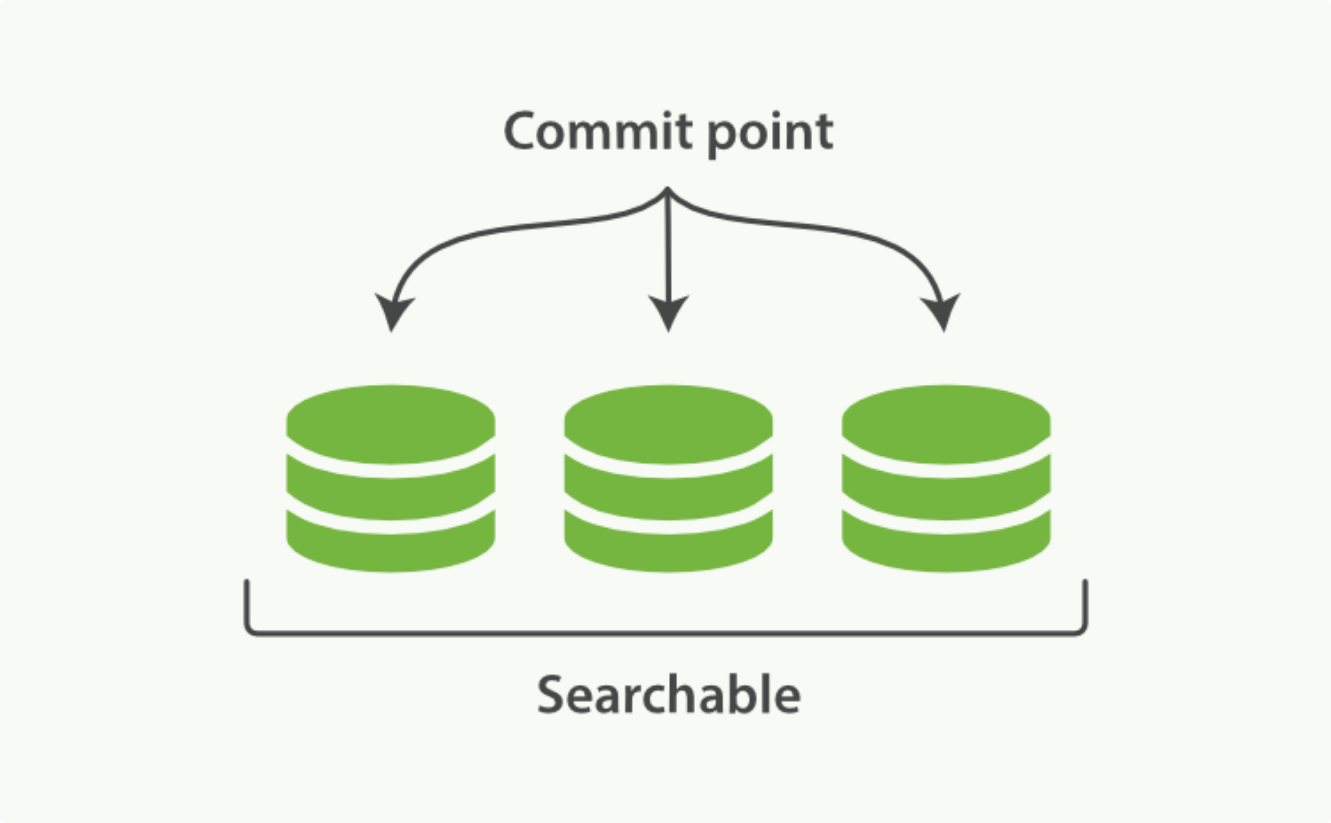

Figure 1. 一个 *Lucene* 索引包含一个提交点和三个段

索引与分片的比

被混 的概念是，一个 *Lucene* 索引 我 在 Elasticsearch 称作 分片 。 一个 Elasticsearch 索引是分片的集合。 当 Elasticsearch 在索引中搜索的 候， 他 送 到一个属于索引的分片(Lucene 索引)，然后像 [distributed-search] 提到的那 ，合并 个分片的 果到一个全局的 果集。

逐段搜索会以如下流程 行工作：

1. 新文 被收集到内存索引 存， 一个在内存 存中包含新文 的 Lucene 索引 。
2. 不 地， 存被 提交 ：
   - 一个新的段—一个追加的倒排索引—被写入磁 。
   - 一个新的包含新段名字的 提交点 被写入磁 。
   - 磁 行 同 — 所有在文件系 存中等待的写入都刷新到磁 ，以 保它被写入物理文件。
3. 新的段被 ， 它包含的文 可 以被搜索。
4. 内存 存被清空，等待接收新的文 。

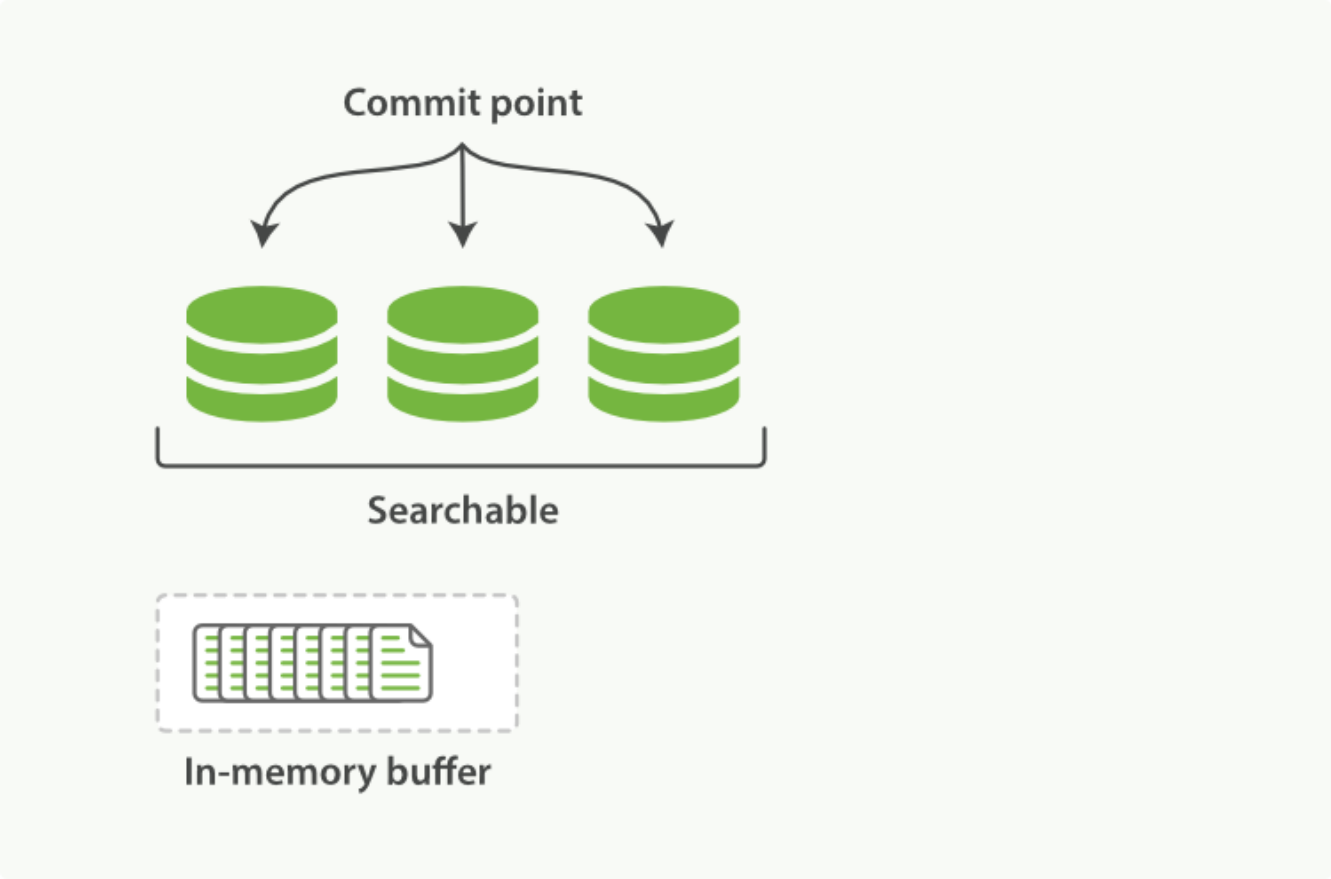

Figure 2. 一个在内存　存中包含新文　的 *Lucene* 索引

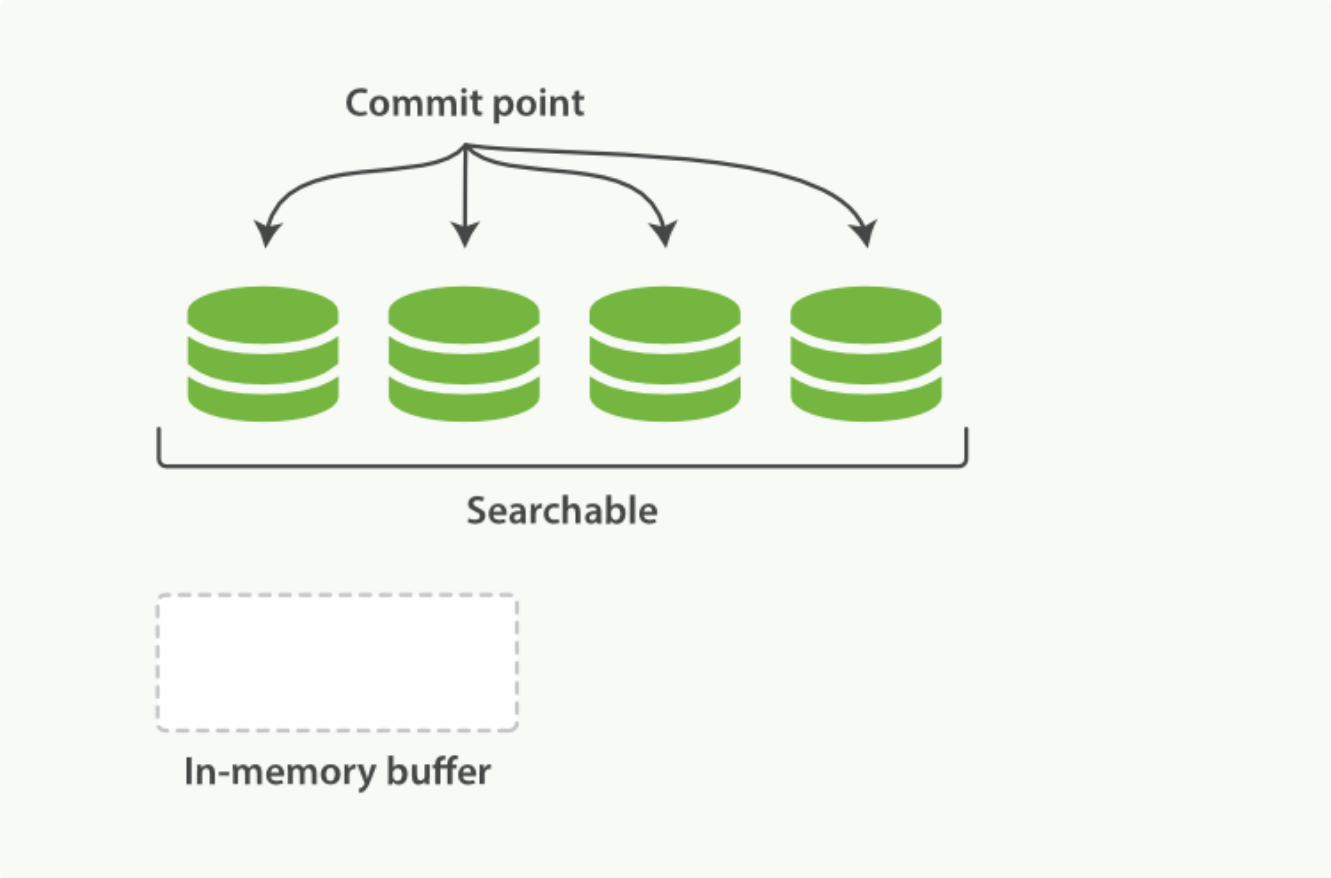

Figure 3. 在一次提交后，一个新的段被添加到提交点而且　存被清空。

当一个　　被触　，所有已知的段按　序被　　。　　会　所有段的　果　行聚合，以保　　个　和个文　的　　都被准　　算。　　方式可以用相　　低的成本将新文　添加到索引。

除和更新

段是不可改 的，所以既不能从把文 从旧的段中移除，也不能修改旧的段来 行反映文 的更新。取而代之的是， 个提交点会包含一个 .del 文件，文件中会列出 些被 除文 的段信息。

当一个文 被“ 除” ，它 上只是在 .del 文件中被 除。一个被 除的文 然可以被 匹配到， 但它会在最 果被返回前从 果集中移除。

文 更新也是 似的操作方式：当一个文 被更新 ，旧版本文 被 除，文 的新版本被索引到一个新的段中。 可能 个版本的文 都会被一个 匹配到，但被 除的那个旧版本文 在 果集返回前就已被移除。

在 段合并 , 我 展示了一个被 除的文 是 被文件系 移除的。

近 搜索

随着按段（per-segment）搜索的 展，一个新的文 从索引到可被搜索的延 著降低了。新文 在几分 之内即可被 索，但 是不 快。

磁 在 里成 了瓶 。提交（Commiting）一个新的段到磁 需要一个 fsync 来保段被物理性地写入磁 ， 在断 的 候就不会 失数据。 但是 fsync 操作代 很大; 如果 次索引一个文 都去 行一次的 会造成很大的性能 。

我 需要的是一个更 量的方式来使一个文 可被搜索， 意味着 fsync 要从整个 程中被移除。

在Elasticsearch和磁 之 是文件系 存。 像之前描述的一 ， 在内存索引 冲区（ 在内存冲区中包含了新文 的 Lucene 索引 ）中的文 会被写入到一个新的段中（ 冲区的内容已被写入一个可被搜索的段中，但 没有 行提交 ）。 但是 里新段会被先写入到文件系 存— 一 代 会比 低， 后再被刷新到磁 — 一 代 比 高。不 只要文件已 在 存中，就可以像其它文件一 被打 和 取了。

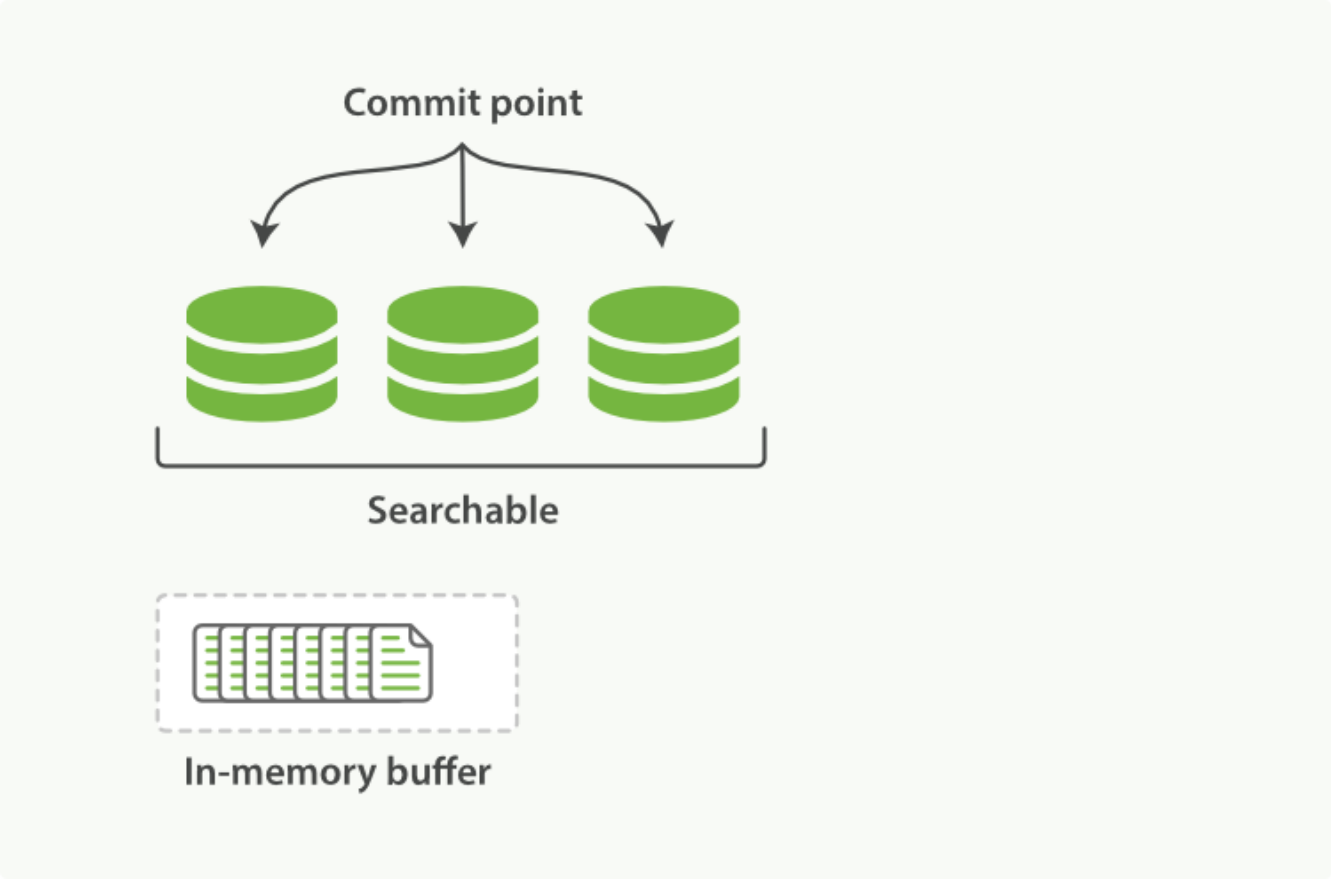

Figure 4. 在内存 冲区中包含了新文 的 *Lucene* 索引

Lucene 允 新段被写入和打 —使其包含的文 在未 行一次完整提交 便 搜索可 。 方式比 行一次提交代 要小得多，并且在不影 性能的前提下可以被 繁地 行。

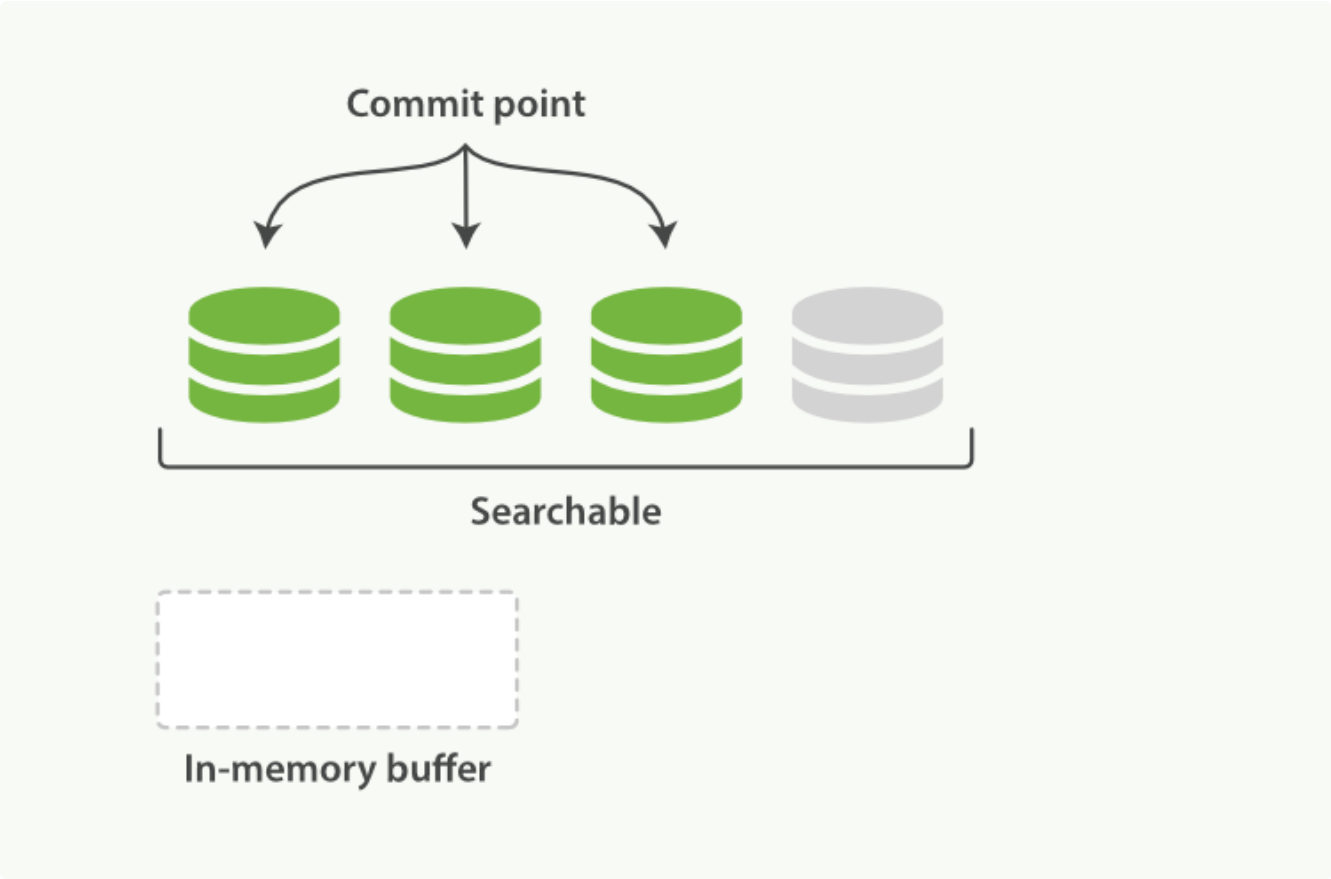

Figure 5. 冲区的内容已 被写入一个可被搜索的段中，但 没有 行提交

refresh API

在 Elasticsearch 中，写入和打 一个新段的 量的 程叫做 *refresh* 。 情况下 个分片会 秒自 刷新一次。 就是 什 我 Elasticsearch 是 近 搜索: 文 的 化并不是立即 搜索可 ，但会在一秒之内 可 。

些行 可能会 新用 造成困惑: 他 索引了一个文 然后 搜索它，但却没有搜到。 个 的解决 法是用 `refresh` API 行一次手 刷新:

```
POST /_refresh ①
POST /blogs/_refresh ②
```

① 刷新（Refresh）所有的索引。

② 只刷新（Refresh） `blogs` 索引。

> **TIP**
>
> 尽管刷新是比提交 量很多的操作，它 是会有性能 。当写 的 候， 手 刷新很有用，但是不要在生 境下 次索引一个文 都去手 刷新。 相反， 的用需要意 到 Elasticsearch 的近 的性 ，并接受它的不足。

并不是所有的情况都需要 秒刷新。可能 正在使用 Elasticsearch 索引大量的日志文件， 可能想 化索引速度而不是近 搜索， 可以通 置 `refresh_interval` ， 降低 个索引的刷新 率：

```
PUT /my_logs
{
  "settings": {
    "refresh_interval": "30s" ①
  }
}
```

① 30秒刷新 `my_logs` 索引。

`refresh_interval` 可以在既存索引上 行 更新。 在生 境中，当 正在建立一个大的新索引 ，可以先 自 刷新，待 始使用 索引 ，再把它 回来：

```
PUT /my_logs/_settings
{ "refresh_interval": -1 } ①

PUT /my_logs/_settings
{ "refresh_interval": "1s" } ②
```

① 自 刷新。

② 秒自 刷新。

> **CAUTION**
>
> `refresh_interval` 需要一个 持 ， 例如 `1s` （1 秒） 或 `2m` （2 分 ）。一个 *1* 表示的是 *1* 秒 --无疑会使 的集群陷入 。

持久化　更

如果没有用　　fsync　　把数据从文件系　　存刷（flush）到硬　，我　不能保　数据在断　甚至是程序正常退出之后依然存在。　了保　　Elasticsearch　　的可　性，需要　保数据化被持久化到磁　。

在　　更新索引，我　　一次完整的提交会将段刷到磁　，并写入一个包含所有段列表的提交点。Elasticsearch　　在　　或重新打　一个索引的　程中使用　个提交点来判断　些段隶属于当前分片。

即使通　　秒刷新（refresh）　　了近　　搜索，我　　然需要　常　行完整提交来　保能从失　中恢　。但在　次提交之　　生　化的文　　？我　也不希望　失掉　些数据。

Elasticsearch　　加了一个　*translog*　，或者叫事　日志，在　一次　　Elasticsearch　　行操作　均　行了日志　　。通　translog，整个流程看起来是下面　　：

1. 一个文　被索引之后，就会被添加到内存　冲区，并且　　追加到了　　translog　　，正如　　新的文　被添加到内存　冲区并且被追加到了事　日志 描述的一　。

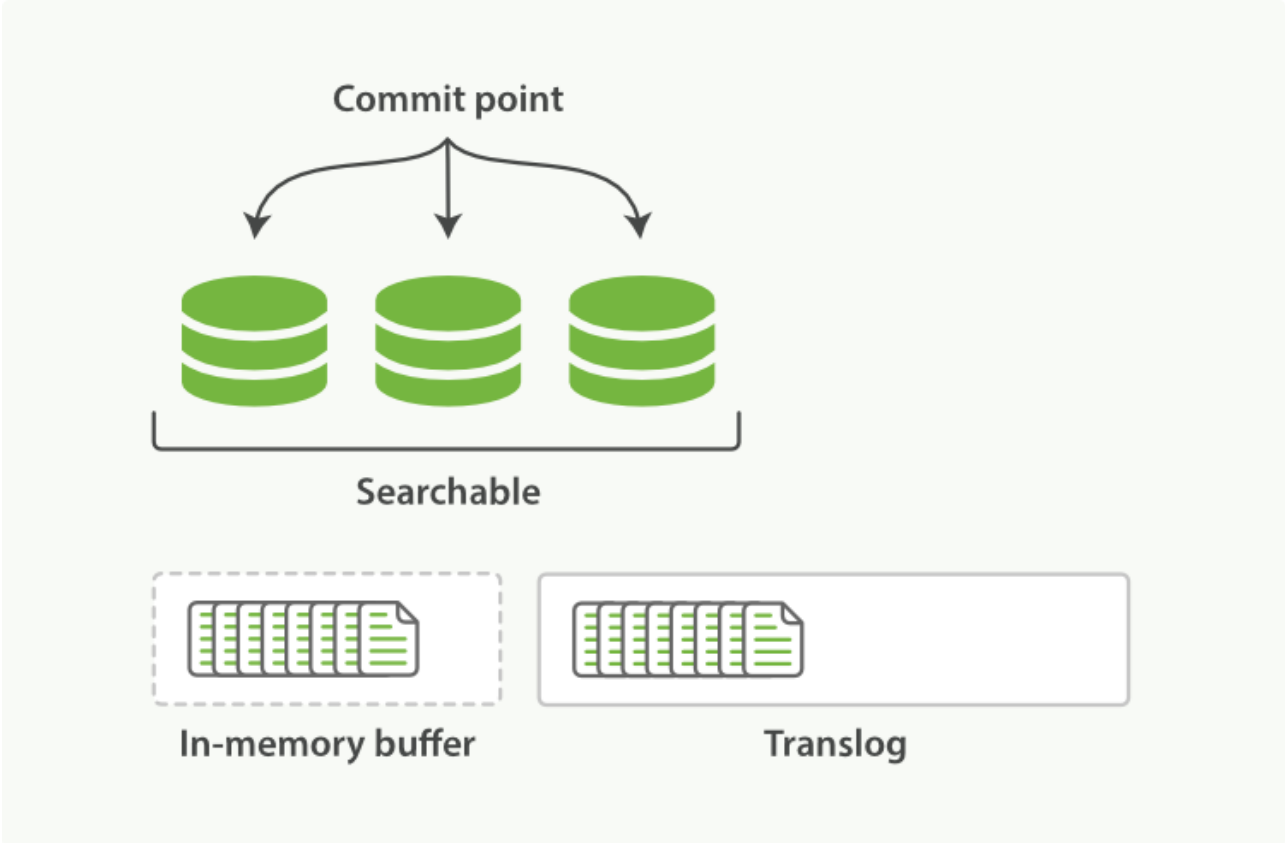

Figure 6. 新的文　被添加到内存　冲区并且被追加到了事　日志

2. 刷新（refresh）使分片　于　　刷新（refresh）完成后，　　存被清空但是事　日志不会　　描述的状　，分片　秒被刷新（refresh）一次：

    - 　些在内存　冲区的文　被写入到一个新的段中，且没有　行 fsync 操作。
    - 　个段被打　，使其可被搜索。
    - 内存　冲区被清空。

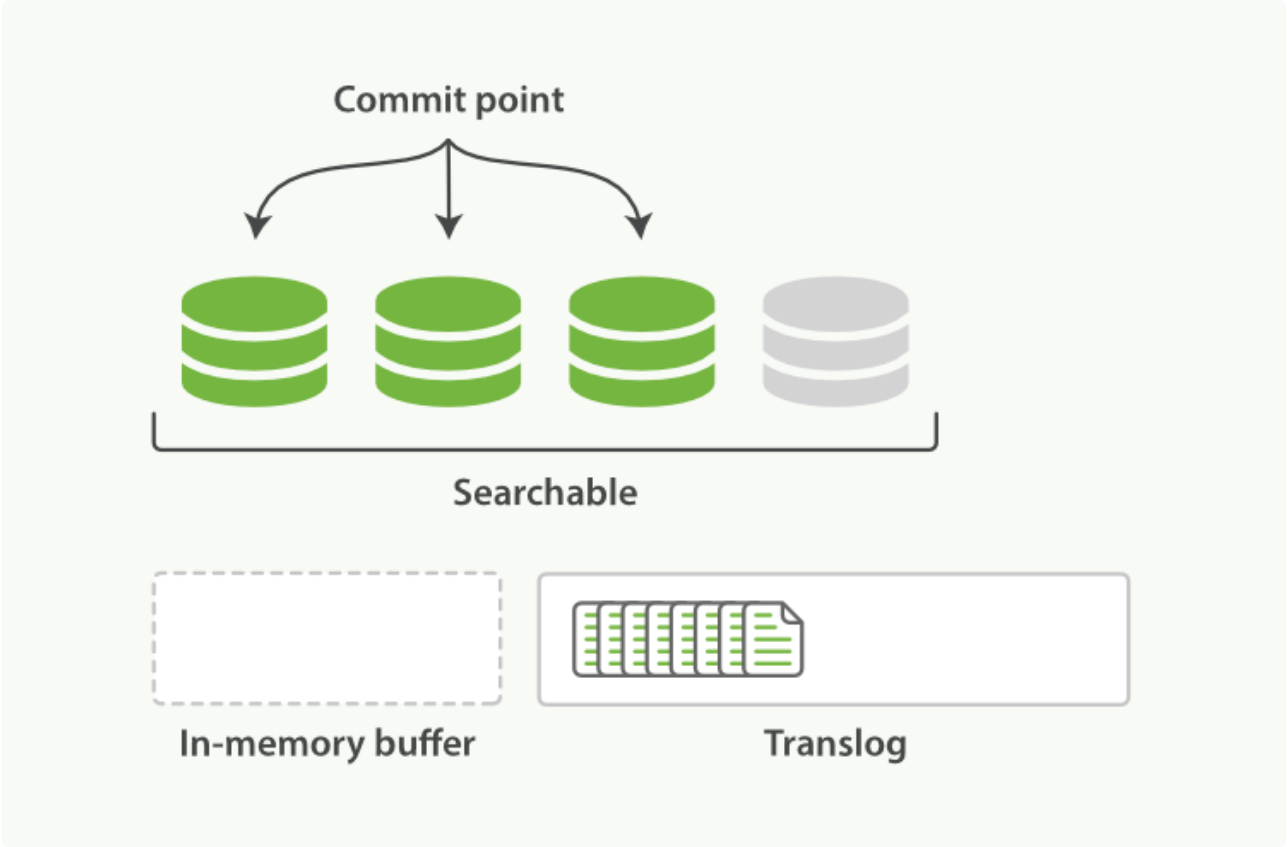

Figure 7. 刷新（*refresh*）完成后，　存被清空但是事　日志不会

3. 个　程　　工作，更多的文　被添加到内存　冲区和追加到事　日志（　事　日志不断　累文　）。

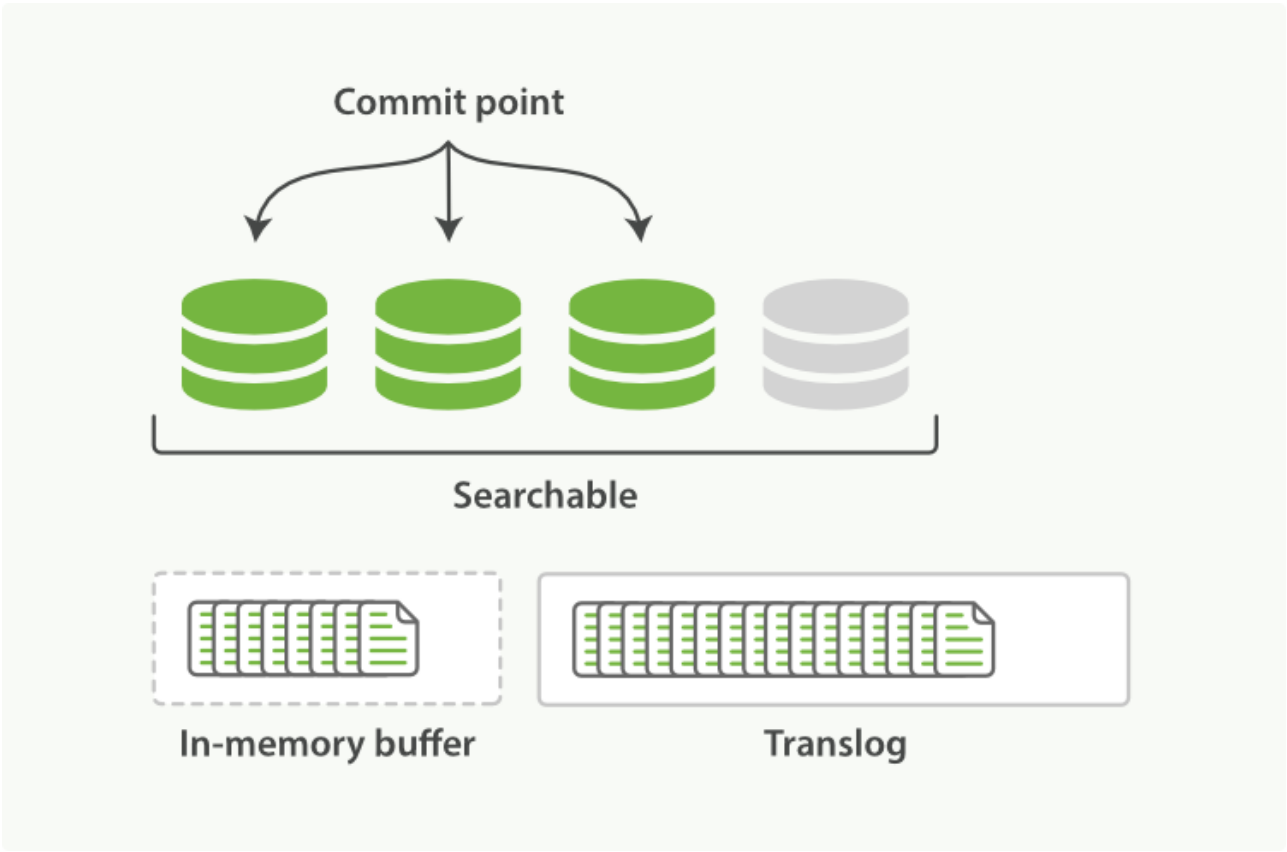

Figure 8. 事　日志不断　累文

4. 隔一段　　—例如　translog　　得越来越大—索引被刷新（flush）；一个新的　translog　被建，并且一个全量提交被　行（　在刷新（flush）之后，段被全量提交，并且事　日志被清空）：

- 所有在内存 冲区的文 都被写入一个新的段。
- 冲区被清空。
- 一个提交点被写入硬 。
- 文件系 存通 `fsync` 被刷新（flush）。
- 老的 translog 被 除。

translog 提供所有 没有被刷到磁 的操作的一个持久化 。当 Elasticsearch 的 候， 它会从磁 中使用最后一个提交点去恢 已知的段，并且会重放 translog 中所有在最后一次提交后 生的 更操作。

translog 也被用来提供 CRUD 。当 着通 ID 、更新、 除一个文 ，它会在 从相的段中 索之前， 首先 translog 任何最近的 更。 意味着它 是能 地 取到文 的最新版本。

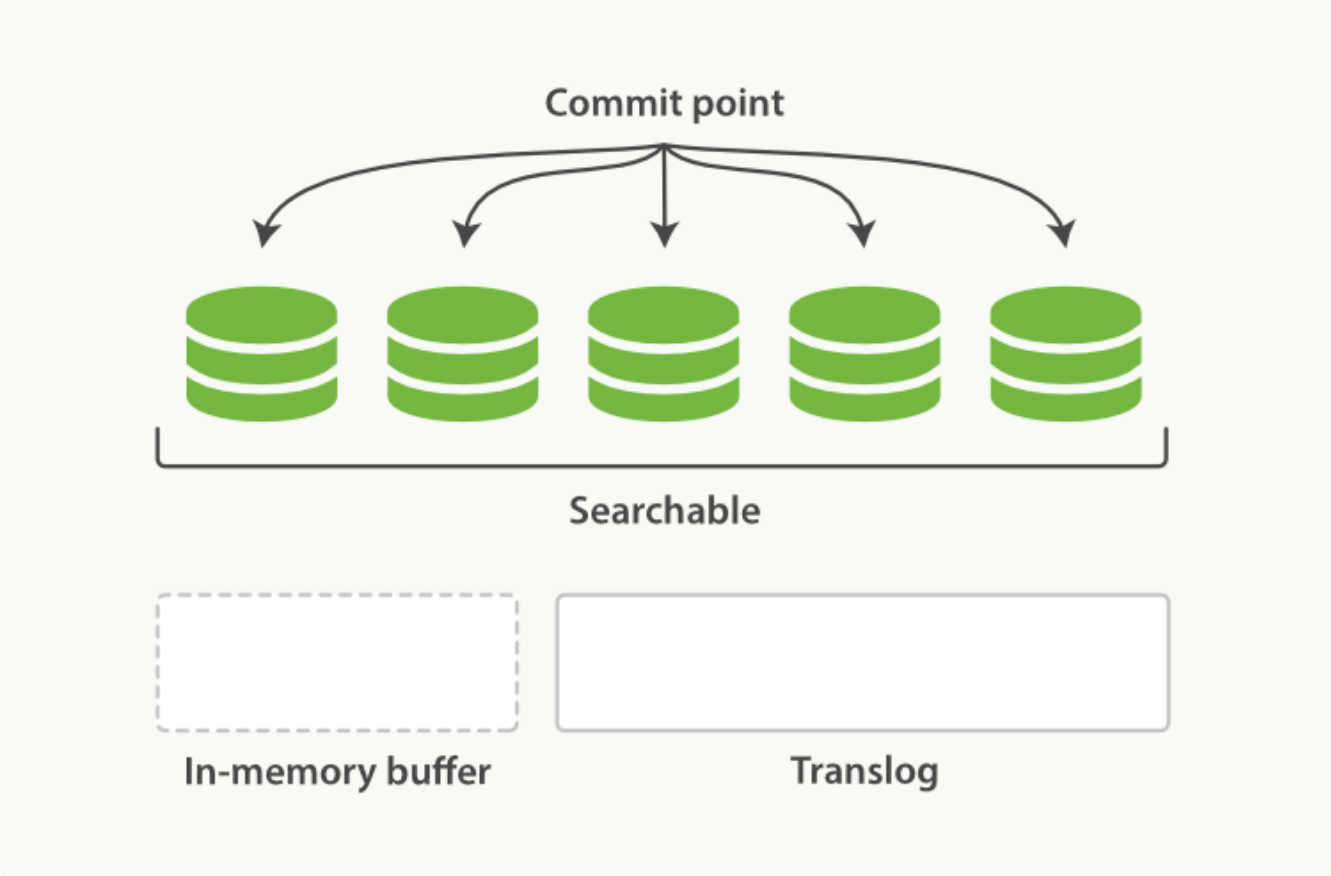

Figure 9. 在刷新（*flush*）之后，段被全量提交，并且事 日志被清空

flush API

个 行一个提交并且截断 translog 的行 在 Elasticsearch 被称作一次 *flush* 。 分片 30分 被自 刷新（flush），或者在 translog 太大的 候也会刷新。 看 {ref}/index-modules-translog.html#_translog_settings[`translog` 文] 来 置，它可以用来 控制 些 ：

{ref}/indices-flush.html[`flush` API] 可以被用来 行一个手工的刷新（flush）：

```
POST /blogs/_flush ①

POST /_flush?wait_for_ongoing ②
```

① 刷新（flush） blogs 索引。

② 刷新（flush）所有的索引并且并且等待所有刷新在返回前完成。

很少需要自己手　　行一个的 flush 操作；通常情况下，自　刷新就足　了。

就是　，在重　　点或　　索引之前　行 flush　有益于　的索引。当 Elasticsearch　　恢　或重新打一个索引，它需要重放 translog 中所有的操作，所以如果日志越短，恢　越快。

Translog 有多安全?

translog 的目的是保　操作不会　失。　引出了　个　　：Translog 有多安全？

在文件被 fsync 到磁　前，被写入的文件在重　之后就会　失。　　translog 是　5 秒被 fsync 刷新到硬　，　或者在　次写　求完成之后　行(e.g. index, delete, update, bulk)。　个　程在主分片和　制分片都会　生。最　，　基本上，　意味着在整个　求被　fsync　到主分片和　制分片的translog之前，　的客　端不会得到一个 200 OK　　。

在　次　求后都　行一个　　fsync　　会　来一些性能　失，尽管　践表明　　失相　　小（特　是bulk　入，它在一次　求中平　了大量文　的　　）。

但是　于一些大容量的偶　　失几秒数据　　也并不　重的集群，使用　　的　　fsync　　是比　有益的。比如，写入的数据被　存到内存中，再　5秒　行一次 fsync 。

　个行　可以通　　置 durability 参数　async 来　用：

```
PUT /my_index/_settings
{
    "index.translog.durability": "async",
    "index.translog.sync_interval": "5s"
}
```

　个　　可以　　索引　独　置，并且可以　　行修改。如果　决定使用　　translog　的　，需要　保　　在　生crash　，　失掉　sync_interval　　段的数据也无所　。　在决定前知　个特性。

如果　不　定　个行　的后果，最好是使用　　的参数（ "index.translog.durability": "request" ）来避免数据　失。

段合并

由于自　刷新流程　秒会　建一个新的段　　，　　会　致短　　内的段数量暴　。而段数目太多会　来　大的麻　。　　一个段都会消耗文件句柄、内存和cpu　行周期。更重要的是，　个搜索　求都必流　　个段；所以段越多，搜索也就越慢。

Elasticsearch通　在后台　行段合并来解决　个　　。小的段被合并到大的段，然后　些大的段再被合并到更大的段。

段合并的　候会将那些旧的已　除文　从文件系　中清除。被　除的文　（或被更新文　的旧版本）不会被拷　到新的大段中。

　　段合并不需要　做任何事。　行索引和搜索　会自　　行。　个流程像在
个提交了的段和一个未提交的段正在被合并到一个更大的段 中提到的一　工作：

1、 当索引的　候，刷新（refresh）操作会　建新的段并将段打　以供搜索使用。

2、　　合并　程　　一小部分大小相似的段，并且在后台将它　合并到更大的段中。
　并不会中断索引和搜索。

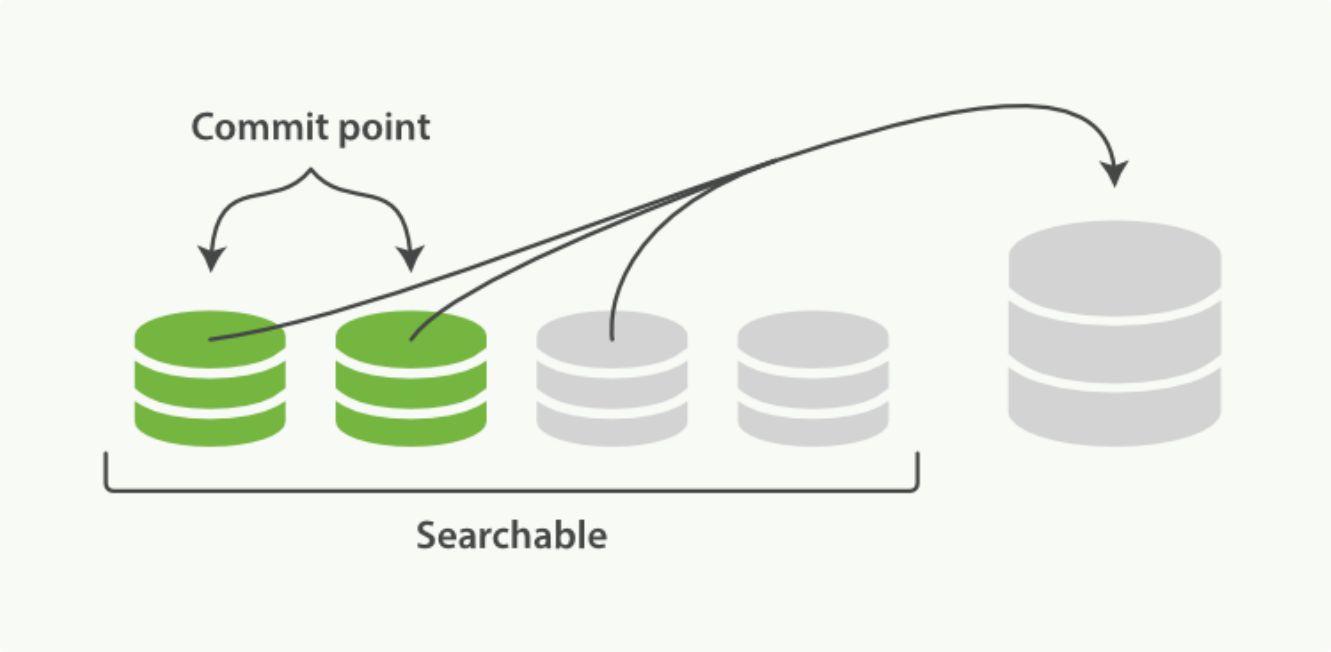

Figure 10.　个提交了的段和一个未提交的段正在被合并到一个更大的段

3、 一旦合并　束，老的段被　除　明合并完成　的活　：

- 新的段被刷新（flush）到了磁　。
- 写入一个包含新段且排除旧的和　小的段的新提交点。
- 新的段被打　用来搜索。
- 老的段被　除。

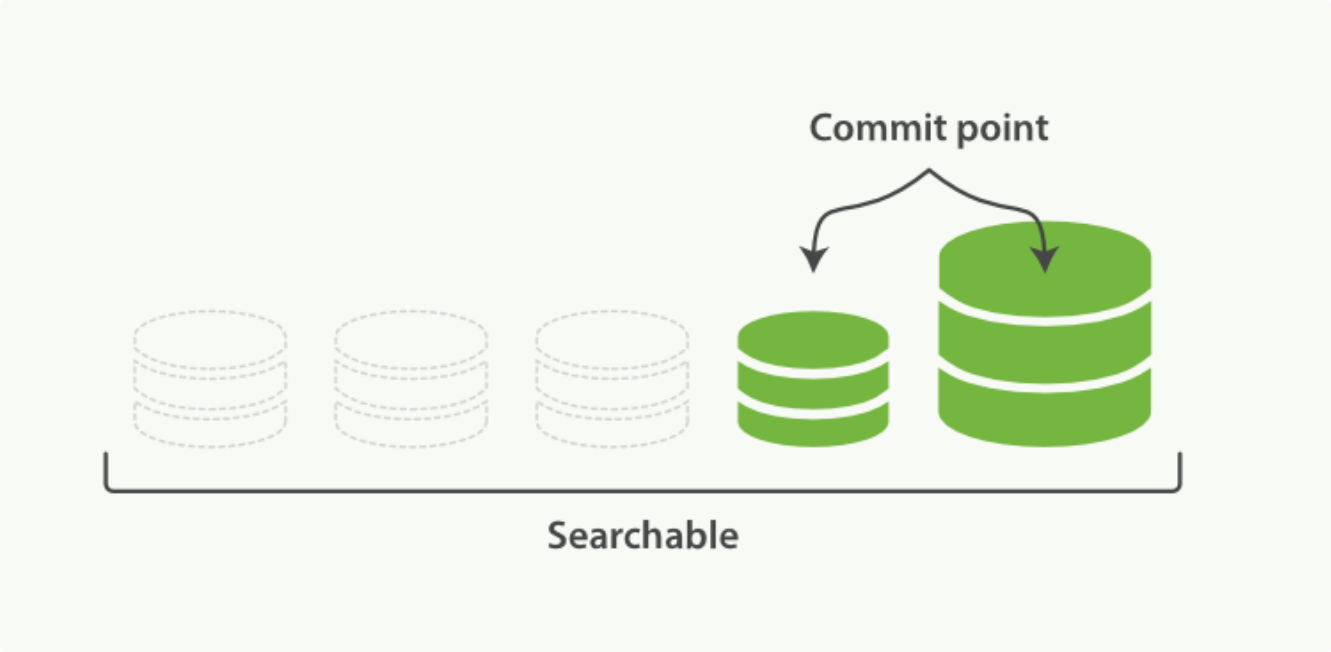

Figure 11. 一旦合并 束，老的段被 除

合并大的段需要消耗大量的I/O和CPU 源，如果任其 展会影 搜索性能。Elasticsearch在 情况下会 合并流程 行 源限制，所以搜索 然 有足 的 源很好地 行。

TIP 看 [segments-and-merging] 来 的 例 取 于合并 整的建 。

optimize API

`optimize` API大可看做是 制合并 API。它会将一个分片 制合并到 `max_num_segments` 参数指定大小的段数目。 做的意 是 少段的数量（通常 少到一个），来提升搜索性能。

WARNING `optimize` API 不 被用在一个 索引————一个正在被活 更新的索引。后台合并流程已 可以很好地完成工作。 optimizing 会阻碍 个程。不要干 它！

在特定情况下，使用 `optimize` API 有益 。例如在日志 用例下， 天、 周、 月的日志被存 在一个索引中。 老的索引 上是只 的；它 也并不太可能会 生 化。

在 情况下，使用optimize 化老的索引，将 一个分片合并 一个 独的段就很有用了； 既可以省 源，也可以使搜索更加快速：

```
POST /logstash-2014-10/_optimize?max_num_segments=1 ①
```

① 合并索引中的 个分片 一个 独的段

WARNING 注意，使用 `optimize` API 触 段合并的操作一点也不会受到任何 源上的限制。可能会消耗掉 点上全部的I/O 源, 使其没有余裕来 理搜索求，从而有可能使集群失去 。 如果 想要 索引 行 `optimize`，需要先使用分片分配（ 看 [migrate-indices]）把索引移到一个安全的 点，再行。