

Lab 12. Improving performance

This lab covers

- Bulk, multiget, and multisearch APIs
- Refresh, flush, merge, and store
- Filter caches and tuning filters
- Tuning scripts
- Query warmers
- Balancing JVM heap size and OS caches

Elasticsearch is commonly referred to as *fast* when it comes to indexing, searching, and extracting statistics through aggregations. *Fast* is a vague concept, making the “How fast?” question inevitable. As with everything, “how fast” depends on the particular use case, hardware, and configuration.

In this lab, our aim is to show you the best practices for configuring Elasticsearch so you can make it perform well for your use case. In every situation, you need to trade something for speed, so you need to pick your battles:

- *Application complexity*—In the first part of the lab, we’ll show how you can group multiple requests, such as `index`, `update`, `delete`, `get`, and `search`, in a single HTTP call. This grouping is something your application

needs to be aware of, but it can speed up your overall performance by a huge margin. Think 20 or 30 times better indexing because you'll have fewer network trips.

- *Indexing speed for search speed or the other way around*—In the second section of the lab, we'll take a deeper look at how Elasticsearch deals with Lucene segments: how refreshes, flushes, merge policies, and store settings work and how they influence index and search performance. Often, tuning for index performance has a negative impact on searches and vice versa.
- *Memory* —A big factor in Elasticsearch's speed is caching. Here's we'll dive into the details of the filter cache and how to use filters to make the best use of it. We'll also look at the shard query cache and how to leave enough room for the operating system to cache your indices, while still leaving enough heap size for Elasticsearch. If running a search on cold caches gets unacceptably slow, you'll be able to keep caches warm by running queries in the background with index warmers.
- *All of the above*—Depending on the use case, the way you analyze the text at index time and the kind of queries you use can be more complicated, slow down other operations, or use more memory. In the last part of the lab, we'll explore the typical tradeoffs you'll have while modeling your data and your queries: should you generate more terms when you index or look through more terms when you search? Should you take advantage of scripts or try to avoid them? How should you handle deep paging?

We'll discuss all these points and answer these questions in this lab. By the end, you'll have learned how to make Elasticsearch fast for your use case, and you'll get a deeper understanding of how it works along the way. Grouping multiple operations in a single HTTP request is often the easiest way to improve performance, and it gives the biggest performance gain. Let's start by looking at how you can do that through the bulk, multiget, and multisearch APIs.

10.1 *Grouping requests*

The single best thing you can do for faster indexing is to send multiple documents to be indexed at once via the bulk API. This will save network round-trips and allow for more indexing throughput. A single bulk can accept any indexing operation; for example, you can create documents or overwrite them. You can also add update or delete operations to a bulk; it's not only for indexing.

If your application needs to send multiple `get` or `search` operations at once, there are bulk equivalents for them, too: the multiget and multisearch APIs. We'll explore them later, but we'll start with the bulk API because in production it's “the way” to index for most use cases.

10.1.1 Bulk indexing, updating, and deleting

So far in this course you've indexed documents one at a time. This is fine for playing around, but it implies performance penalties from at least two directions:

- Your application has to wait for a reply from Elasticsearch before it can move on.
- Elasticsearch has to process all data from the request for every indexed document.

If you need more indexing speed, Elasticsearch offers a bulk API, which you can use to index multiple documents at once, as shown in figure 10.1.

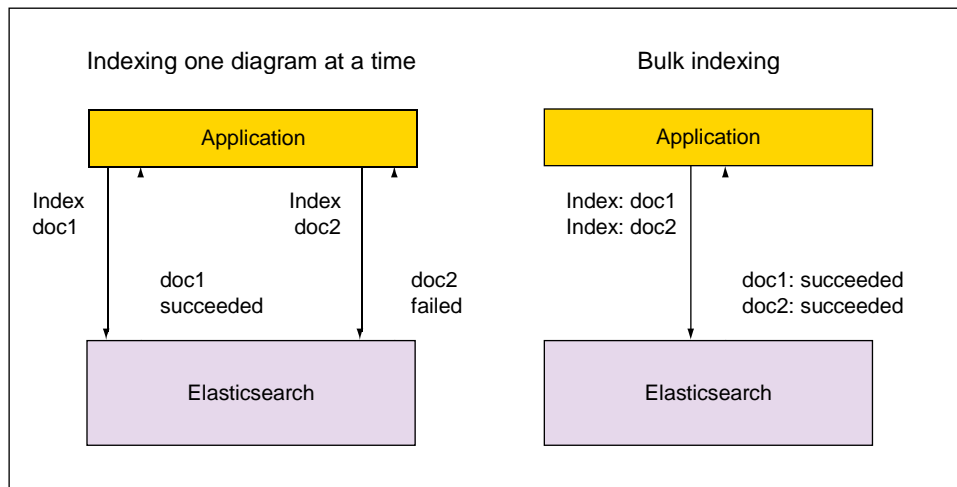


Figure 10.1 Bulk indexing allows you to send multiple documents in the same request.

As the figure illustrates, you can do that using HTTP, as you've used for indexing documents so far, and you'll get a reply containing the results of all the indexing requests.

INDEXING IN BULKS

In listing 10.1 you'll index a bulk of two documents. To do that, you have to do an HTTP POST to the `_bulk` endpoint, with data in a specific format. The format has the following requirements:

- Each indexing request is composed of two JSON documents separated by a new-line: one with the operation (`index` in your case) and metadata (like `index`, `type`, and `ID`) and one with the document contents.
- JSON documents should be one per line. This implies that each line needs to end with a newline (`\n`, or the ASCII 10 character), including the last line of the whole bulk of requests.

Listing 10.1 Indexing two documents in a single bulk

Every JSON needs to end in a newline (including the last one) and can't be pretty-printed.

```

REQUESTS_FILE=/tmp/test_bulk

echo '{"index":{"_index":"get-together", "_type":"group", "_id":"10"}}
{"name":"Elasticsearch Bucharest"}
{"index":{"_index":"get-together", "_type":"group", "_id":"11"}}
{"name":"Big Data Bucharest"}
' > $REQUESTS_FILE
curl -XPOST localhost:9200/_bulk --data-binary @$REQUESTS_FILE

```

Using a file and pointing to it via `--data-binary @file-name` to preserve newline characters

First line of the requests contains operation (index) and metadata (index,type,ID)

Document content

Using a file and pointing to it via `--data-binary @file-name` to preserve newline characters

For each of the two indexing requests, in the first line you add the operation type and some metadata. The main field name is the operation type: it indicates what Elasticsearch has to do with the data that follows. For now, you've used `index` for indexing, and this operation will overwrite documents with the same ID if they already exist. You can change that to `create`, to make sure documents don't get overwritten, or even `update` or `delete` multiple documents at once, as you'll see later.

`_index` and `_type` indicate where to index each document. You can put the index name or both the index and the type in the URL. This will make them the default index and type for every operation in the bulk. For example:

```
curl -XPOST localhost:9200/get-together/_bulk --data-binary @$REQUESTS_FILE
```

or

```
curl -XPOST localhost:9200/get-together/group/_bulk --data-binary
@$REQUESTS_FILE
```

You can then omit the `_index` and `_type` fields from the request itself. If you specify them, `index` and `type` values from the request override those from the URL.

The `_id` field indicates the ID of the document you're indexing. If you omit that, Elasticsearch will automatically generate an ID for you, which is helpful if you don't already have a unique ID for your documents. Logs, for example, work well with generated IDs because they don't typically have a natural unique ID and you don't need to retrieve logs by ID.

If you don't need to provide IDs and you index all documents in the same index and type, the bulk request from listing 10.1 gets quite a lot simpler, as shown in the following listing.

Listing 10.2 Indexing two documents in the same index and type with automatic IDs

```

REQUESTS_FILE=/tmp/test_bulk
echo '{"index":{}}
{"name":"Elasticsearch Bucharest"}
{"index":{}}
{"name":"Big Data Bucharest"}
' > $REQUESTS_FILE
URL='localhost:9200/get-together/group'

curl -XPOST $URL/_bulk?pretty --data-binary @$REQUESTS_FILE

```

← Specifying only the operation, because index and type are provided in the URL and IDs will be automatically generated

← Specifying the index and type in the URL

The result of your bulk insert should be a JSON containing the time it took to index your bulk and the responses for each operation. There's also an `errors` flag, which indicates whether any of the operations failed. The whole response should look something like this:

```

{
  "took" : 2,
  "errors" : false,
  "items" : [ {
    "create" : {
      "_index" : "get-together",
      "_type" : "group",
      "_id" : "AUyDuQED0pziDTnH-426",
      "_version" : 1,
      "status" : 201
    }
  }, {
    "create" : {
      "_index" : "get-together",
      "_type" : "group",
      "_id" : "AUyDuQED0pziDTnH-426",
      "_version" : 1,
      "status" : 201
    }
  } ]
}

```

Note that because you've used automatic ID generation, the `index` operations were changed to `create`. If one document can't be indexed for some reason, it doesn't mean the whole bulk has failed, because items from the same bulk are independent of each other. That's why you get a reply for each operation, instead of one for the whole bulk. You can use the response JSON in your application to determine which operation succeeded and which failed.

TIP When it comes to performance, bulk size matters. If your bulks are too big, they take too much memory. If they're too small, there's too much network overhead. The sweet spot depends on document size—you'd put a few big documents or more smaller ones in a bulk—and on the cluster's firepower. A big cluster with strong machines can process bigger bulks faster and still serve searches with decent performance. In the end, you have to test

and find the sweet spot for your use case. You can start with values like 1,000 small documents (such as logs) per bulk and increase until you don't get a significant gain. Be sure to monitor your cluster in the meantime, as we'll discuss in lab 11.

UPDATING OR DELETING IN BULKS

Within a single bulk, you can have any number of `index` or `create` operations and also any number of `update` or `delete` operations.

`update` operations look similar to the `index/create` operations we just discussed, except for the fact that you must specify the ID. Also, the document content would contain `doc` or `script` according to the way you want to update, just as you specified `doc` or `script` in lab 3 when you did individual updates.

`delete` operations are a bit different than the rest because you have no document content. You just have the metadata line, like with updates, which has to contain the document's ID.

In the next listing you have a bulk that contains all four operations: `index`, `create`, `update`, and `delete`.

Listing 10.3 Bulk with `index`, `create`, `update`, and `delete`

```
echo '{"index":{}}
{"title":"Elasticsearch Bucharest"}
{"create":{}}
{"title":"Big Data in Romania"}
{"update":{"_id": "11"}}
{"doc":{"created_on" : "2014-05-06"} }
{"delete":{"_id": "10"}}
' > $REQUESTS_FILE
URL='localhost:9200/get-together/group'
```

Update operation:
specify the ID and the
partial document.

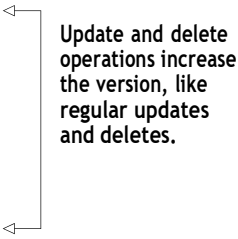
Delete operation:
no document is
needed, just the ID.

```
curl -XPOST $URL/_bulk?pretty --data-binary @$REQUESTS_FILE
# expected reply
{
  "took" : 37,
  "errors" : false,
  "items" : [ {
    "create" : {
      "_index" : "get-together",
      "_type" : "group",
      "_id" : "rVPTtooieSxqfM6_JX-UCkg",
      "_version" : 1,
      "status" : 201
    }
  }, {
    "create" : {
      "_index" : "get-together",
      "_type" : "group",
      "_id" : "8w3GoNg5T_WEIL5jSTz_Ug",
      "_version" : 1,
      "status" : 201
    }
  }, {
    "update" : {
      "_index" : "get-together",
      "_type" : "group",
      "_id" : "11",
      "doc" : {
        "created_on" : "2014-05-06"
      },
      "status" : 201
    }
  }, {
    "delete" : {
      "_index" : "get-together",
      "_type" : "group",
      "_id" : "10",
      "status" : 201
    }
  }
]
```

```

    "update" : {
      "_index" : "get-together",
      "_type" : "group",
      "_id" : "11",
      "_version" : 2,
      "status" : 200
    }
  }, {
    "delete" : {
      "_index" : "get-together",
      "_type" : "group",
      "_id" : "10",
      "_version" : 2,
      "status" : 200,
      "found" : true
    }
  }

```



Update and delete operations increase the version, like regular updates and deletes.

If the bulk APIs can be used to group multiple `index`, `update`, and `delete` operations together, you can do the same for `search` and `get` requests with the `multisearch` and `multiget` APIs, respectively. We'll look at these next.

10.1.2 Multisearch and multiget APIs

The benefit of using `multisearch` and `multiget` is the same as with bulks: when you have to do multiple `search` or `get` requests, grouping them together saves time otherwise spent on network latency.

MULTISEARCH

One use case for sending multiple search requests at once occurs when you're searching in different types of documents. For example, let's assume you have a search box in your `get-together` site. You don't know whether a search is for groups or for events, so you're going to search for both and offer different tabs in the UI: one for groups and one for events. Those two searches would have completely different scoring criteria, so you'd run them in different requests, or you could group these requests together in a `multisearch` request.

The `multisearch` API has many similarities with the bulk API:

- You hit the `_msearch` endpoint, and you may or may not specify an index and a type in the URL.
- Each request has two single-line JSON strings: the first may contain parameters like index, type, routing value, or search type—that you'd normally put in the URI of a single request. The second line contains the query body, which is normally the payload of a single request.

The listing that follows shows an example `multisearch` request for events and groups about Elasticsearch.

Listing 10.4 Multisearch request for events and groups about Elasticsearch

For every other search, you have a header and a body line.

The header of each search contains data that can go to the URL of a single search.

The body contains the query, as you have with single searches.

As with bulk requests, it's important to preserve newline characters.

The response is an array of individual search results.

Reply for the first query about groups

All replies look like individual query replies.

```

echo '{"index" : "get-together", "type": "group"}
{"query" : {"match" : {"name": "elasticsearch"}}}
{"index" : "get-together", "type": "event"}
{"query" : {"match" : {"title": "elasticsearch"}}}
' > request
curl localhost:9200/_msearch?pretty --data-binary @request
# reply
{
  "responses" : [ {
    "took" : 4,
    [...]
    "hits" : [ {
      "_index" : "get-together",
      "_type" : "group",
      "_id" : "2",
      "_score" : 1.8106999,
      "_source":{
        "name": "Elasticsearch Denver",
        [...]
      }, {
        "took" : 7,
        [...]

        "hits" : [ {
          "_index" : "get-together",
          "_type" : "event",
          "_id" : "103",
          "_score" : 0.9581454,
          "_source":{
            "host": "Lee",
            "title": "Introduction to Elasticsearch",
            [...]
          }
        ]
      }
    ]
  }
]

```

MULTIGET

Multiget makes sense when some processing external to Elasticsearch requires you to fetch a set of documents without doing any search. For example, if you're storing system metrics and the ID is a timestamp, you might need to retrieve specific metrics from specific times without doing any filtering. To do that, you'd call the `_mget` end-point and send a `docs` array with the index, type, and ID of the documents you want to retrieve, as in the next listing.

Listing 10.5 `_mget` endpoint and `docs` array with index, type, and ID of documents

```

curl localhost:9200/_mget?pretty -d '{
  "docs" : [
    {
      "_index" : "get-together",
      "_type" : "group",
      "_id" : "1"
    }
  ]
}'

```


The docs array identifies all documents that you want to retrieve.

```

    },
    {
      "_index" : "get-together",
      "_type" : "group",
      "_id" : "2"
    }
  ]
}'
# reply
{
  "docs" : [ {
    "_index" : "get-together",
    "_type" : "group",
    "_id" : "1",
    "_version" : 1,
    "found" : true,
    "_source":{
      "name": "Denver Clojure",
    [...]
  }, {
    "_index" : "get-together",
    "_type" : "group",
    "_id" : "2",
    "_version" : 1,
    "found" : true,
    "_source":{
      "name": "Elasticsearch Denver",
    [...]
  }
]
}
```

The reply
also contains
a docs array.

Each element of
the array is the
document as you
get it with single
GET requests.

As with most other APIs, the index and type are optional, because you can also put them in the URL of the request. When the index and type are common for all IDs, it's recommended to put them in the URL and put the IDs in an `ids` array, making the request from listing 10.5 much shorter:

```
% curl localhost:9200/get-together/group/_mget?pretty -d '{
  "ids" : [ "1", "2" ]
}'
```

Grouping multiple operations in the same requests with the multiget API might introduce a little complexity to your application, but it will make such requests faster without significant costs. The same applies to the multisearch and bulk APIs, and to make the best use of them, you can experiment with different request sizes and see which size works best for your documents and your hardware.

Next, we'll look at how Elasticsearch processes documents in bulks internally, in the form of Lucene segments, and how you can tune these processes to speed up indexing and searching.

10.2 Optimizing the handling of Lucene segments

Once Elasticsearch receives documents from your application, it indexes them in memory in inverted indices called *segments*. From time to time, these segments are

written to disk. Recall from lab 3 that these segments can't be changed—only deleted—to make it easy for the operating system to cache them. Also, bigger segments are periodically created from smaller segments to consolidate the inverted indices and make searches faster.

There are lots of knobs to influence how Elasticsearch handles these segments at every step, and configuring them to fit your use case often gives important performance gains. In this section, we'll discuss these knobs and divide them into three categories:

- *How often to refresh and flush*—*Refreshing* reopens Elasticsearch's view on the index, making newly indexed documents available for search. *Flushing* commits indexed data from memory to the disk. Both refresh and flush operations are expensive in terms of performance, so it's important to configure them correctly for your use case.
- *Merge policies*—Lucene (and by inheritance, Elasticsearch) stores data into immutable groups of files called segments. As you index more data, more segments are created. Because a search in many segments is slow, small segments are merged in the background into bigger segments to keep their number manageable. Merging is performance intensive, especially for the I/O subsystem. You can adjust the merge policy to influence how often merges happen and how big segments can get.
- *Store and store throttling*—Elasticsearch limits the impact of merges on your system's I/O to a certain number of bytes per second. Depending on your hardware and use case, you can change this limit. There are also other options for how Elasticsearch uses the storage. For example, you can choose to store your indices only in memory.

We'll start with the category that typically gives you the biggest performance gain of the three: choosing how often to refresh and flush.

10.2.1 *Refresh and flush thresholds*

Recall from lab 2 that Elasticsearch is often called near real time; that's because searches are often not run on the very latest indexed data (which would be real time) but close to it.

This near-real-time label fits because normally Elasticsearch keeps a point-in-time view of the index opened, so multiple searches would hit the same files and reuse the same caches. During this time, newly indexed documents won't be visible to those searches until you do a refresh.

Refreshing, as the name suggests, refreshes this point-in-time view of the index so your searches can hit your newly indexed data. That's the upside. The downside is that each refresh comes with a performance penalty: some caches will be invalidated, slowing down searches, and the reopening process itself needs processing power, slowing down indexing.

WHEN TO REFRESH

The default behavior is to refresh every index automatically every second. You can change the interval for every index by changing its settings, which can be done at run-time. For example, the following command will set the automatic refresh interval to 5 seconds:

```
% curl -XPUT localhost:9200/get-together/_settings -d '{
  "index.refresh_interval": "5s"
}'
```

TIP To confirm that your changes were applied, you can get all the index settings by running `curl localhost:9200/get-together/_settings?pretty`.

As you increase the value of `refresh_interval`, you'll have more indexing throughput because you'll spend fewer system resources on refreshing.

Alternatively, you can set `refresh_interval` to `-1` to effectively disable automatic refreshes and rely on manual refresh. This works well for use cases where indices change only periodically in batches, such as for a retail chain where products and stocks are updated every night. Indexing throughput is important because you want to consume those updates quickly, but data freshness isn't, because you don't get the updates in real time, anyway. So you can do nightly bulk index/updates with automatic refresh disabled and refresh manually when you've finished.

To refresh manually, hit the `_refresh` endpoint of the index (or indices) you want to refresh:

```
% curl localhost:9200/get-together/_refresh
```

WHEN TO FLUSH

If you're used to older versions of Lucene or Solr, you might be inclined to think that when a refresh happens, all data that was indexed (in memory) since the last refresh is also committed to disk.

With Elasticsearch (and Solr 4.0 or later) the process of refreshing and the process of committing in-memory segments to disk are independent. Indeed, data is indexed first in memory, but after a refresh, Elasticsearch will happily search the in-memory segments as well. The process of committing in-memory segments to the actual Lucene index you have on disk is called a *flush*, and it happens whether the segments are searchable or not.

To make sure that in-memory data isn't lost when a node goes down or a shard is relocated, Elasticsearch keeps track of the indexing operations that weren't flushed yet in a transaction log. Besides committing in-memory segments to disk, a flush also clears the transaction log, as shown in figure 10.2.

A flush is triggered in one of the following conditions, as shown in figure 10.3:

- The memory buffer is full.
- A certain amount of time passed since the last flush.
- The transaction log hit a certain size threshold.

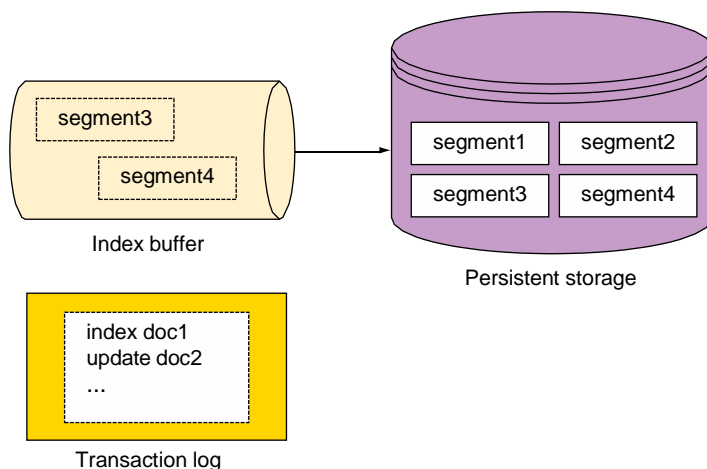


Figure 10.2 A flush moves segments from memory to disk and clears the transaction log.

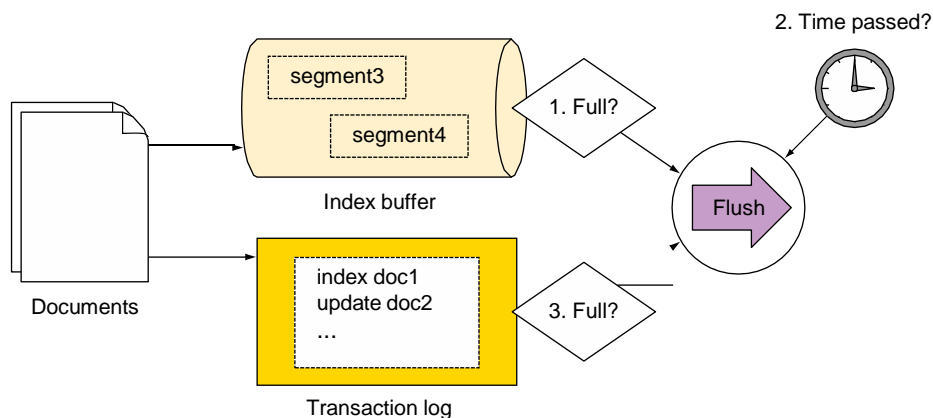


Figure 10.3 A flush is triggered when the memory buffer or transaction log is full or at an interval.

To control how often a flush happens, you have to adjust the settings that control those three conditions.

The memory buffer size is defined in the `elasticsearch.yml` configuration file through the `indices.memory.index_buffer_size` setting. This controls the overall buffer for the entire node, and the value can be either a percent of the overall JVM heap like 10% or a fixed value like 100 MB.

Transaction log settings are index specific and control both the size at which a flush is triggered (via `index.translog.flush_threshold_size`) and the time since

the last flush (via `index.translog.flush_threshold_period`). As with most index settings, you can change them at runtime:

```
% curl -XPUT localhost:9200/get-together/_settings -d '{
  "index.translog": {
    "flush_threshold_size": "500mb",
    "flush_threshold_period": "10m"
  }
}'
```

When a flush is performed, one or more segments are created on the disk. When you run a query, Elasticsearch (through Lucene) looks in all segments and merges the results in an overall shard result. Then, as you saw in lab 2, per-shard results are aggregated into the overall results that go back to your application.

The key thing to remember here about segments is that the more segments you have to search through, the slower the search. To keep the number of segments at bay, Elasticsearch (again, through Lucene) merges multiple sets of smaller segments into bigger segments in the background.

10.2.2 Merges and merge policies

We first introduced segments in lab 3 as immutable sets of files that Elasticsearch uses to store indexed data. Because they don't change, segments are easily cached, making searches fast. Also, changes to the dataset, such as the addition of a document, won't require rebuilding the index for data stored in existing segments. This makes indexing new documents fast, too—but it's not all good news. Updating a document can't change the actual document; it can only index a new one. This requires deleting the old document, too. Deleting, in turn, can't remove a document from its segment (that would require rebuilding the inverted index), so it's only marked as deleted in a separate `.del` file. Documents are only actually removed during segment merging.

This brings us to the two purposes of merging segments: to keep the total number of segments in check (and with it, query performance) and to remove deleted documents.

Segment merging happens in the background, according to the defined merge policy. The default merge policy is tiered, which, as illustrated in figure 10.4, divides segments into tiers, and if you have more than the set maximum number of segments in a tier, a merge is triggered in that tier.

There are other merge policies, but in this lab we'll focus only on the tiered merge policy, which is the default, because it works best for most use cases.

TIP There are some nice videos and explanations of different merge policies on Mike McCandless's blog (he's a co-author of *Lucene in Action*, Second Edition [Manning Publications, 2010]): <http://blog.mikemccandless.com/2011/02/visualizing-lucenes-segment-merges.html>.

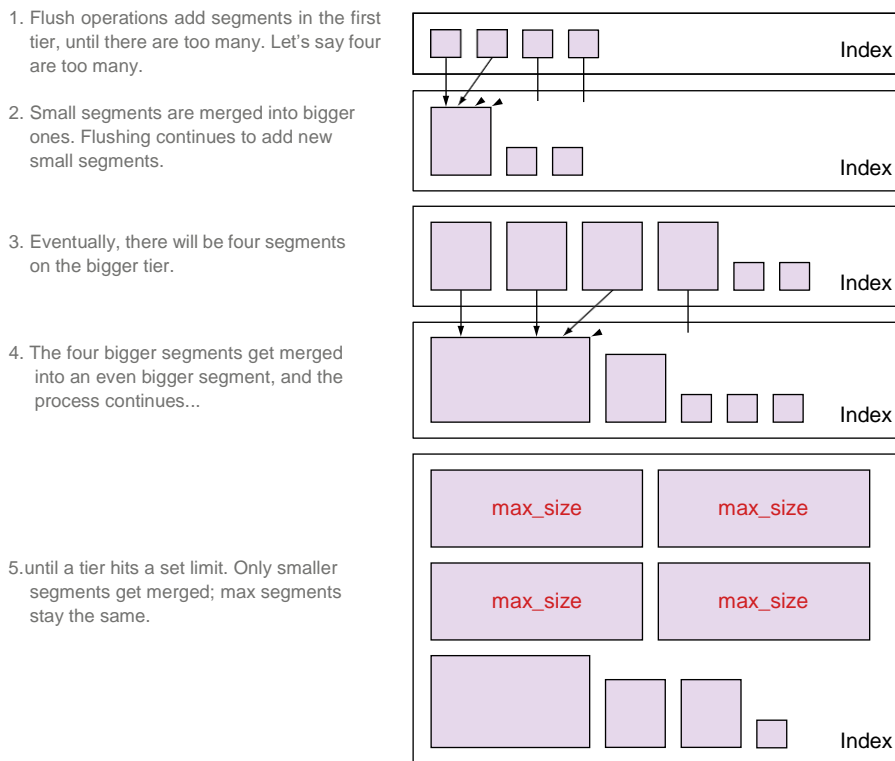


Figure 10.4 Tiered merge policy performs a merge when it finds too many segments in a tier.

TUNING MERGE POLICY OPTIONS

The overall purpose of merging is to trade I/O and some CPU time for search performance. Merging happens when you index, update, or delete documents, so the more you merge, the more expensive these operations get. Conversely, if you want faster indexing, you'll need to merge less and sacrifice some search performance.

In order to have more or less merging, you have a few configuration options. Here are the most important ones:

- `index.merge.policy.segments_per_tier`—The higher the value, the more segments you can have in a tier. This will translate to less merging and better indexing performance. If you have little indexing and you want better search performance, lower this value.
- `index.merge.policy.max_merge_at_once`—This setting limits how many segments can be merged at once. You'd typically make it equal to the `segments_per_tier` value. You could lower the `max_merge_at_once` value to force less merging, but it's better to do that by increasing `segments_per_tier`. Make sure

`max_merge_at_once` isn't higher than `segments_per_tier` because that will cause too much merging.

- `index.merge.policy.max_merged_segment`—This setting defines the maximum segment size; bigger segments won't be merged with other segments. You'd lower this value if you wanted less merging and faster indexing because larger segments are more difficult to merge.
- `index.merge.scheduler.max_thread_count`—Merging happens in the background on separate threads, and this setting controls the maximum number of threads that can be used for merging. This is the hard limit of how many merges can happen at once. You'd increase this setting for an aggressive merge policy on a machine with many CPUs and fast I/O, and you'd decrease it if you had a slow CPU or I/O.

All those options are index-specific, and, as with transaction log and refresh settings, you can change them at runtime. For example, the following snippet forces more merging by reducing `segments_per_tier` to 5 (and with it, `max_merge_at_once`), lowers the maximum segment size to 1 GB, and lowers the thread count to 1 to work better with spinning disks:

```
% curl -XPUT localhost:9200/get-together/_settings -d '{
  "index.merge": {
    "policy": {
      "segments_per_tier": 5,
      "max_merge_at_once": 5,
      "max_merged_segment": "1gb"
    },
    "scheduler.max_thread_count": 1
  }
}'
```

OPTIMIZING INDICES

As with refreshing and flushing, you can trigger a merge manually. A forced merge call is also known as *optimize*, because you'd typically run it on an index that isn't going to be changed later to optimize it to a specified (low) number of segments for faster searching.

As with any aggressive merge, optimizing is I/O intensive and invalidates lots of caches. If you continue to index, update, or delete documents from that index, new segments will be created and the advantages of optimizing will be lost. Thus, if you want fewer segments on an index that's constantly changing, you should tune the merge policy.

Optimizing makes sense on a static index. For example, if you index social media data and you have one index per day, you know you'll never change yesterday's index until you remove it for good. It might help to optimize it to a low number of segments, as shown in figure 10.5, which will reduce its total size and speed up queries once caches are warmed up again.

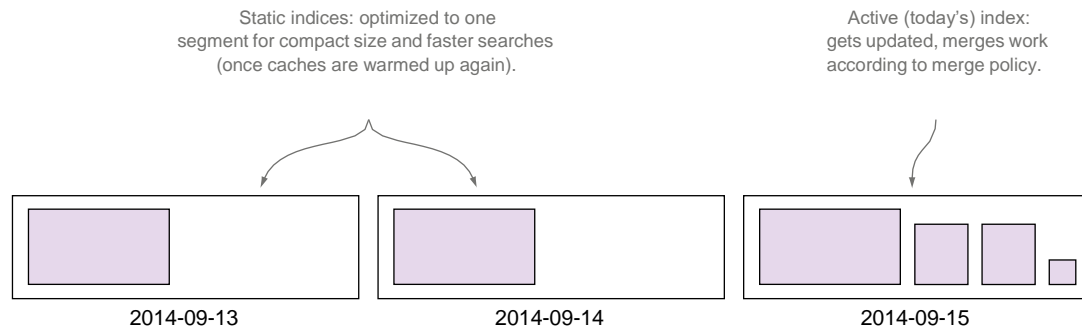


Figure 10.5 Optimizing makes sense for indices that don't get updates.

To optimize, you'd hit the `_optimize` endpoint of the index or indices you need to optimize. The `max_num_segments` option indicates how many segments you should end up with per shard:

```
% curl localhost:9200/get-together/_optimize?max_num_segments=1
```

An optimize call can take a long time on a large index. You can send it to the background by setting `wait_for_merge` to `false`.

One possible reason for an optimize (or any merge) being slow is that Elasticsearch, by default, limits the amount of I/O throughput merge operations can use. This limiting is called *store throttling*, and we'll discuss it next, along with other options for storing your data.

10.2.3 Store and store throttling

In early versions of Elasticsearch, heavy merging could slow down the cluster so much that indexing and search requests would take unacceptably long, or nodes could become unresponsive altogether. This was all due to the pressure of merging on the I/O throughput, which would make the writing of new segments slow. Also, CPU load was higher due to I/O wait.

As a result, Elasticsearch now limits the amount of I/O throughput that merges can use through store throttling. By default, there's a node-level setting called `indices.store.throttle.max_bytes_per_sec`, which defaults to 20mb as of version 1.5.

This limit is good for stability in most use cases but won't work well for everyone. If you have fast machines and lots of indexing, merges won't keep up, even if there's enough CPU and I/O to perform them. In such situations, Elasticsearch makes internal indexing work only on one thread, slowing it down to allow merges to keep up. In the end, if your machines are fast, indexing might be limited by store throttling. For nodes with SSDs, you'd normally increase the throttling limit to 100–200 MB.

CHANGING STORE THROTTLING LIMITS

If you have fast disks and need more I/O throughput for merging, you can raise the store throttling limit. You can also remove the limit altogether by setting `indices.store.throttle.type` to `none`. On the other end of the spectrum, you can apply the store throttling limit to all of Elasticsearch's disk operations, not just merge, by setting `indices.store.throttle.type` to `all`.

Those settings can be changed from `elasticsearch.yml` on every node, but they can also be changed at runtime through the Cluster Update Settings API. Normally, you'd tune them while monitoring how much merging and other disk activities are actually happening—we'll show you how to do that in lab 11.

TIP Elasticsearch 2.0, which will be based on Lucene 5.0, will use Lucene's auto-io-throttle feature,¹ which will automatically throttle merges based on how much indexing is going on. If there's little indexing, merges will be throttled more so they won't affect searches. If there's lots of indexing, there will be less merge throttling, so that merges won't fall behind.

The following command would raise the throttling limit to 500 MB/s but apply it to all operations. It would also make the change persistent to survive full cluster restarts (which is opposed to transient settings that are lost when the cluster is restarted):

```
% curl -XPUT localhost:9200/_cluster/settings -d '{
  "persistent": {
    "indices.store.throttle": {
      "type": "all",
      "max_bytes_per_sec": "500mb"
    }
  }
}'
```

TIP As with index settings, you can also get cluster settings to see if they're applied. You'd do that by running `curl localhost:9200/_cluster/settings?pretty`.

CONFIGURING STORE

When we talked about flushes, merges, and store throttling, we said “disk” and “I/O” because that's the default: Elasticsearch will store indices in the data directory, which defaults to `/var/lib/elasticsearch/data` if you installed Elasticsearch from a RPM/DEB package, or the `data/` directory from the unpacked tar.gz or ZIP archive if you installed it manually. You can change the data directory from the `path.data` property of `elasticsearch.yml`.

¹ For more details, check the Lucene issue, <https://issues.apache.org/jira/browse/LUCENE-6119>, and the Elasticsearch issue, <https://github.com/elastic/elasticsearch/pull/9243>.

TIP You can specify multiple directories in `path.data` which—in version 1.5, at least—will put different files in different directories to achieve striping (assuming those directories are on different disks). If that’s what you’re after, you’re often better off using RAID0, in terms of both performance and reliability. For this reason, the plan is to put each shard in the same directory instead of striping it.²

The default store implementation stores index files in the file system, and it works well for most use cases. To access Lucene segment files, the default store implementation uses Lucene’s `MMapDirectory` for files that are typically large or need to be randomly accessed, such as term dictionaries. For the other types of files, such as stored fields, Elasticsearch uses Lucene’s `NIOFSDirectory`.

MMapDirectory

`MMapDirectory` takes advantage of file system caches by asking the operating system to map the needed files in virtual memory in order to access that memory directly. To Elasticsearch, it looks as if all the files are available in memory, but that doesn’t have to be the case. If your index size is larger than your available physical memory, the operating system will happily take unused files out of the caches to make room for new ones that need to be read. If Elasticsearch needs those uncached files again, they’ll be loaded in memory while other unused files are taken out and so on. The virtual memory used by `MMapDirectory` works similarly to the system’s virtual memory (swap), where the operating system uses the disk to page out unused memory in order to be able to serve multiple applications.

NIOFSDirectory

Memory-mapped files also imply an overhead because the application has to tell the operating system to map a file before accessing it. To reduce this overhead, Elasticsearch uses `NIOFSDirectory` for some types of files. `NIOFSDirectory` accesses files directly, but it has to copy the data it needs to read in a buffer in the JVM heap. This makes it good for small, sequentially accessed files, whereas `MMapDirectory` works well for large, randomly accessed files.

The default store implementation is best for most use cases. You can, however, choose other implementations by changing `index.store.type` in the index settings to values other than default:

- `mmapfs`—This will use the `MMapDirectory` alone and will work well, for example, if you have a relatively static index that fits in your physical memory.
- `niofs`—This will use `NIOFSDirectory` alone and would work well on 32-bit systems, where virtual memory address space is limited to 4 GB, which will prevent you from using `mmapfs` or `default` for larger indices.

² More details can be found on Elasticsearch’s bug tracker: <https://github.com/elastic/elasticsearch/issues/9498>.

Store type settings need to be configured when you create the index. For example, the following command creates an mmap-ed index called unit-test:

```
% curl -XPUT localhost:9200/unit-test -d '{
  "index.store.type": "mmapfs"
}'
```

If you want to apply the same store type for all newly created indices, you can set `index.store.type` to `mmapfs` in `elasticsearch.yml`. In lab 11 we'll introduce index templates, which allow you to define index settings that would apply to new indices matching specific patterns. Templates can also be changed at runtime, and we recommend using them instead of the more static `elasticsearch.yml` equivalent if you often create new indices.

Open files and virtual memory limits

Lucene segments that are stored on disk can spread onto many files, and when a search runs, the operating system needs to be able to open many of them. Also, when you're using the default store type or `mmapfs`, the operating system has to map some of those stored files into memory—even though these files aren't in memory, to the application it's like they are, and the kernel takes care of loading and unloading them in the cache. Linux has configurable limits that prevent the applications from opening too many files at once and from mapping too much memory. These limits are typically more conservative than needed for Elasticsearch deployments, so it's recommended to increase them. If you're installing Elasticsearch from a DEB or RPM package, you don't have to worry about this because they're increased by default. You can find these variables in `/etc/default/elasticsearch` or `/etc/sysconfig/elasticsearch`:

```
MAX_OPEN_FILES=65535
MAX_MAP_COUNT=262144
```

To increase those limits manually, you have to run `ulimit -n 65535` as the user who starts Elasticsearch for the open files and run `sysctl -w vm.max_map_count=262144` as root for the virtual memory.

The default store type is typically the fastest because of the way the operating system caches files. For caching to work well, you need to have enough free memory.

TIP From Elasticsearch 2.0 on, you'll be able to compress stored fields (and `_source`) further by setting `index.codec` to `best_compression`.³ The default (named `default`, as with store types) still compresses stored fields by using

³ For more details, check the Elasticsearch issue, <https://github.com/elastic/elasticsearch/pull/8863>, and the main Lucene issue, <https://issues.apache.org/jira/browse/LUCENE-5914>.

LZ4, but `best_compression` uses deflate.⁴ Higher compression will slow down operations that need `_source`, like fetching results or highlighting. Other operations, such as aggregations, should be at least equally fast because the overall index will be smaller and easier to cache.

We mentioned how `merge` and `optimize` operations invalidate caches. Managing caches for Elasticsearch to perform well deserves more explanation, so we'll discuss that next.

10.3 *Making the best use of caches*

One of Elasticsearch's strong points—if not the strongest point—is the fact that you can query billions of documents in milliseconds with commodity hardware. And one of the reasons this is possible is its smart caching. You might have noticed that after indexing lots of data, the second query can be orders of magnitude faster than the first one. It's because of caching—for example, when you combine filters and queries—that the filter cache plays an important role in keeping your searches fast.

In this section we'll discuss the filter cache and two other types of caches: the shard query cache, useful when you run aggregations on static indices because it caches the overall result, and the operating system caches, which keep your I/O throughput high by caching indices in memory.

Finally, we'll show you how to keep all those caches warm by running queries at each refresh with index warmers. Let's start by looking at the main type of Elasticsearch-specific cache—the filter cache—and how you can run your searches to make the best use of it.

10.3.1 *Filters and filter caches*

In lab 4 you saw that lots of queries have a filter equivalent. Let's say that you want to look for events on the get-together site that happened in the last month. To do that, you could use the range query or the equivalent range filter.

In lab 4 we said that of the two, we recommend using the filter, because it's cacheable. The range filter is cached by default, but you can control whether a filter is cached or not through the `_cache` flag.

TIP Elasticsearch 2.0 will cache, by default, only frequently used filters and only on bigger segments (that were merged at least once). This should prevent caching too aggressively but should also catch frequent filters and optimize them. More implementation details can be found in the Elasticsearch⁵ and Lucene⁶ issues about filter caching. This flag applies to all filters; for

⁴ <https://en.wikipedia.org/wiki/DEFLATE>

⁵ <https://github.com/elastic/elasticsearch/pull/8573>

⁶ <https://issues.apache.org/jira/browse/LUCENE-6077>

example, the following snippet will filter events with "elasticsearch" in the `verbatim` tag but won't cache the results:

```
% curl localhost:9200/get-together/group/_search?pretty -d '{
  "query": {
    "filtered": {
      "filter": {
        "term": {
          "tags.verbatim": "elasticsearch",
          "_cache": false
        }
      }
    }
  }
}'
```

NOTE Although all filters have the `_cache` flag, it doesn't apply in 100% of cases. For the range filter, if you use "now" as one of the boundaries, the flag is ignored. For the `has_child` or `has_parent` filters, the `_cache` flag doesn't apply at all.

FILTER CACHE

The results of a filter that's cached are stored in the filter cache. This cache is allocated at the node level, like the index buffer size you saw earlier. It defaults to 10%, but you can change it from `elasticsearch.yml` according to your needs. If you use filters a lot and cache them, it might make sense to increase the size. For example:

```
indices.cache.filter.size: 30%
```

How do you know if you need more (or less) filter cache? By monitoring your actual usage. As we'll explore in lab 11 on administration, Elasticsearch exposes lots of metrics, including the amount of filter cache that's actually used and the number of cache evictions. An *eviction* happens when the cache gets full and Elasticsearch drops the least recently used (LRU) entry in order to make room for the new one.

In some use cases, filter cache entries have a short lifespan. For example, users typically filter get-together events by a particular subject, refine their queries until they find what they want, and then leave. If nobody else is searching for events on the same subject, that cache entry will stick around doing nothing until it eventually gets evicted. A full cache with many evictions would make performance suffer because every search will consume CPU cycles to squeeze new cache entries by evicting old ones.

In such use cases, to prevent evictions from happening exactly when queries are run, it makes sense to set a time to live (TTL) on cache entries. You can do that on a per-index basis by adjusting `index.cache.filter.expire`. For example, the following snippet will expire filter caches after 30 minutes:

```
% curl -XPUT localhost:9200/get-together/_settings -d '{
  "index.cache.filter.expire": "30m"
}'
```

Besides making sure you have enough room in your filter caches, you need to run your filters in a way that takes advantage of these caches.

COMBINING FILTERS

You often need to combine filters—for example, when you’re searching for events in a certain time range, but also with a certain number of attendees. For best performance, you’ll need to make sure that caches are well used when filters are combined and that filters run in the right order.

To understand how to best combine filters, we need to revisit a concept discussed in lab 4: bitsets. A *bitset* is a compact array of bits, and it’s used by Elasticsearch to cache whether a document matches a filter or not. Most filters (such as the `range` and `terms` filter) use bitsets for caching. Other filters, such as the `script` filter, don’t use bitsets because Elasticsearch has to iterate through all documents anyway. Table 10.1 shows which of the important filters use bitsets and which don’t.

Table 10.1 Which filters use bitsets

Filter type	Uses bitset
<code>term</code>	Yes
<code>terms</code>	Yes, but you can configure it differently, as we’ll explain in a bit
<code>exists/missing</code>	Yes
<code>prefix</code>	Yes
<code>regexp</code>	No
<code>nested/has_parent/has_child</code>	No
<code>script</code>	No
<code>geo filters</code> (see appendix A)	No

For filters that don’t use bitsets, you can still set `_cache` to `true` in order to cache results of that exact filter. *Bitsets* are different than simply caching the results because they have the following characteristics:

- They’re compact and easy to create, so the overhead of creating the cache when the filter is first run is insignificant.
- They’re stored per individual filter; for example, if you use a `term` filter in two different queries or within two different `bool` filters, the bitset of that `term` can be reused.
- They’re easy to combine with other bitsets. If you have two queries that use bitsets, it’s easy for Elasticsearch to do a bitwise `AND` or `OR` in order to figure out which documents match the combination.

To take advantage of bitsets, you need to combine filters that use them in a `bool` filter that will do that bitwise `AND` or `OR`, which is easy for your CPU. For example, if you want

to show only groups where either Lee is a member or that contain the tag `elasticsearch`, it could look like this:

```
"filter": {
  "bool": {
    "should": [
      {
        "term": {
          "tags.verbatim": "elasticsearch"
        }
      },
      {
        "term": {
          "members": "lee"
        }
      }
    ]
  }
}
```

The alternative to combining filters is using the `and`, `or`, and `not` filters. These filters work differently because unlike the `bool` filter, they don't use bitwise AND or OR. They run the first filter, pass the matching documents to the next one, and so on. As a result, `and`, `or`, and `not` filters are better when it comes to combining filters that don't use bitsets. For example, if you want to show groups having at least three members, with events organized in July 2013, the filter might look like this:

```
"filter": {
  "and": [
    {
      "has_child": {
        "type": "event",
        "filter": {
          "range": {
            "date": {
              "from": "2013-07-01T00:00",
              "to": "2013-08-01T00:00"
            }
          }
        }
      }
    },
    {
      "script": {
        "script": "doc['members'].values.length > minMembers",
        "params": {
          "minMembers": 2
        }
      }
    }
  ]
}
```


If you're using both bitset and nonbitset filters, you can combine the bitset ones in a `bool` filter and put that `bool` filter in an `and/or/not` filter, along with the nonbitset filters. For example, in the next listing you'll look for groups with at least two members where either Lee is one of them or the group is about Elasticsearch.

Listing 10.6 Combine bitset filters in a `bool` filter inside an `and/or/not` filter

```
curl localhost:9200/get-together/group/_search?pretty -d' {
  "query": {
    "filtered": {
      "filter": {
        "and": [
          {
            "bool": {
              "should": [
                {
                  "term": {
                    "tags.verbatim": "elasticsearch"
                  }
                },
                {
                  "term": {
                    "members": "lee"
                  }
                }
              ]
            }
          },
          {
            "script": {
              "script": "doc[\"members\"].values.length > minMembers",
              "params": {
                "minMembers": 2
              }
            }
          }
        ]
      }
    }
  }
}
```

The AND filter will run the bool filter first.

Filtered query means if you add a query here, it will run only on documents matching the filter.

bool is fast when cached because it makes use of the two bitsets of the term filters.

The script filter will work only on documents matching the bool filter.

Whether you combine filter with the `bool`, `and`, `or`, or `not` filters, the order in which those filters are executed is important. Cheaper filters, such as the `term` filter, should be placed before expensive filters, such as the `script` filter. This would make the expensive filter run on a smaller set of documents—those that already matched previous filters.

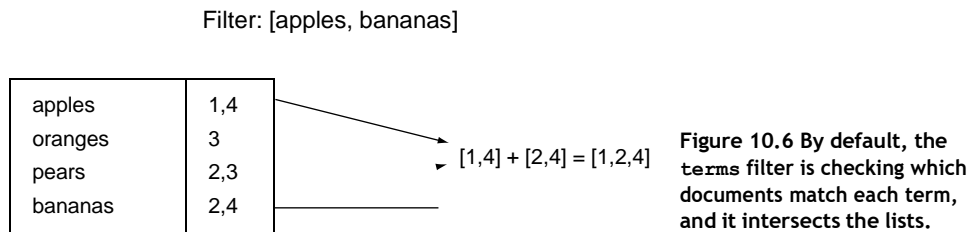
RUNNING FILTERS ON FIELD DATA

So far, we've discussed how bitsets and cached results make your filters faster. Some filters use bitsets; some can cache the overall results. Some filters can also run on field data. We first discussed field data in lab 6 as an in-memory structure that keeps a

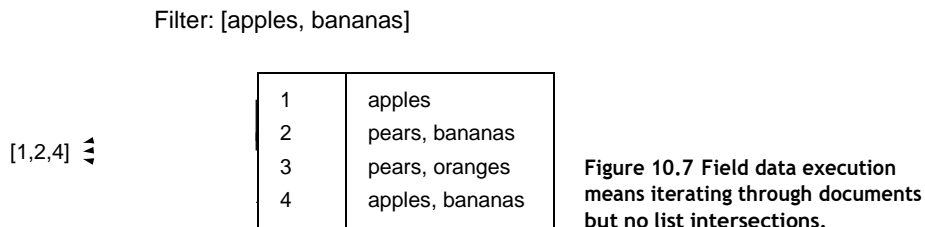
mapping of documents to terms. This mapping is the opposite of the inverted index, which maps terms to documents. Field data is typically used when sorting and during aggregations, but some filters can use it, too: the `terms` and the `range` filters.

NOTE An alternative to the in-memory field data is to use doc values, which are calculated at index time and stored on disk with the rest of your index. As we pointed out in lab 6, doc values work for numeric and not-analyzed string fields. In Elasticsearch 2.0, doc values will be used by default for those fields because holding field data in the JVM heap is usually not worth the performance increase.

A `terms` filter can have lots of terms, and a `range` filter with a wide range will (under the hood) match lots of numbers (and numbers are also terms). Normal execution of those filters will try to match every term separately and return the set of unique documents, as illustrated in figure 10.6.



As you can imagine, filtering on many terms could get expensive because there would be many lists to intersect. When the number of terms is large, it can be faster to take the actual field values one by one and see if the terms match instead of looking in the index, as illustrated in figure 10.7.



These field values would be loaded in the field data cache by setting `execution` to `fielddata` in the `terms` or `range` filters. For example, the following `range` filter will get events that happened in 2013 and will be executed on field data:

```
"filter": {
  "range": {
    "date": {
```

```

    "gte": "2013-01-01T00:00",
    "lt": "2014-01-01T00:00"
  },
  "execution": "fielddata"
}

```

Using field data execution is especially useful when the field data is already used by a sort operation or an aggregation. For example, running a `terms` aggregation on the `tags` field will make a subsequent `terms` filter for a set of tags faster because the `fielddata` is already loaded.

Other execution modes for the terms filter: `bool` and `and/or`

The `terms` filter has other execution modes, too. If the default execution mode (called `plain`) builds a bitset to cache the overall result, you can set it to `bool` in order to have a bitset for each term instead. This is useful when you have different `terms` filters, which have lots of terms in common.

Also, there are `and/or` execution modes that perform a similar process, except the individual `term` filters are wrapped in an `and/or` filter instead of a `bool` filter.

Usually, the `and/or` approach is slower than `bool` because it doesn't take advantage of bitsets. `and/or` might be faster if the first `term` filters match only a few documents, which makes subsequent filters extremely fast.

To sum up, you have three options for running your filters:

- Caching them in the filter cache, which is great when filters are reused
- Not caching them if they aren't reused
- Running `terms` and `range` filters on field data, which is good when you have many terms, especially if the field data for that field is already loaded

Next, we'll look at the shard query cache, which is good for when you reuse entire search requests over static data.

10.3.2 *Shard query cache*

The filter cache is purpose-built to make parts of a search—namely filters that are configured to be cached—run faster. It's also segment-specific: if some segments get removed by the merge process, other segments' caches remain intact. By contrast, the shard query cache maintains a mapping between the whole request and its results on the shard level, as illustrated in figure 10.8. If a shard has already answered an identical request, it can serve it from the cache.

As of version 1.4, results cached at the shard level are limited to the total number of hits (not the hits themselves), aggregations, and suggestions. That's why (in version 1.5, at least) shard query cache works only when your query has `search_type` set to `count`.

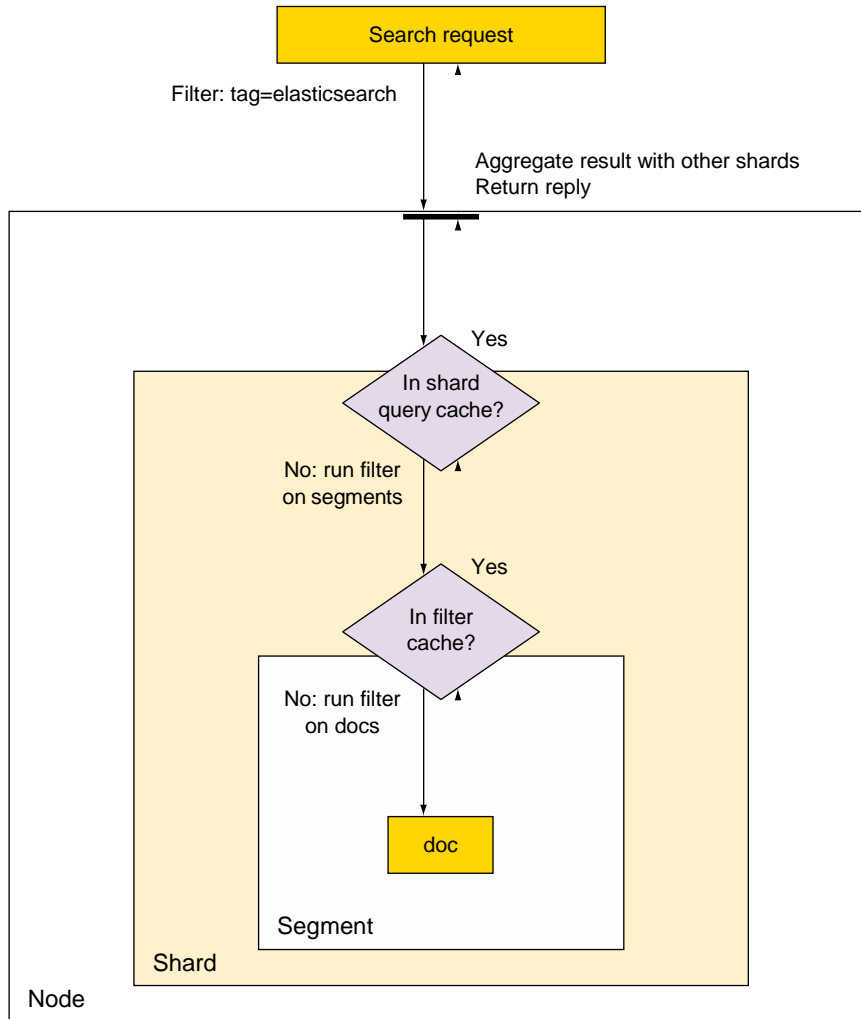


Figure 10.8 The shard query cache is more high-level than the filter cache.

NOTE By setting `search_type` to `count` in the URI parameters, you tell Elasticsearch that you're not interested in the query results, only in their number. We'll look at `count` and other search types later in this section. In Elasticsearch 2.0, setting `size` to 0 will also work and `search_type=count` will be deprecated.⁷

⁷ <https://github.com/elastic/elasticsearch/pull/9296>

The shard query cache entries differ from one request to another, so they apply only to a narrow set of requests. If you're searching for a different term or running a slightly different aggregation, it will be a cache miss. Also, when a refresh occurs and the shard's contents change, all shard query cache entries are invalidated. Otherwise, new matching documents could have been added to the index, and you'd get out-dated results from the cache.

This narrowness of cache entries makes the shard query cache valuable only when shards rarely change and you have many identical requests. For example, if you're indexing logs and have time-based indices, you may often run aggregations on older indices that typically remain unchanged until they're deleted. These older indices are ideal candidates for a shard query cache.

To enable the shard query cache by default on the index level, you can use the indices update settings API:

```
% curl -XPUT localhost:9200/get-together/_settings -d '{
  "index.cache.query.enable": true
}'
```

TIP As with all index settings, you can enable the shard query cache at index creation, but it makes sense to do that only if your new index gets queried a lot and updated rarely.

For every query, you can also enable or disable the shard query cache, overriding the index-level setting, by adding the `query_cache` parameter. For example, to cache the frequent `top_tags` aggregation on our `get-together` index, even if the default is disabled, you can run it like this:

```
% URL="localhost:9200/get-together/group/_search"
% curl "$URL?search_type=count&query_cache&pretty" -d '{
  "aggs": {
    "top_tags": {
      "terms": {
        "field": "tags.verbatim"
      }
    }
  }
}'
```

Like the filter cache, the shard query cache has a `size` configuration parameter. The limit can be changed at the node level by adjusting `indices.cache.query.size` from `elasticsearch.yml`, from the default of 1% of the JVM heap.

When sizing the JVM heap itself, you need to make sure you have enough room for both the filter and the shard query caches. If memory (especially the JVM heap) is limited, you should lower cache sizes to make more room for memory that's used anyway by index and search requests in order to avoid out-of-memory exceptions.

Also, you need to have enough free RAM besides the JVM heap to allow the operating system to cache indices stored on disk; otherwise you'll have a lot of disk seeks.

Next we'll look at how you can balance the JVM heap with the OS caches and why that matters.

10.3.3 JVM heap and OS caches

If Elasticsearch doesn't have enough heap to finish an operation, it throws an out-of-memory exception that effectively makes the node crash and fall out of the cluster. This puts an extra load on other nodes as they replicate and relocate shards in order to get back to the configured state. Because nodes are typically equal, this extra load is likely to make at least another node run out of memory. Such a domino effect can bring down your entire cluster.

When the JVM heap is tight, even if you don't see an out-of-memory error in the logs, the node may become just as unresponsive. This can happen because the lack of memory pressures the garbage collector (GC) to run longer and more often in order to free memory. As the GC takes more CPU time, there's less computing power on the node for serving requests or even answering pings from the master, causing the node to fall out of the cluster.

Too much GC? Let's search the web for some GC tuning tips!

When GC is taking a lot of CPU time, the engineer in us is tempted to find that magic JVM setting that will cure everything. More often than not, it's the wrong place to search for a solution because heavy GC is just a symptom of Elasticsearch needing more heap than it has.

Although increasing the heap size is an obvious solution, it's not always possible. The same applies to adding more data nodes. Instead, you can look at a number of tricks to reduce your heap usage:

- Reduce the index buffer size that we discussed in section 10.2.
- Reduce the filter cache and/or shard query cache.
- Reduce the `size` value of searches and aggregations (for aggregations, you also have to take care of `shard_size`).
- If you have to make do with large sizes, you can add some non-data and non-master nodes to act as clients. They'll take the hit of aggregating per-shard results of searches and aggregations.

Finally, Elasticsearch uses another cache type to work around the way Java does garbage collection. There's a young generation space where new objects are allocated. They're "promoted" to old generation if they're needed for long enough or if lots of new objects are allocated and the young space fills up. This last problem appears especially with aggregations, which have to iterate through large sets of documents and create lots of objects that might be reused with the next aggregation.

Normally you want these potentially reusable objects used by aggregations to be promoted to the old generation instead of some random temporary objects that just happen to be there when the young generation fills up. To achieve this, Elasticsearch

(continued)

implements a `PageCacheRecycler`⁸ where big arrays used by aggregations are kept from being garbage collected. This default page cache is 10% of the total heap, and in some cases it might be too much (for example, you have 30 GB of heap, making the cache a healthy 3 GB). You can control the size of this cache from `elasticsearch.yml` via `cache.recycler.page.limit.heap`.

Still, there are times when you'd need to tune your JVM settings (although the defaults are very good), such as when you have almost enough memory but the cluster has trouble when some rare but long GC pauses kick in. You have some options to make GC kick in more often but stop the world less, effectively trading overall throughput for better latency:

- Increase the survivor space (lower `-XX:SurvivorRatio`) or the whole young generation (lower `-XX:NewRatio`) compared to the overall heap. You can check if this is needed by monitoring different generations.⁹ More space should give more time for the young GC to clean up short-lived objects before they get promoted to the old generation, where a GC will stop the world for longer. But making these spaces too large will make the young GC work too hard and become inefficient, because longer-living objects have to be copied between the two survivor spaces
- Use the G1 GC (`-XX:+UseG1GC`), which will dynamically allocate space for different generations and is optimized for large-memory, low-latency use cases. It's not used as the default as of version 1.5 because there are still some bugs showing up¹⁰ on 32-bit machines, so make sure you test it thoroughly before using G1 in production.

CAN YOU HAVE TOO LARGE OF A HEAP?

It might have been obvious that a heap that's too small is bad, but having a heap that's too large isn't great either. A heap size of more than 32 GB will automatically make pointers uncompressed and waste memory. How much wasted memory? It depends on the use case: it can vary from as little as 1 GB for 32 GB if you're doing mostly aggregations (which use big arrays that have few pointers) to something like 10 GB if you're using filter caches a lot (which have many small entries with many pointers). If you really need more than 32 GB of heap, you're sometimes better off running two or more nodes on the same machine, each with less than 32 GB of heap, and dividing the data between them through sharding.

NOTE If you end up with multiple Elasticsearch nodes on the same physical machine, you need to make sure that two replicas of the same shard aren't allocated on the same physical machine under different Elasticsearch nodes. Otherwise, if a physical machine goes down, you'll lose two copies of that shard. To prevent this, you can use shard allocation, as described in lab 11.

⁸ <https://github.com/elastic/elasticsearch/issues/4557>

⁹ Sematext's SPM can do that for you, as described in appendix D.

¹⁰ <https://wiki.apache.org/lucene-java/JavaBugs>

Below 32 GB too much heap still isn't ideal (actually, at exactly 32 GB you already lose compressed pointers, so it's best to stick with 31 GB as a maximum). The RAM on your servers that isn't occupied by the JVM is typically used by the operating system to cache indices that are stored on the disk. This is especially important if you have magnetic or network storage because fetching data from the disk while running a query will delay its response. Even with fast SSDs, you'll get the best performance if the amount of data you need to store on a node can fit in its OS caches.

So far we've seen that a heap that's too small is bad because of GC and out-of-memory issues, and one that's too big is bad, too, because it diminishes OS caches. What's a good heap size, then?

IDEAL HEAP SIZE: FOLLOW THE HALF RULE

Without knowing anything about the actual heap usage for your use case, the rule of thumb is to allocate half of the node's RAM to Elasticsearch, but no more than 32 GB. This "half" rule often gives a good balance between heap size and OS caches.

If you can monitor the actual heap usage (and we'll show you how to do that in lab 11), a good heap size is just large enough to accommodate the regular usage plus any spikes you might expect. Memory usage spikes could happen—for example, if someone decides to run a `terms` aggregation with size 0 on an analyzed field with many unique terms. This will force Elasticsearch to load all terms in memory in order to count them. If you don't know what spikes to expect, the rule of thumb is again half: set a heap size 50% higher than your regular usage.

For OS caches, you depend mostly on the RAM of your servers. That being said, you can design your indices in a way that works best with your operating system's caching. For example, if you're indexing application logs, you can expect that most indexing and searching will involve recent data. With time-based indices, the latest index is more likely to fit in the OS cache than the whole dataset, making most operations faster. Searches on older data will often have to hit the disk, but users are more likely to expect and tolerate slow response times on these rare searches that span longer periods of time. In general, if you can put "hot" data in the same set of indices or shards by using time-based indices, user-based indices, or routing, you'll make better use of OS caches.

All the caches we discussed so far—filter caches, shard query caches, and OS caches—are typically built when a query first runs. Loading up the caches makes that first query slower, and the slowdown increases with the amount of data and the complexity of the query. If that slowdown becomes a problem, you can warm up the caches in advance by using index warmers, as you'll see next.

10.3.4 Keeping caches up with warmers

A *warmer* allows you to define any kind of search request: it can contain queries, filters, sort criteria, and aggregations. Once it's defined, the warmer will make Elasticsearch run the query with every refresh operation. This will slow down the refresh, but the user queries will always run on "warm" caches.

Warmers are useful when first-time queries are too slow and it's preferable for the refresh operation to take that hit rather than the user. If our get-together site example had millions of events and consistent search performance was important, warmers would be useful. Slower refreshes shouldn't concern you too much, because you expect groups and events to be searched for more often than they're modified.

To define a warmer on an existing index, you'd issue a `PUT` request to the index's URI, with `_warmer` as the type and the chosen warmer name as an `ID`, as shown in listing 10.7. You can have as many warmers as you want, but keep in mind that the more warmers you have, the slower your refreshes will be. Typically, you'd use a few popular queries as your warmers. For example, in the following listing, you'll put two warmers: one for upcoming events and one for popular group tags.

Listing 10.7 Two warmers for upcoming events and popular group tags

```
curl -XPUT 'localhost:9200/get-together/event/_warmer/upcoming_events' -d '{
  "sort": [ {
    "date": { "order": "desc" }
  } ]
}'
# {"acknowledged": true}
curl -XPUT 'localhost:9200/get-together/group/_warmer/top_tags' -d '{
  "aggs": {
    "top_tags": {
      "terms": {
        "field": "tags.verbatim"
      }
    }
  }
}'
# {"acknowledged": true}
```

Later on, you can get the list of warmers for an index by doing a `GET` request on the `_warmer` type:

```
curl localhost:9200/get-together/_warmer?pretty
```

You can also delete warmers by sending a `DELETE` request to the warmer's URI:

```
curl -XDELETE localhost:9200/get-together/_warmer/top_tags
```

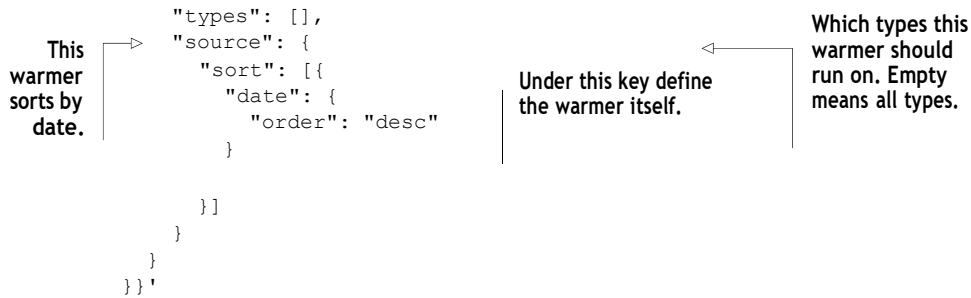
If you're using multiple indices, it makes sense to register warmers at index creation. To do that, define them under the `warmers` key in the same way you do with mappings and settings, as shown in the following listing.

Listing 10.8 Register warmer at index creation time

```
curl -XPUT 'localhost:9200/hot_index' -d '{
  "warmers": {
    "date_sorting": {
```

Name of this warmer. You can register multiple warmers, too.





TIP If new indices are created automatically, which might occur if you’re using time-based indices, you can define warmers in an index template that will be applied automatically to newly created indices. We’ll talk more about index templates in lab 11, which is all about how to administer your Elasticsearch cluster.

So far we’ve talked about general solutions: how to keep caches warm and efficient to make your searches fast, how to group requests to reduce network latency, and how to configure segment refreshing, flushing, and storing in order to make your indexing and searching fast. All of this also should reduce the load on your cluster.

Next we’ll talk about narrower best practices that apply to specific use cases, such making your scripts fast or doing deep paging efficiently.

10.4 Other performance tradeoffs

In previous sections, you might have noticed that to make an operation fast, you need to pay with something. For example, if you make indexing faster by refreshing less often, you pay with searches that may not “see” recently indexed data. In this section we’ll continue looking at such tradeoffs, especially those that occur in more specific use cases, by answering questions on the following topics:

- *Inexact matches*—Should you get faster searches by using ngrams and shingles at index time? Or is it better to use fuzzy and wildcard queries?
- *Scripts*—Should you trade some flexibility by calculating as much as possible at index time? If not, how can you squeeze more performance out of them?
- *Distributed search*—Should you trade some network round-trips for more accurate scoring?
- *Deep paging*—Is it worth trading memory to get page 100 faster?

By the time this lab ends, we’ll have answered all these questions and lots of others that will come up along the way. Let’s start with inexact matches.

10.4.1 Big indices or expensive searches

Recall from lab 4 that to get inexact matches—for example, to tolerate typos— you can use a number of queries:

- *Fuzzy query*—This query matches terms at a certain edit distance from the original. For example, omitting or adding an extra character would make a distance of 1.
- *Prefix query or filter*—These match terms starting with the sequence you provide.
- *Wildcards*—These allow you to use ? and * to substitute one or many characters. For example, "e*search" would match “elasticsearch.”

These queries offer lots of flexibility, but they’re also more expensive than simple queries, such as term queries. For an exact match, Elasticsearch has to find only one term in the term dictionary, whereas fuzzy, prefix, and wildcard queries have to find all terms matching the given pattern.

There’s also another solution for tolerating typos and other inexact matches: ngrams. Recall from lab 5 that ngrams generate tokens from each part of the word. If you use them at both index and query time, you’ll get similar functionality to a fuzzy query, as you can see in figure 10.9.

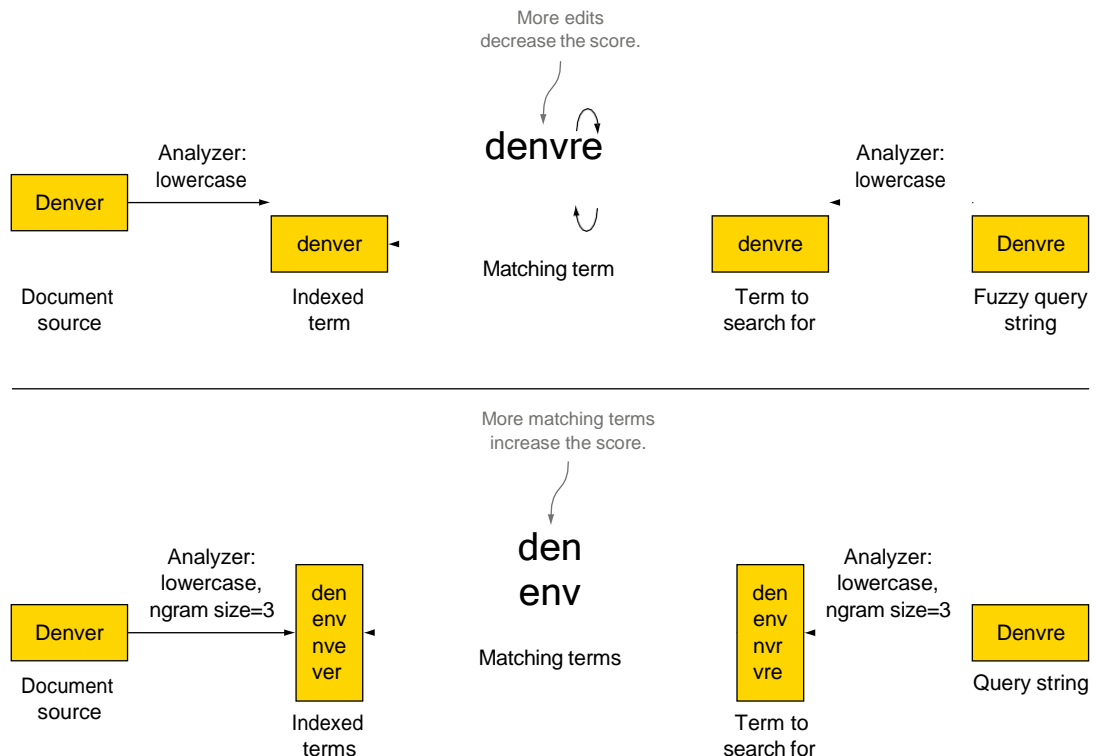


Figure 10.9 Ngrams generate more terms than you need with fuzzy queries, but they match exactly.

Which approach is best for performance? As with everything in this lab, there’s a tradeoff, and you need to choose where you want to pay the price:

- Fuzzy queries slow down your searches, but your index is the same as with exact matches.
- Ngrams, on the other hand, increase the size of your index. Depending on ngram and term sizes, the index size with ngrams can increase a few times. Also, if you want to change ngram settings, you have to re-index all data, so there’s less flexibility, but searches are typically faster overall with ngrams.

The ngram method is typically better when query latency is important or when you have lots of concurrent queries to support, so you need each one to take less CPU. Ngrams cause indices to be bigger, but they need to still fit in OS caches or you need fast disks—otherwise performance will degrade because your index is too big.

The fuzzy approach, on the other hand, is better when you need indexing throughput, where index size is an issue, or you have slow disks. Fuzzy queries also help if you need to change them often, such as by adjusting the edit distance, because you can make those changes without re-indexing all data.

PREFIX QUERIES AND EDGE NGRAMS

For inexact matches, you often assume that the beginning is right. For example, a search for “elastic” might be looking for “elasticsearch.” Like fuzzy queries, prefix queries are more expensive than regular term queries because there are more terms to look through.

The alternative could be to use edge ngrams, which were introduced in lab 5.

Figure 10.10 shows edge ngrams and prefix queries side by side.

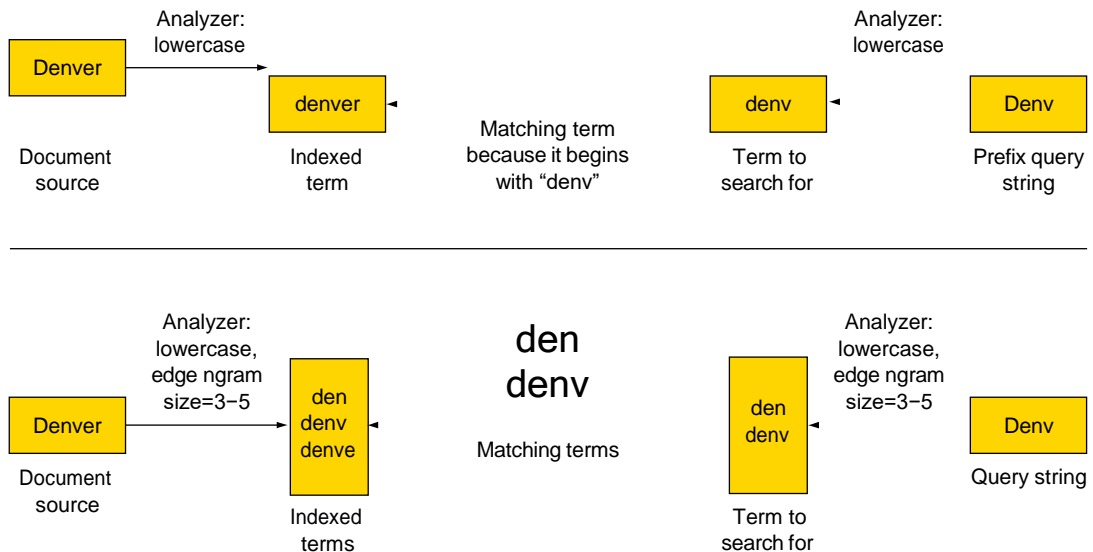


Figure 10.10 A prefix query has to match more terms but works with a smaller index than edge ngrams.

As with the fuzzy queries and ngrams, the tradeoff is between flexibility and index size, which are better in the prefix approach, and query latency and CPU usage, which are better for edge ngrams.

WILDCARDS

A wildcard query where you always put a wildcard at the end, such as `elastic*`, is equivalent in terms of functionality to a prefix query. In this case, you have the same alternative of using edge ngrams.

If the wildcard is in the middle, as with `e*search`, there's no real index-time equivalent. You can still use ngrams to match the provided letters *e* and *search*, but if you have no control over how wildcards are used, then the wildcard query is your only choice.

If the wildcard is always in the beginning, the wildcard query is typically more expensive than trailing wildcards because there's no prefix to hint in which part of the term dictionary to look for matching terms. In this case, the alternative can be to use the `reverse` token filter in combination with edge ngrams, as you saw in lab 5. This alternative is illustrated in figure 10.11.

PHRASE QUERIES AND SHINGLES

When you need to account for words that are next to each other, you can use the `match` query with `type` set to `phrase`, as you saw in lab 4. Phrase queries are slower because they have to account not only for the terms but also for their positions in the documents.

NOTE Positions are enabled by default for all analyzed fields because `index_options` is set to `positions`. If you don't use phrase queries, only term queries, you can disable indexing positions by setting `index_options` to `freqs`. If you don't care about scoring at all—for example, when you index application logs and you always sort results by timestamp—you can also skip indexing frequencies by setting `index_options` to `docs`.

The index-time alternative to phrase queries is to use shingles. As you saw in lab 5, shingles are like ngrams but for terms instead of characters. A text that was tokenized

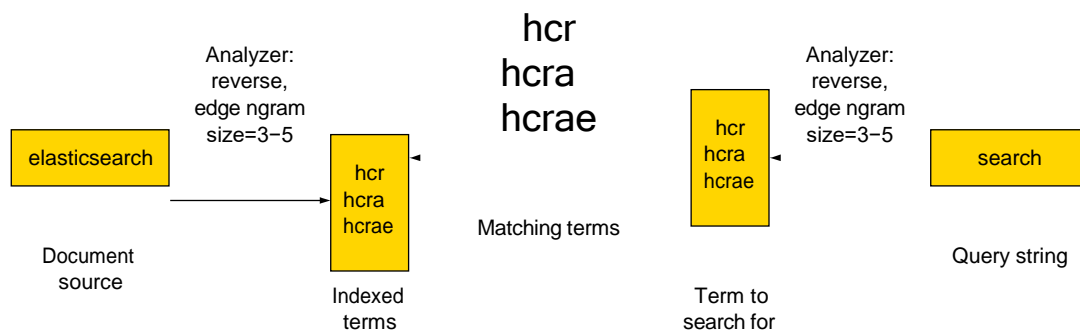


Figure 10.11 You can use the `reverse` and `edge ngram` token filters to match suffixes.

into `Introduction`, `to`, and `Elasticsearch` with a shingle size of 2 would produce the terms “`Introduction to`” and “`to Elasticsearch`.”

The resulting functionality is similar to phrase queries, and the performance implications are similar to the ngram situations we discussed earlier: shingles will increase the index size and slow down indexing in exchange for faster queries.

The two approaches are not exactly equivalent, in the same way wildcards and ngrams aren’t equivalent. With phrase queries, for example, you can specify a `slop`, which allows for other words to appear in your phrase. For example, a `slop` of 2 would allow a sequence like “`buy the best phone`” to match a query for “`buy phone`.” That works because at search time, Elasticsearch is aware of the position of each term, whereas shingles are effectively single terms.

The fact that shingles are single terms allows you to use them for better matching of compound words. For example, many people still refer to Elasticsearch as “`elastic search`,” which can be a tricky match. With shingles, you can solve this by using an empty string as a separator instead of the default white space, as shown in figure 10.12.

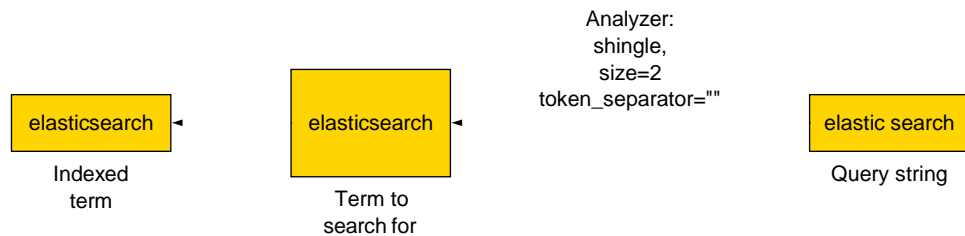


Figure 10.12 Using shingles to match compound words

As you’ve seen in our discussion of shingles, ngrams, and fuzzy and wildcard queries, there’s often more than one way to search your documents, but that doesn’t mean those ways are equivalent. Choosing the best one in terms of performance and flexibility depends a lot on your use case. Next we’ll look more deeply at scripts, where you’ll find more of the same: multiple ways to achieve the same result, but each method comes with its own advantages and disadvantages.

10.4.2 Tuning scripts or not using them at all

We first introduced scripts in lab 3 because they can be used for updates. You saw them again in lab 6, where you used them for sorting. In lab 7 you used scripts again, this time to build virtual fields at search time using script fields.

You get a lot of flexibility through scripting, but this flexibility has an important impact on performance. Results of a script are never cached because Elasticsearch doesn’t know what’s in the script. There can be something external, like a random number, that will make a document match now but not match for the next run.

There's no choice for Elasticsearch other than running the same script for all documents involved.

When used, scripts are often the most time- and CPU-consuming part of your searches. If you want to speed up your queries, a good starting point is to try skipping scripts altogether. If that's not possible, the general rule is to get as close to native code as you can to improve their performance.

How can you get rid of scripts or optimize them? The answer depends heavily on the exact use case, but we'll try to cover the best practices here.

AVOIDING THE USE OF SCRIPTS

If you're using scripts to generate script fields, as you did in lab 7, you can do this at index time. Instead of indexing documents directly and counting the number of group members in a script by looking at the array length, you can count the number of members in your indexing pipeline and add it to a new field. In figure 10.13, we compare the two approaches.

As with ngrams, this approach to doing the computation at index time works well if query latency is a higher priority than indexing throughput.

Besides precomputing, the general rule for performance optimization for scripting is to reuse as much of Elasticsearch's existing functionality as possible. Before using scripts, can you fulfill the requirements with the function score query that we discussed in lab 6? The function score query offers lots of ways to manipulate the score. Let's say you want to run a query for "elasticsearch" events, but you'll boost the score in the following ways, based on these assumptions:

- *Events happening soon are more relevant.* You'll make events' scores drop exponentially the farther in the future they are, up to 60 days.
- *Events with more attendees are more popular and more relevant.* You'll increase the score linearly the more attendees an event has.

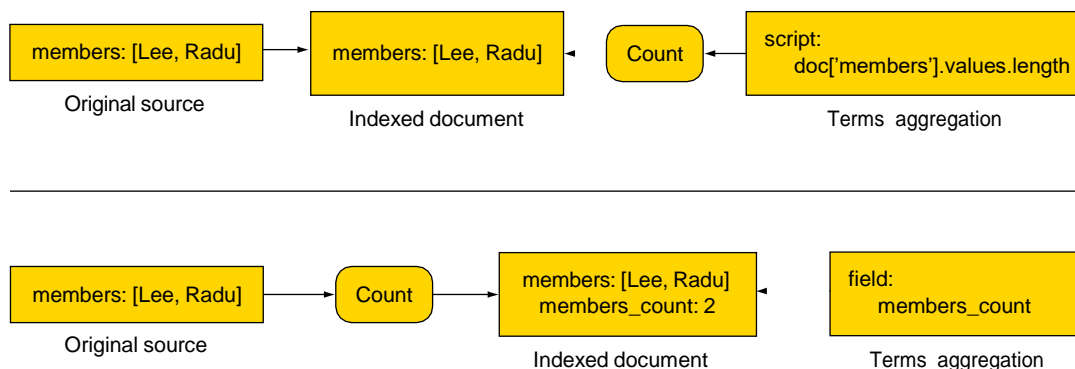


Figure 10.13 Counting members in a script or while indexing

If you calculate the number of event attendees at index time (name the field `attendees_count`), you can achieve both criteria without using any script:

```
"function_score": {
  "functions": [
    {
      "linear": {
        "date": {
          "origin": "2013-07-25T18:00",
          "scale": "60d"
        }
      }
    },
    {
      "field_value_factor": {
        "field": "attendees_count"
      }
    }
  ]
}
```

NATIVE SCRIPTS

If you want the best performance from a script, writing native scripts in Java is the best way to go. Such a native script would be an Elasticsearch plugin, and you can look in appendix B for a complete guide on how to write one.

The main disadvantage with native scripts is that they have to be stored on every node in Elasticsearch's classpath. Changing a script implies updating it on all the nodes of your cluster and restarting them. This won't be a problem if you don't have to change your queries often.

To run a native script in your query, set `lang` to `native` and the name of the script as the `script` content. For example, if you have a plugin with a script called `numberOfAttendees` that calculates the number of event attendees on the fly, you can use it in a `stats` aggregation like this:

```
"aggregations": {
  "attendees_stats": {
    "stats": {
      "script": "numberOfAttendees",
      "lang": "native"
    }
  }
}
```

LUCENE EXPRESSIONS

If you have to change scripts often or you want to be prepared to change them without restarting all your clusters, and your scripts work with numerical fields, Lucene expressions are likely to be the best choice.

With Lucene expressions, you provide a JavaScript expression in the script at query time, and Elasticsearch compiles it in native code, making it as quick as a native script.

The big limitation is that you have access only to indexed numeric fields. Also, if a document misses the field, the value of 0 is taken into account, which might skew results in some use cases.

To use Lucene expressions, you'd set `lang` to `expression` in your script. For example, you might have the number of attendees already, but you know that only half of them usually show up, so you want to calculate some stats based on that number:

```
"aggs": {
  "expected_attendees": {
    "stats": {
      "script": "doc['attendees_count'].value/2",
      "lang": "expression"
    }
  }
}
```

If you have to work with non-numeric or non-indexed fields and you want to be able to easily change scripts, you can use Groovy—the default language for scripting since Elasticsearch 1.4. Let's see how you can optimize Groovy scripts.

TERM STATISTICS

If you need to tune the score, you can access Lucene-level term statistics without having to calculate the score in the script itself—for example, if you only want to compute the score based on the number of times that term appears in the document. Unlike Elasticsearch's defaults, you don't care about the length of the field in that document or the number of times that term appears in other documents. To do that, you can have a script score that only specifies the term frequency (number of times the term appears in the document), as shown in the following listing.

Listing 10.9 Script score that only specifies term frequency

```
curl 'localhost:9200/get-together/event/_search?pretty' -d '{
  "query": {
    "function_score": {
      "filter": {
        "term": {
          "title": "elasticsearch"
        }
      },
      "functions": [
        {
          "script_score": {
            "script": "_index[\"title\"][_\"elasticsearch\"].tf() +
              _index[\"description\"][_\"elasticsearch\"].tf()",
            "lang": "groovy"
          }
        }
      ]
    }
  }
}
```

Filter all documents with the term "elasticsearch" in the title field.

Compute relevancy by looking at the term's frequency in the title and description fields.

Access term frequency via the `tf()` function belonging to the term, which belongs to the field.

ACCESSING FIELD DATA

If you need to work with the actual content of a document's fields in a script, one option is to use the `_source` field. For example, you'd get the `organizer` field by using `_source['organizer']`.

In lab 3, you saw how you can store individual fields instead of alongside `_source`. If an individual field is stored, you can access the stored content, too. For example, the same `organizer` field can be retrieved with `_fields['organizer']`.

The problem with `_source` and `_fields` is that going to the disk to fetch the field content of that particular field is expensive. Fortunately, this slowness is exactly what made field data necessary when Elasticsearch's built-in sorting and aggregations needed to access field content. Field data, as we discussed in lab 6, is tuned for random access, so it's best to use it in your scripts, too. It's often orders of magnitude faster than the `_source` or `_fields` equivalent, even if field data isn't already loaded for that field when the script is first run (or if you use doc values, as explained in chapter 6).

To access the `organizer` field via field data, you'd refer to `doc['organizer']`. For example, you can return groups where the organizer isn't a member, so you can ask them why they don't participate to their own groups:

```
% curl 'localhost:9200/get-together/group/_search?pretty' -d '{
  "query": {
    "filtered": {
      "filter": {
        "script": {
          "script": "return
doc.organizer.values.intersect(doc.members.values).isEmpty()",
        }
      }
    }
  }
}'
```

There's one caveat for using `doc['organizer']` instead of `_source['organizer']` or the `_fields` equivalent: you'll access the terms, not the original field of the document. If an organizer is 'Lee', and the field is analyzed with the default analyzer, you'll get 'Lee' from `_source` and 'lee' from `doc`. There are tradeoffs everywhere, but we assume you've gotten used to them at this point in the lab.

Next, we'll take a deeper look at how distributed searches work and how you can use search types to find a good balance between having accurate scores and low-latency searches.

10.4.3 Trading network trips for less data and better distributed scoring

Back in lab 2, you saw how when you hit an Elasticsearch node with a search request, that node distributes the request to all the shards that are involved and aggregates the individual shard replies into one final reply to return to the application.

Let's take a deeper look at how this works. The naïve approach would be to get N documents from all shards involved (where N is the value of `size`), sort them on the node that received the HTTP request (let's call it the coordinating node), pick the top N documents, and return them to the application. Let's say that you send a request with the default size of 10 to an index with the default number of 5 shards. This means that the coordinating node will fetch 10 whole documents from each shard, sort them, and return only the top 10 from those 50 documents. But what if there were 10 shards and 100 results? The network overhead of transferring the documents and the memory overhead of handling them on the coordinating node would explode, much like specifying large `shard_size` values for aggregations are bad for performance.

How about returning only the IDs of those 50 documents and the metadata needed for sorting to the coordinating node? After sorting, the coordinating node can fetch only the required top 10 documents from the shards. This would reduce the network overhead for most cases but will involve two round-trips.

With Elasticsearch, both options are available by setting the `search_type` parameter to the search. The naïve implementation of fetching all involved documents is `query_and_fetch`, whereas the two-trip method is called `query_then_fetch`, which is also the default. A comparison of the two is shown in figure 10.14.

The default `query_then_fetch` (shown on the right of the figure) gets better as you hit more shards, as you request more documents via the `size` parameter, and

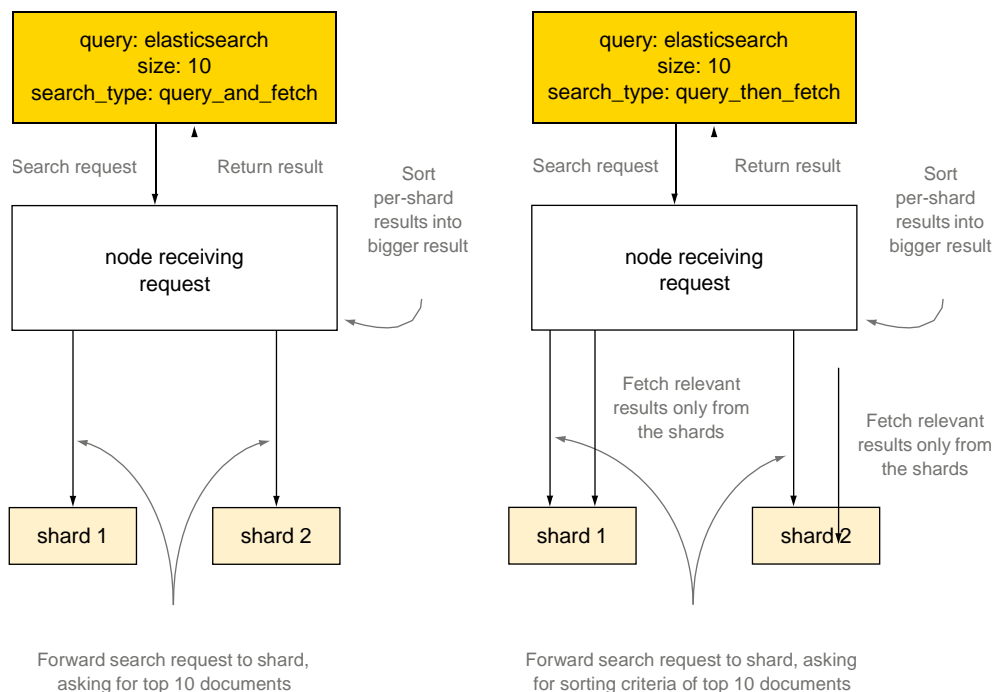


Figure 10.14 Comparison between `query_and_fetch` and `query_then_fetch`

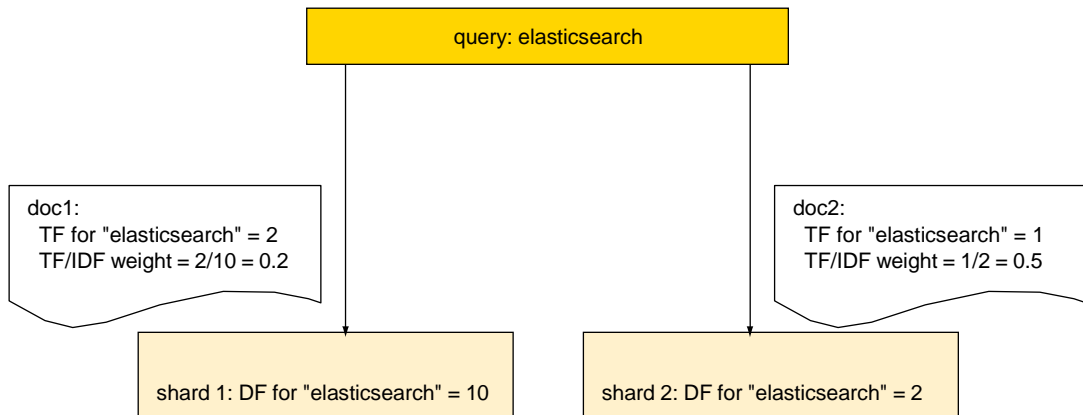


Figure 10.15 Uneven distribution of DF can lead to incorrect ranking.

as documents get bigger because it will transfer much less data over the network. `query_and_fetch` is only faster when you hit one shard—that’s why it’s used implicitly when you search a single shard, when you use routing, or when you only get the counts (we’ll discuss this later). Right now you can specify `query_and_fetch` explicitly, but in version 2.0 it will only be used internally for these specific use cases.¹¹

DISTRIBUTED SCORING

By default, scores are calculated per shard, which can lead to inaccuracies. For example, if you search for a term, one of the factors is the *document frequency* (DF), which shows how many times the term you search for appears in all documents. Those “all documents” are by default “all documents in this shard.” If the DF of a term is significantly different between shards, scoring might not reflect reality. You can see this in figure 10.15, where doc 2 gets a higher score than doc 1, even though doc 1 has more occurrences of “elasticsearch,” because there are fewer documents with that term in its shard.

You can imagine that with a high enough number of documents, DF values would naturally balance across shards, and the default behavior would work just fine. But if score accuracy is a priority or if DF is unbalanced for your use case (for example, if you’re using custom routing), you’ll need a different approach.

That approach could be to change the search type from `query_then_fetch` to `dfs_query_then_fetch`. The `dfs` part will tell the coordinating node to make an extra call to the shards in order to gather document frequencies of the searched terms. The aggregated frequencies will be used to calculate the score, as you can see in figure 10.16, ranking your doc 1 and doc 2 correctly.

¹¹ <https://github.com/elastic/elasticsearch/issues/9606>

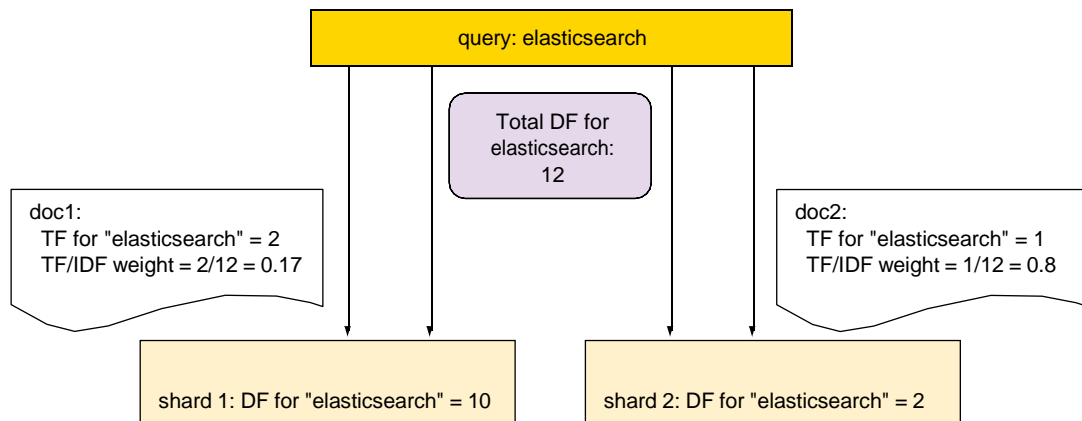


Figure 10.16 `dfs` search types use an extra network hop to compute global DFs, which are used for scoring.

You probably already figured out that DFS queries are slower because of the extra network call, so make sure that you actually get better scores before switching. If you have a low-latency network, this overhead can be negligible. If, on the other hand, your network isn't fast enough or you have high query concurrency, you may see a significant overhead.

RETURNING ONLY COUNTS

But what if you don't care about scoring at all and you don't need the document content, either? For example, you need only the document count or the aggregations. In such cases, the recommended search type is `count`. `count` asks the involved shards only for the number of documents that match and adds up those numbers.

TIP In version 2.0, adding `size=0` to a query will automatically do the same logic that `search_type=count` currently does, and `search_type=count` will be deprecated. More details can be found here: <https://github.com/elastic/elasticsearch/pull/9296>.

10.4.4 Trading memory for better deep paging

In lab 4, you learned that you'd use `size` and `from` to paginate the results of your query. For example, to search for "elasticsearch" in get-together events and get the fifth page of 100 results, you'd run a request like this:

```
% curl 'localhost:9200/get-together/event/_search?pretty' -d '{
  "query": {
    "match": {
      "title": "elasticsearch"
    }
  },
  "from": 400,
  "size": 100
}'
```

This will effectively fetch the top 500 results, sort them, and return only the last 100. You can imagine how inefficient this gets as you go deeper with pages. For example, if you change the mapping and want to re-index all existing data into a new index, you might not have enough memory to sort through all the results in order to return the last pages.

For this kind of scenario you can use the `scan` search type, as you'll do in listing 10.10, to go through all the get-together groups. The initial reply returns only the scroll ID, which uniquely identifies this request and will remember which pages were already returned. To start fetching results, send a request with that scroll ID. Repeat the same request to fetch the next page until you either have enough data or there are no more hits to return—in which case the `hits` array is empty.

Listing 10.10 Use `scan` search type

```
curl "localhost:9200/get-together/event/_search?pretty&q=elasticsearch\
&search_type=scan\
&scroll=1m\
&size=100"
# reply
{
  "_scroll_id":
    "c2Nhbjx0Zk2OjdZdkdQOTJLU1NpNGpxRW4S0RWUVE7MTt0b3RhbF9oaXRzOjc7",
  [...]
  "hits": {
    "total": 7,
    "max_score": 0,
    "hits": []
  }
}

curl 'localhost:9200/_search/scroll?scroll=1m&pretty' -d
  'c2Nhbjx0Zk2OjdZdkdQOTJLU1NpNGpxRW4S0RWUVE7MTt0b3RhbF9oaXRzOjc7'
# reply
{
  "_scroll_id" : "c2Nhbjx0Zk2OjdZdkdQOTJLU1NpNGpxRW4S0RWUVE7MTt0b3RhbF9oaXRzOjc7",
  [...]
  "hits" : {
    "total" : 7,
    "max_score" : 0.0,
    "hits" : [ {
      "_index" : "get-together",
      [...]
    }
  ]
}

curl 'localhost:9200/_search/scroll?scroll=1m&pretty' -d
  'c2Nhbjx0Zk2OjdZdkdQOTJLU1NpNGpxRW4S0RWUVE7MTt0b3RhbF9oaXRzOjc7'
```

← The size of each page

← Elasticsearch will wait one minute for the next request (see below).

← You don't get any results yet, just their number.

← You get back a scroll ID that you'll use in the next request.

← Fetch the first page with the scroll ID you got previously; specify a timeout for the next request.

← This time you get a page of results.

← You get another scroll ID to use for the next request.

← Continue getting pages by using the last scroll ID, until the hits array is empty again.

As with other searches, scan searches accept a `size` parameter to control the page size. But this time, the page size is calculated per shard, so the actual returned size would be `size` times the number of shards. The timeout given in the `scroll` parameter of each request is renewed each time you get a new page; that's why you can have a

different timeout with every new request.

NOTE It may be tempting to have big timeouts so that you're sure a scroll doesn't expire while you're processing it. The problem is that if a scroll is active and not used, it wastes resources, taking up some JVM heap to remember the current page and disk space taken by Lucene segments that can't be deleted by merges until the scroll is completed or expired.

The `scan` search type always returns results in the order in which it encounters them in the index, regardless of the sort criteria. If you need both deep paging and sorting, you can add a `scroll` parameter to a regular search request. Sending a GET request to the scroll ID will get the next page of results. This time, `size` works accurately, regardless of the number of shards. You also get the first page of results with the first request, just like you get with regular searches:

```
% curl 'localhost:9200/get-together/event/_search?pretty&scroll=1m' -d '{
  "query": {
    "match": {
      "title": "elasticsearch"
    }
  }
}'
```

From a performance perspective, adding `scroll` to a regular search is more expensive than using the `scan` search type because there's more information to keep in memory when results are sorted. That being said, deep paging is much more efficient than the default because Elasticsearch doesn't have to sort all previous pages to return the current page.

Scrolling is useful only when you know in advance that you want to do deep paging; it's not recommended for when you need only a few pages of results. As with everything in this lab, you pay a price for every performance improvement. In the case of scrolling, that price is to keep information about the current search in memory until the scroll expires or you have no more hits.

10.5 **Summary**

In this lab we looked at a number of optimizations you can do to increase the capacity and responsiveness of your cluster:

- Use the bulk API to combine multiple `index`, `create`, `update`, or `delete` operations in the same request.
- To combine multiple `get` or `search` requests, you can use the `multiget` or `multi-search` API, respectively.
- A flush operation commits in-memory Lucene segments to disk when the index buffer size is full, the transaction log is too large, or too much time has passed since the last flush.
- A refresh makes new segments—flushed or not—available for searching. During heavy indexing, it's best to lower the refresh rate or disable refresh altogether.

- The merge policy can be tuned for more or less segments. Fewer segments make searches faster, but merges take more CPU time. More segments make indexing faster by spending less time on merging, but searches will be slower.
- An `optimize` operation forces a merge, which works well for static indices that get lots of searches.
- Store throttling may limit indexing performance by making merges fall behind. Increase or remove the limits if you have fast I/O.
- Combine filters that use bitsets in a `bool` filter and filters that don't in `and/or/not` filters.
- Cache counts and aggregations in the shard query cache if you have static indices.
- Monitor JVM heap and leave enough headroom so you don't experience heavy garbage collection or out-of-memory errors, but leave some RAM for OS caches, too.
- Use index warmers if the first query is too slow and you don't mind slower indexing.
- If you have room for bigger indices, using ngrams and shingles instead of fuzzy, wildcard, or phrase queries should make your searches faster.
- You can often avoid using scripts by creating new fields with needed data in your documents before indexing them.
- Try to use Lucene expressions, term statistics, and field data in your scripts whenever they fit.
- If your scripts don't need to change often, look at appendix B to learn how to write a native script in an Elasticsearch plugin.
- Use `dfs_query_then_fetch` if you don't have balanced document frequencies between shards.
- Use the `count` search type if you don't need any hits and the `scan` search type if you need many.

Licensed to Ernesto Lee Lee <socrates73@gmail.com>