
Elasticsearch
LEARNING
VOYAGE

Table of Contents


1. Scaling out
LEARNING
VOYAGE

Scaling out

This lesson covers

- Adding nodes to your Elasticsearch cluster
- Master election in your Elasticsearch cluster
- Removing and decommissioning nodes
- Using the _cat API to understand your cluster
- Planning and scaling strategies
- Aliases and custom routing

Adding nodes to your Elasticsearch cluster

- Even if you don't end up in a situation at work like the one just described, during the course of your experimentation with Elasticsearch you'll eventually come to the point where you need to add more processing power to your Elasticsearch cluster.
- You need to be able to search and index data in your indices faster, with more parallelization; you've run out of disk space on your machine, or perhaps your Elasticsearch node is now running out of memory when performing queries against your data.

Adding nodes to your cluster

- The first step in creating an Elasticsearch cluster is to add another node (or nodes) to the single node to make it a cluster of nodes.

- Adding a node to your local development environment is as simple as extracting the Elasticsearch distribution to a separate directory, entering the directory, and running the bin/elasticsearch command, as the following code snippet shows.

Adding nodes to your cluster

```
% bin/elasticsearch

[in another terminal window or tab]
% mkdir elasticsearch2
% cd elasticsearch2
% tar zxf elasticsearch-1.5.0.tar.gz
% cd elasticsearch-1.5.0
% bin/elasticsearch
```

The originally running Elasticsearch node from chapter 2

The newly started Elasticsearch node

Listing 9.1 Getting cluster health for a two-node cluster

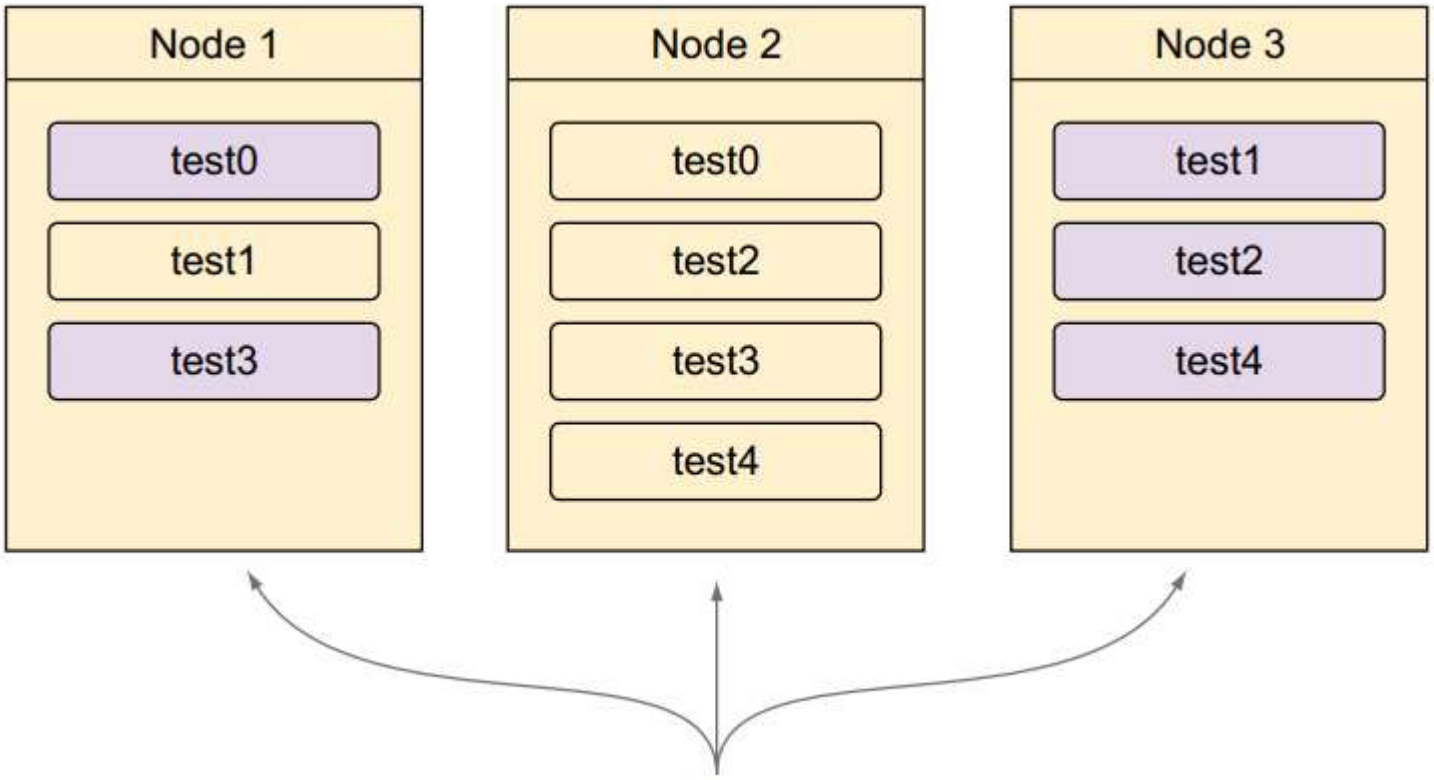
```
% curl -XGET 'http://localhost:9200/_cluster/health?pretty'
{
  "cluster_name" : "elasticsearch",
  "status" : "green",
  "timed_out" : false,
  "number_of_nodes" : 2,
  "number_of_data_nodes" : 2,
  "active_primary_shards" : 5,
  "active_shards" : 10,
  "relocating_shards" : 0,
  "initializing_shards" : 0,
  "unassigned_shards" : 0
}
```

The cluster is now green instead of yellow.

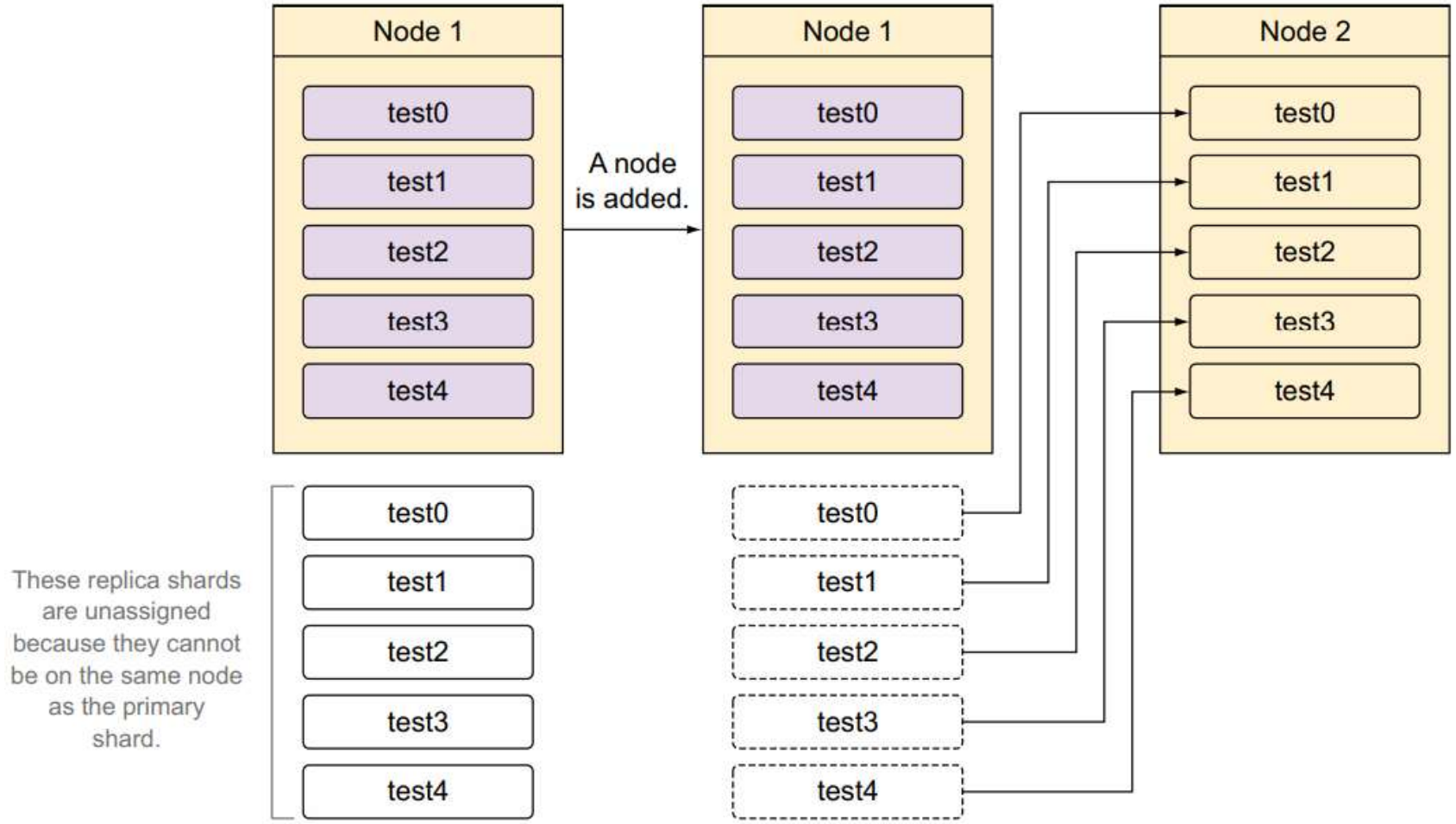
Two nodes that can handle data are now in the cluster.

All 10 shards are now active.

There are no longer any unassigned shards.



Elasticsearch has rebalanced the shards across all three nodes.


Node 1
test0
test1
test2
test3
test4
A node is added.
Node 1
test0
test1
test2
test3
test4
Node 2
test0
test1
test2
test3
test4
test0
test1
test2
test3
test4
test0
test1
test2
test3
test4
These replica shards are unassigned because they cannot be on the same node as the primary shard.

Adding nodes to your cluster

- If even more nodes are added to this cluster, Elasticsearch will try to balance the number of shards evenly across all nodes because each node added in this way shares the burden by taking a portion of the data (in the form of shards).

- Congratulations, you just horizontally scaled your Elasticsearch cluster!

Discovering other Elasticsearch nodes

- You might be wondering exactly how the second node you added to your cluster discovered the first node and automatically joined the cluster.
- Out of the box, Elasticsearch nodes can use two different ways to discover one another: multicast or unicast.
- Elasticsearch can use both at once but by default is configured to use only multicast because unicast requires a list of known nodes to connect to.

Multicast discovery

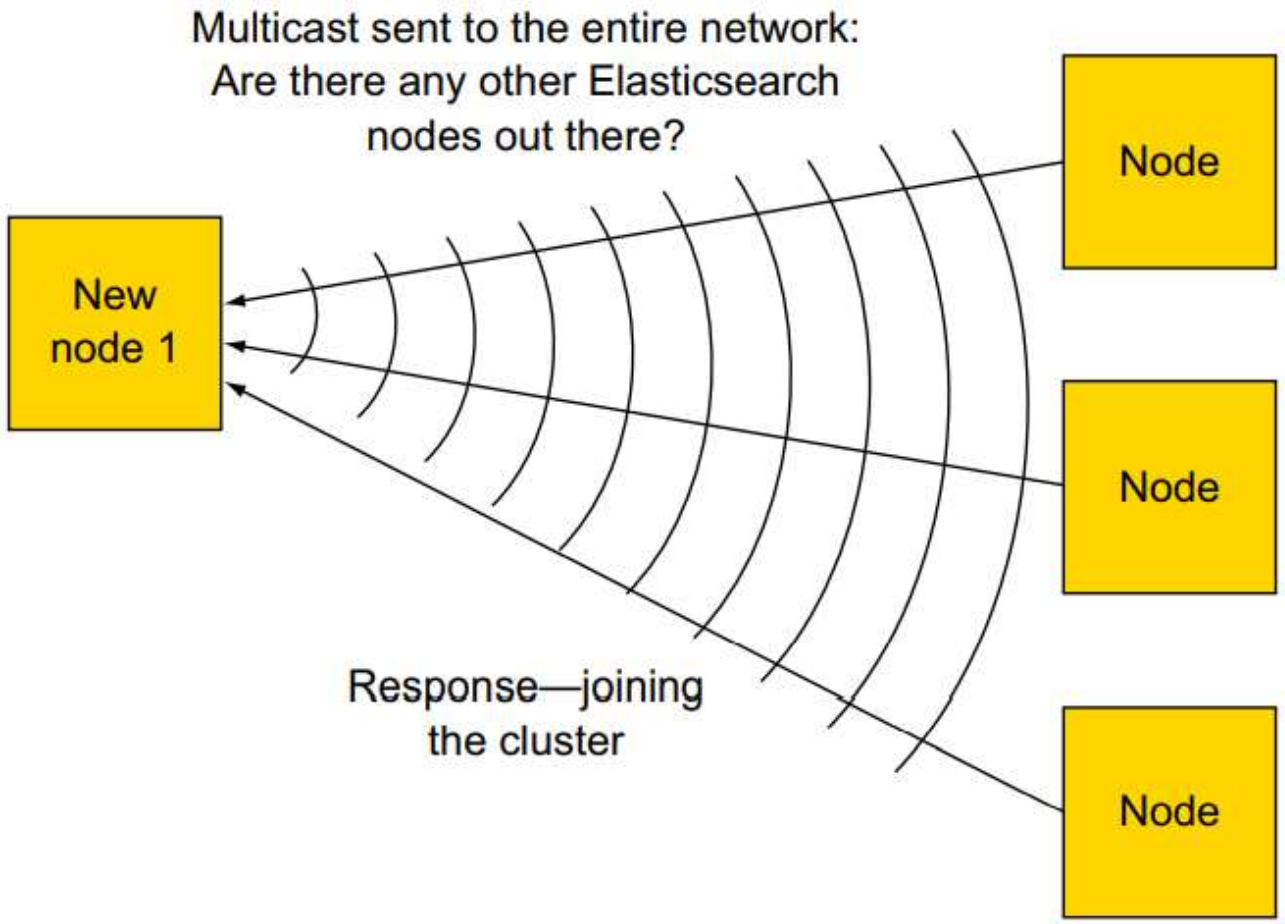
- When Elasticsearch starts up, it sends a multicast ping to the address 224.2.2.4 on port 54328, which in turn is responded to by other Elasticsearch nodes with the same cluster name.
- If you notice a coworker's local copy of Elasticsearch running and joining your cluster, make sure to change the cluster.name setting inside your elasticsearch.yml configuration file from the default elasticsearch to a more specific name.

Multicast discovery

- Multicast discovery has a few options that you can change or disable entirely by setting the following options in elasticsearch.yml, shown with their default values:

```
discovery.zen.ping.multicast:
  group: 224.2.2.4
  port: 54328
  ttl: 3
  address: null
  enabled: true
```

An address of null means to bind to all network interfaces.

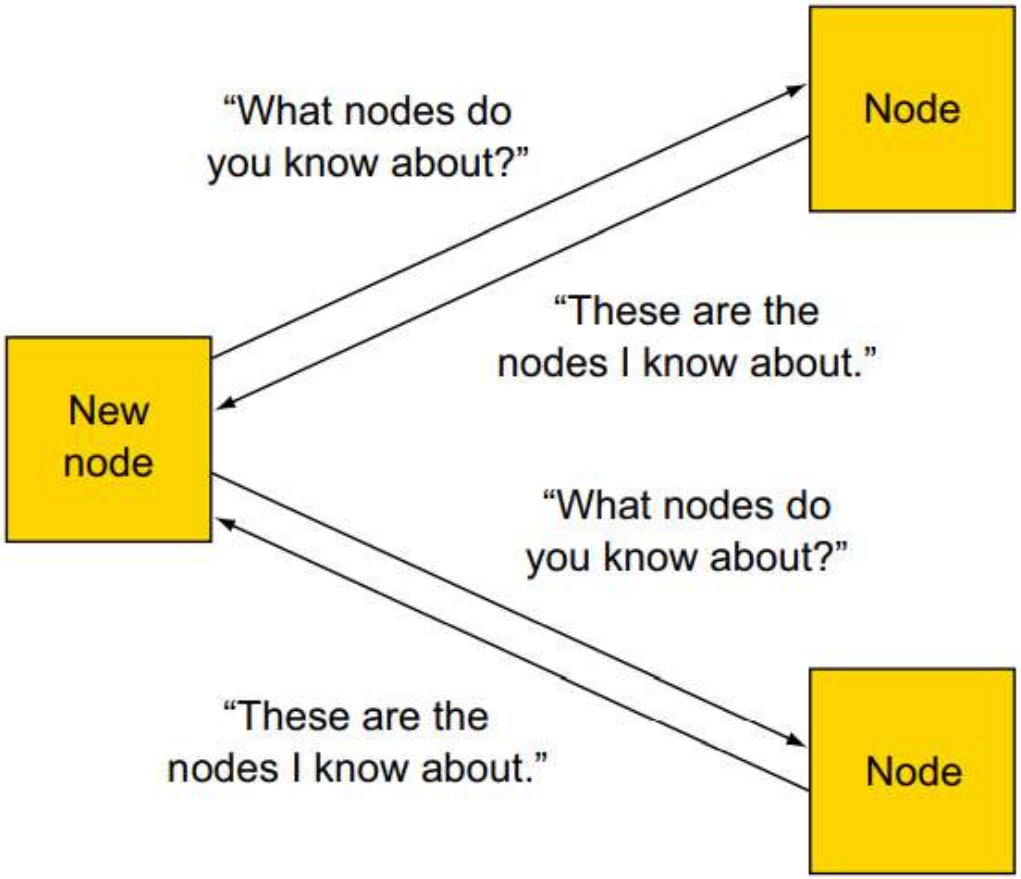


Elasticsearch using multicast discovery to discover other nodes in the cluster

Unicast discovery

- Unicast discovery uses a list of hosts for Elasticsearch to connect to and attempt to find more information about the cluster.

- This is ideal for cases where the IP address of the node won't change frequently or for production Elasticsearch systems where only certain nodes should be communicated with instead of the entire network.



Elasticsearch using unicast discovery to discover other nodes in the cluster

Electing a master node and detecting faults

- Once the nodes in your cluster have discovered each other, they'll negotiate who becomes the master.
- The master node is in charge of managing the state of the cluster—that is, the current settings and state of the shards, indices, and nodes in the cluster.
- After the master node has been elected, it sets up a system of internal pings to make sure each node stays alive and healthy while in the cluster; this is called fault detection.

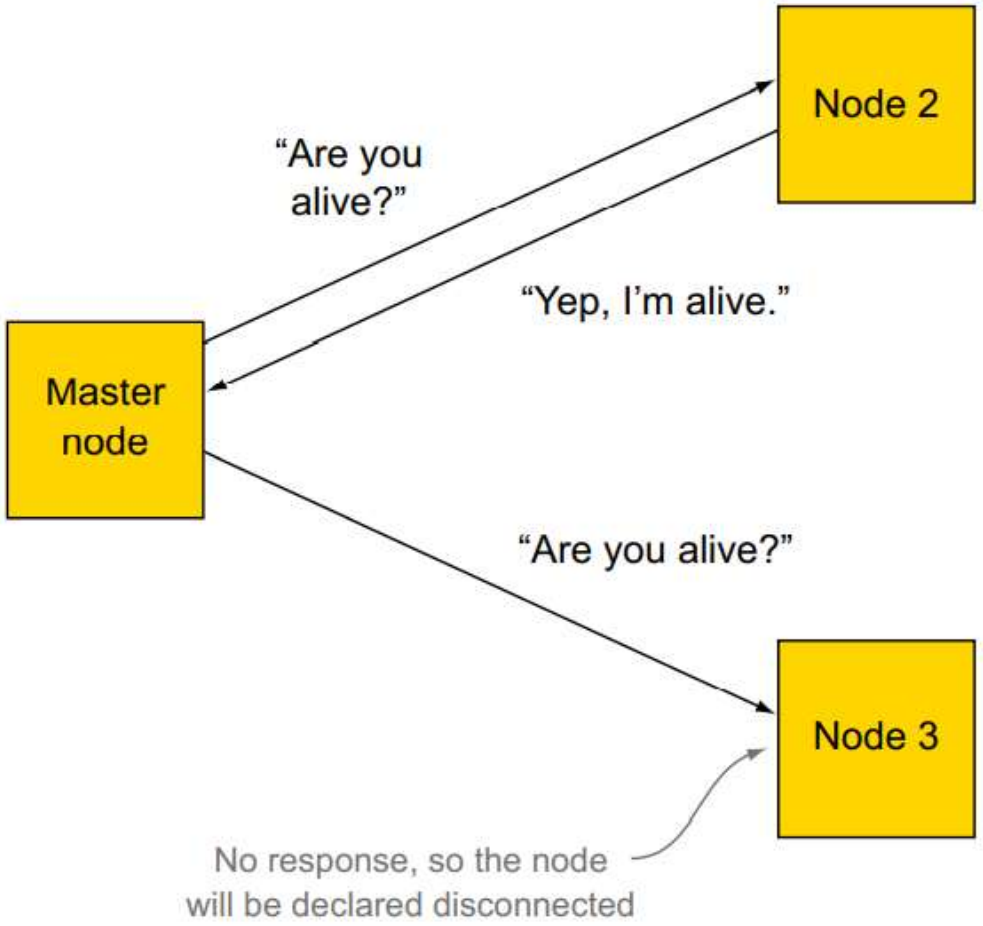
Listing 9.2 Getting information about nodes in the cluster with `curl`

```
% curl 'http://localhost:9200/_cluster/state/master_node,nodes?pretty'
{
  "cluster_name" : "elasticsearch",
  "master_node" : "5jDQs-LwRrqyrLm4DS_7wQ",
  "nodes" : {
    "5jDQs-LwRrqyrLm4DS_7wQ" : {
      "name" : "Kosmos",
      "transport_address" : "inet[/192.168.0.20:9300]",
      "attributes" : { }
    },
    "Rylg633AQmSnqbsPZwKqRQ" : {
      "name" : "Bolo",
      "transport_address" : "inet[/192.168.0.20:9301]",
      "attributes" : { }
    }
  }
}
```

The ID of the node currently elected as master

First node in the cluster

Second node in the cluster

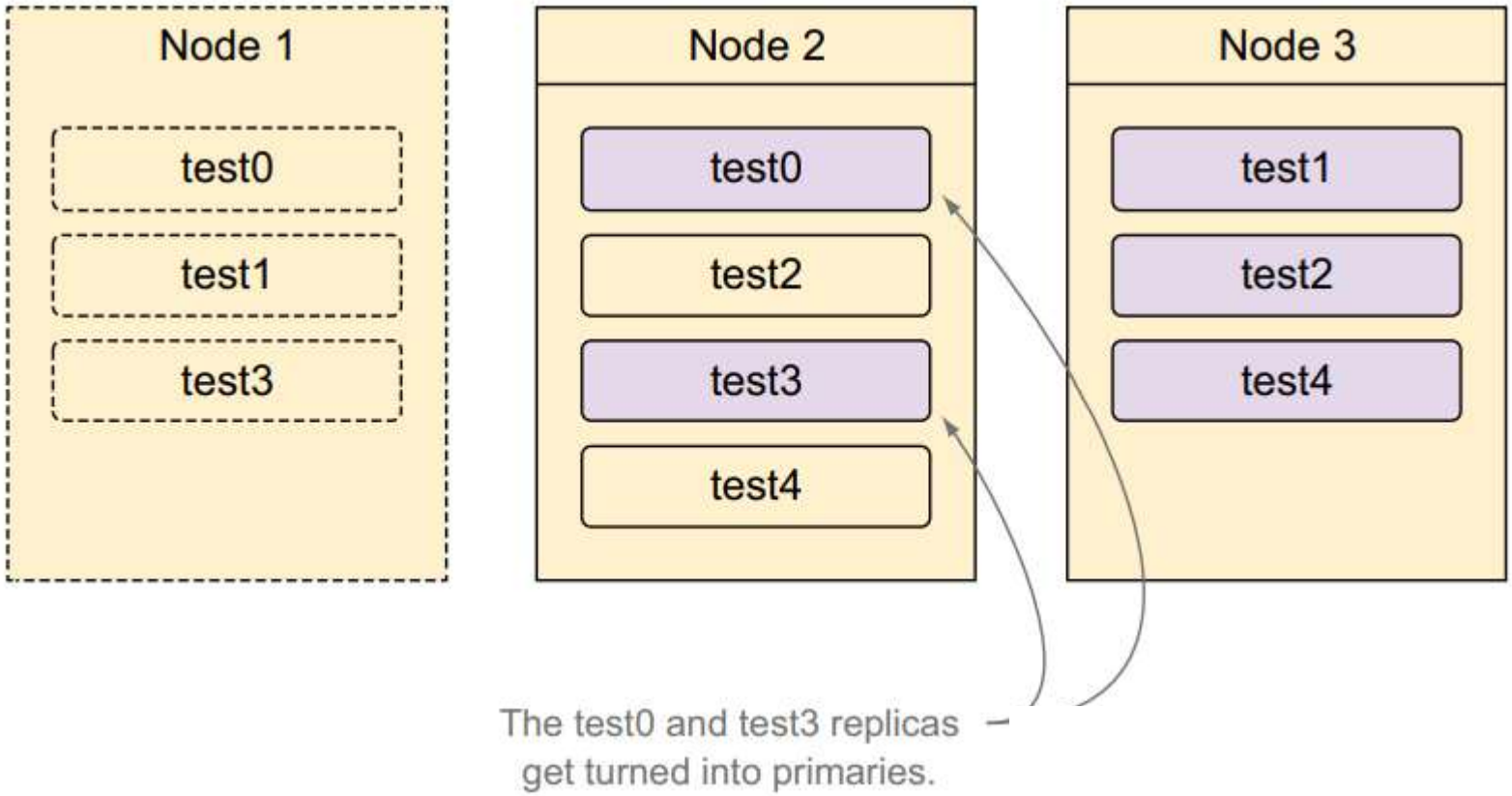


Cluster fault detection by the master node

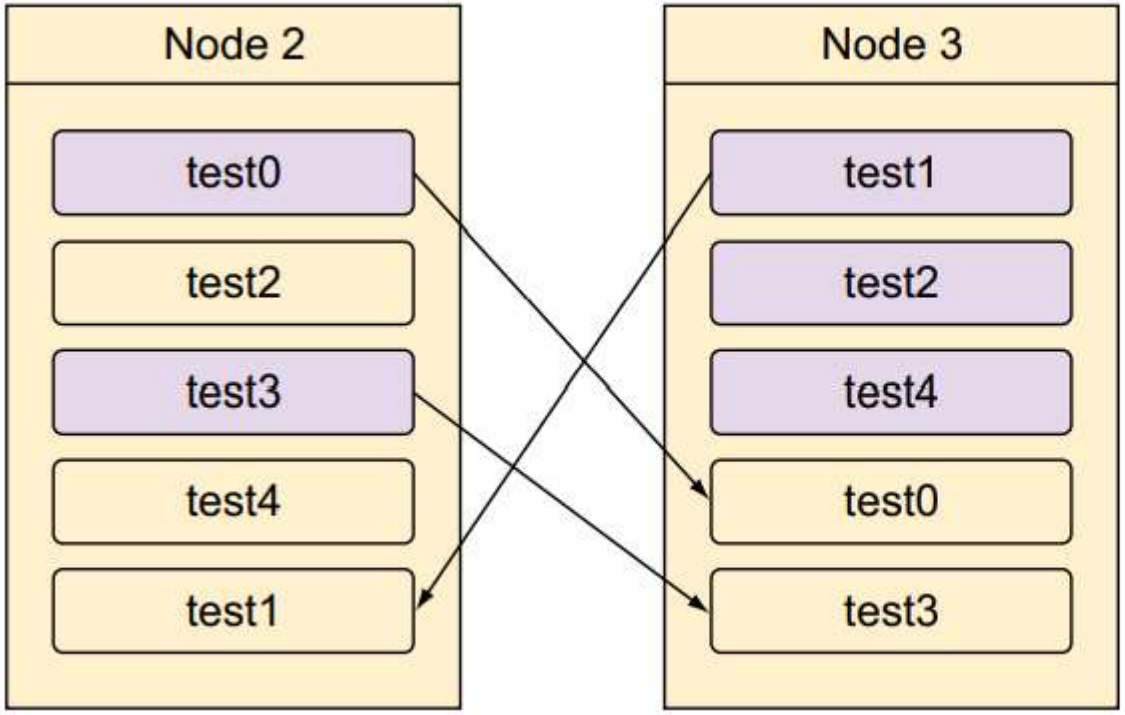
Removing nodes from a cluster

- Adding nodes is a great way to scale, but what happens when a node drops out of the Elasticsearch cluster or you stop the node?

- Use the three-node example cluster you created (2nd figure of this lesson), containing the test index with five primary shards and one replica spread across the three nodes.


Node 1
test0
test1
test3
Node 2
test0
test2
test3
test4
Node 3
test1
test2
test4
The test0 and test3 replicas get turned into primaries.

Removing nodes from a cluster



Re-creating replica shards after losing a node

Removing nodes from a cluster

- Once the replica shards have been re-created to account for the node loss, the cluster will be back in the green state with all primary and replica shards assigned to a node.

- Keep in mind that during this time the entire cluster will be available for searching and indexing because no data was actually lost.

Decommissioning nodes

- Having Elasticsearch automatically create new replicas when a node goes down is great, but when maintaining a cluster, you're eventually going to want to shut down a node that has data on it without the cluster going into a yellow state.

- Perhaps the hardware is degraded, or you aren't receiving the same number of requests you previously were and don't need to keep the node around.

Decommissioning nodes

- You decommission a node by making a temporary change to the cluster settings, as shown in the following listing.

Listing 9.3 Decommissioning a node in the cluster

```
curl -XPUT localhost:9200/_cluster/settings -d '{
    "transient" : {
        "cluster.routing.allocation.exclude._ip" : "192.168.1.10"
    }
}'
```

This setting is transient, meaning it won't persist through a cluster restart.

192.168.1.10 is the IP address of Node1.

Listing 9.4 Determining shard location from the cluster state

```
% curl -s 'localhost:9200/_nodes?pretty'
{
  "cluster_name" : "elasticsearch",
  "nodes" : {
    "lFd3ANXiQlug-0eJztvaeA" : {
      "name" : "Hayden, Alex",
      "transport_address" : "inet[/192.168.0.10:9300]",
      "ip": "192.168.0.10",
      "host" : "Perth",
      "version" : "1.5.0",
      "http_address" : "inet[/192.168.0.10:9200]"
    },
    "JGG7qQmBTB-LNfoz7VS97Q" : {
      "name" : "Magma",
      "transport_address" : "inet[/192.168.0.11:9300]",
      "ip": "192.168.0.10",
      "host" : "Xanadu",
      "version" : "1.5.0",
      "http_address" : "inet[/192.168.0.11:9200]"
    },
```

First retrieve the list of nodes in the cluster.

The unique ID of the node

IP address of the node that was decommissioned

```
    "McUL2T6vTSOGEAjSEuI-Zw" : {
      "name" : "Toad-In-Waiting",
      "transport_address" : "inet[/192.168.0.12:9300]",
      "ip": "192.168.0.10",
      "host" : "Corinth",
      "version" : "1.5.0",
      "http_address" : "inet[/192.168.0.12:9200]"
    }
  }
}

% curl 'localhost:9200/_cluster/state/routing_table,routing_nodes?pretty'
{
  "cluster_name" : "elasticsearch",
  "routing_table" : {
    "indices" : {
      "test" : {
        "shards" : {
          ...
        }
      }
    }
  },
```

Retrieving a filtered cluster state

Shortened to fit on this page

```
"routing_nodes" : {
  "unassigned" : [ ],
  "nodes" : {
    "JGG7qQmBTB-LNfoz7VS97Q" : [ {
      "state" : "STARTED",
      "primary" : true,
      "node" : "JGG7qQmBTB-LNfoz7VS97Q",
      "relocating_node" : null,
      "shard" : 0,
      "index" : "test"
    }, {
      "state" : "STARTED",
      "primary" : true,
      "node" : "JGG7qQmBTB-LNfoz7VS97Q",
      "relocating_node" : null,
      "shard" : 1,
      "index" : "test"
    }, {
      "state" : "STARTED",
      "primary" : true,
      "node" : "JGG7qQmBTB-LNfoz7VS97Q",
      "relocating_node" : null,
      "shard" : 2,
      "index" : "test"
    }, ...],
    "McUL2T6vTSOGEAjSEuI-Zw" : [ {
      "state" : "STARTED",
      "primary" : false,
      "node" : "McUL2T6vTSOGEAjSEuI-Zw",
      "relocating_node" : null,
      "shard" : 0,
      "index" : "test"
```

This key lists each node with the shards currently assigned to it.

```
      }, {
        "state" : "STARTED",
        "primary" : false,
        "node" : "McUL2T6vTSOGEAjSEuI-Zw",
        "relocating_node" : null,
        "shard" : 1,
        "index" : "test"
      }, {
        "state" : "STARTED",
        "primary" : false,
        "node" : "McUL2T6vTSOGEAjSEuI-Zw",
        "relocating_node" : null,
        "shard" : 2,
        "index" : "test"
      }, ...]
    }
  },
  "allocations" : [ ]
}.
```

Upgrading Elasticsearch nodes

- There comes a point with every installation of Elasticsearch when it's time to upgrade to the latest version.
- We recommend that you always run the latest version of Elasticsearch because there are always new features being added, as well as bugs being fixed.
- That said, depending on the constraints of your environment, upgrading may be more or less complex.

Performing a rolling restart

- A rolling restart is another way of restarting your cluster in order to upgrade a node or make a nondynamic configuration change without sacrificing the availability of your data.

- This can be particularly good for production deployments of Elasticsearch. Instead of shutting down the whole cluster at once, you shut nodes down one at a time.

Performing a rolling restart

1. Disable allocation for the cluster.
2. Shut down the node that will be upgraded.
3. Upgrade the node.
4. Start the upgraded node.
5. Wait until the upgraded node has joined the cluster.
6. Enable allocation for the cluster.
7. Wait for the cluster to return to a green state.

Performing a rolling restart

- Repeat this process for each node that needs to be upgraded.
- To disable allocation for the cluster, you can use the cluster settings API with the following settings:

```
curl -XPUT 'localhost:9200/_cluster/settings' -d '{
  "transient" : {
    "cluster.routing.allocation.enable" : "none"
  }
}'
```

Setting this to none means no shards can be allocated in the cluster.

Performing a rolling restart

- You can reenable allocation by setting the cluster.routing.allocation.enable setting to all instead of none, like this:

```
curl -XPUT 'localhost:9200/_cluster/settings' -d '{
  "transient" : {
    "cluster.routing.allocation.enable" : "all"
  }
}'
```

Setting this to all means all shards can be allocated, both primaries and replicas.

Minimizing recovery time for a restart

- You may notice that even with the disable and enable allocation steps, it can still take a while for the cluster to return to a green state when upgrading a single node.
- Unfortunately, this is because the replication that Elasticsearch uses is for each shard segment, rather than document-level.
- This means that the Elasticsearch node sending data to be replicated is saying, “Do you have segments_1?” If it doesn’t have the file or the file isn’t the same, the entire segment file is copied.

Using the _cat API

- Using the curl commands is a great way to see what's going on with your cluster, but sometimes it's helpful to see the output in a more readable format

- (if you don't believe us, try curling the http://localhost:9200/ _cluster/state URL on a large cluster and see how much information comes back!).

Listing 9.5 Using the _cat API to find cluster health and nodes

```
curl -XGET 'localhost:9200/_cluster/health?pretty'
{
  "cluster_name" : "elasticsearch",
  "status" : "green",
  "timed_out" : false,
  "number_of_nodes" : 2,
  "number_of_data_nodes" : 2,
  "active_primary_shards" : 5,
  "active_shards" : 10,
  "relocating_shards" : 0,
  "initializing_shards" : 0,
  "unassigned_shards" : 0
}

%  curl -XGET 'localhost:9200/_cat/health?v'
cluster       status node.total node.data shards pri relo init
unassignelasticsearch red          2         2     42  22    0    0        23
```

Checking cluster health using the cluster health API

Checking cluster health using the _cat API

```
% curl -XGET 'localhost:9200/_cluster/state/master_node,nodes&pretty'
{
  "cluster_name" : "elasticsearch",
  "master_node" : "5jDQs-LwRrqyrLm4DS_7wQ",
  "nodes" : {
    "5jDQs-LwRrqyrLm4DS_7wQ" : {
      "name" : "Kosmos",
      "transport_address" : "inet[/192.168.0.20:9300]",
      "attributes" : { }
    },
    "Rylg633AQmSnqbsPZwKqRQ" : {
      "name" : "Bolo",
      "transport_address" : "inet[/192.168.0.21:9300]",
      "attributes" : { }
    }
  }
}
```

Retrieving a list of nodes as well as which node is the master using the JSON API...

```
% curl -XGET 'localhost:9200/_cat/nodes?v'
host          heap.percent ram.percent load node.role master name
Xanadu.local             8          56 2.29 d         *      Bolo
Xanadu.local             4          56 2.29 d         m      Kosmos
```

...and doing it with the _cat API. The node with "m" in the master column is the master node.

Using the _cat API

Listing 9.6 Using the _cat API to show shard allocation

The allocation command lists the count of shards across each node.

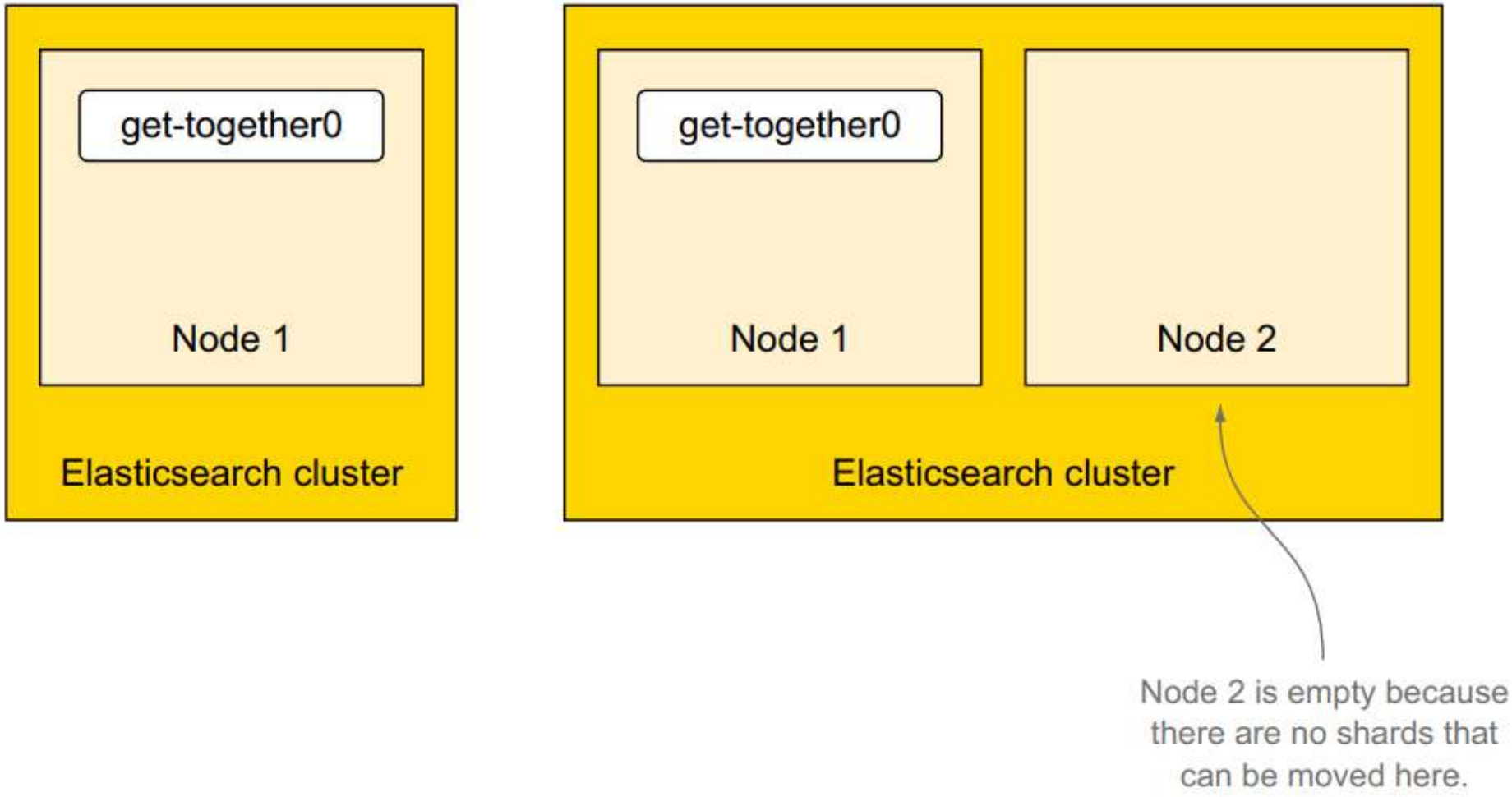
```
% curl -XGET 'localhost:9200/_cat/allocation?v'
shards disk.used disk.avail disk.total disk.percent host     ip     node
     2   196.5gb     36.1gb    232.6gb           84 Xanadu.local
     192.168.192.16 Molten Man
     2   196.5gb     36.1gb    232.6gb           84 Xanadu.local
     192.168.192.16 Grappler

% curl -XGET 'localhost:9200/_cat/shards?v'
index        shard prirep state   docs  store ip             node
get-together 0     p      STARTED   12 15.1kb 192.168.192.16 Molten Man
get-together 0     r      STARTED   12 15.1kb 192.168.192.16 Grappler
get-together 1     r      STARTED    8 11.4kb 192.168.192.16 Molten Man
get-together 1     p      STARTED    8 11.4kb 192.168.192.16 Grappler
```

Notice all the primary shards are on one node, the replicas on another.

Scaling strategies

- It might seem easy enough to add nodes to a cluster to increase the performance, but this is actually a case where a bit of planning goes a long way toward getting the best performance out of your cluster.

- Every use of Elasticsearch is different, so you'll have to pick the best options for your cluster based on how you'll index data, as well as how you'll search it.


get-together0
Node 1
Elasticsearch cluster
get-together0
Node 1
Node 2
Elasticsearch cluster
Node 2 is empty because there are no shards that can be moved here.

Over-sharding

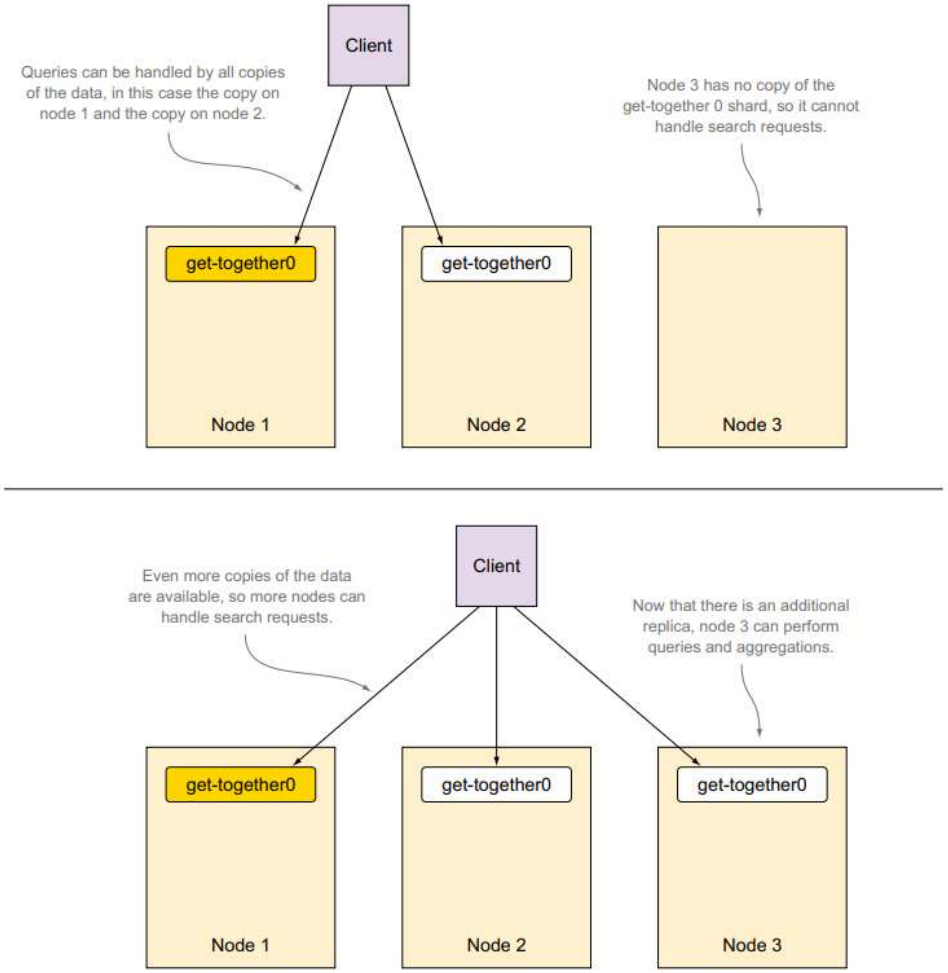
- Whoops! You've totally removed any benefit you get from adding nodes to the cluster.
- By adding another node, you're unable to scale because all of the indexing and querying load will still be handled by the node with the single shard on it.

Splitting data into indices and shards

- Unfortunately for now, there's no way to increase or decrease the number of primary shards in an index, but you could always plan your data to span multiple indices.
- This is another perfectly valid way to split data.
- Taking our get-together example, there's nothing stopping you from creating an index for every different city an event occurs in.

Maximizing throughput

- Maximizing throughput is one of those fuzzy, hazy terms that can mean an awful lot of things. Are you trying to maximize the indexing throughput? Make searches faster?
- Execute more searches at once? There are different ways to tweak Elasticsearch to accomplish each task.
- For example, if you received thousands of new groups and events, how would you go about indexing them as fast as possible?


Client
Queries can be handled by all copies of the data, in this case the copy on node 1 and the copy on node 2.
Node 3 has no copy of the get-together 0 shard, so it cannot handle search requests.
get-together0
get-together0
Node 1
Node 2
Node 3
Client
Even more copies of the data are available, so more nodes can handle search requests.
Now that there is an additional replica, node 3 can perform queries and aggregations.
get-together0
get-together0
get-together0
Node 1
Node 2
Node 3

Aliases

- Now let's talk about one of the easiest and potentially most useful features of Elasticsearch: aliases.
- Aliases are exactly what they sound like; they're a pointer or a name you can use that corresponds to one or more concrete indices.
- This turns out to be quite useful because of the flexibility it provides when scaling your cluster and managing how data is laid out across your indices.

What is an alias, really?

- You may be wondering what an alias is exactly and what kind of overhead is involved with Elasticsearch in creating one.
- An alias spends its life inside the cluster state, managed by the master node; this means that if you have an alias called idaho that points to an index named potatoes, the overhead is an extra key in the cluster state map that maps the name idaho to the concrete index potatoes.

WHY ARE ALIASES USEFUL?

- We recommend that everyone use an alias for their Elasticsearch indices because it will give a lot more flexibility in the future when it comes to re-indexing.

- Let's say that you start off by creating an index with a single primary shard and then later decide that you need more capacity on your index.

MANAGING ALIASES

Listing 9.7 Adding and removing aliases

```
curl -XPOST 'localhost:9200/_aliases' -d'
{
  "actions": [
    {
      "add" : {
        "index": "get-together",
        "alias": "gt-alias"
      }
    },
    {
      "remove": {
        "index": "old-get-together",
        "alias": "gt-alias"
      }
    }
  ]
}'
```

The operation—in this case, adding an index to an alias

The index get-together will be added to the alias gt-alias.

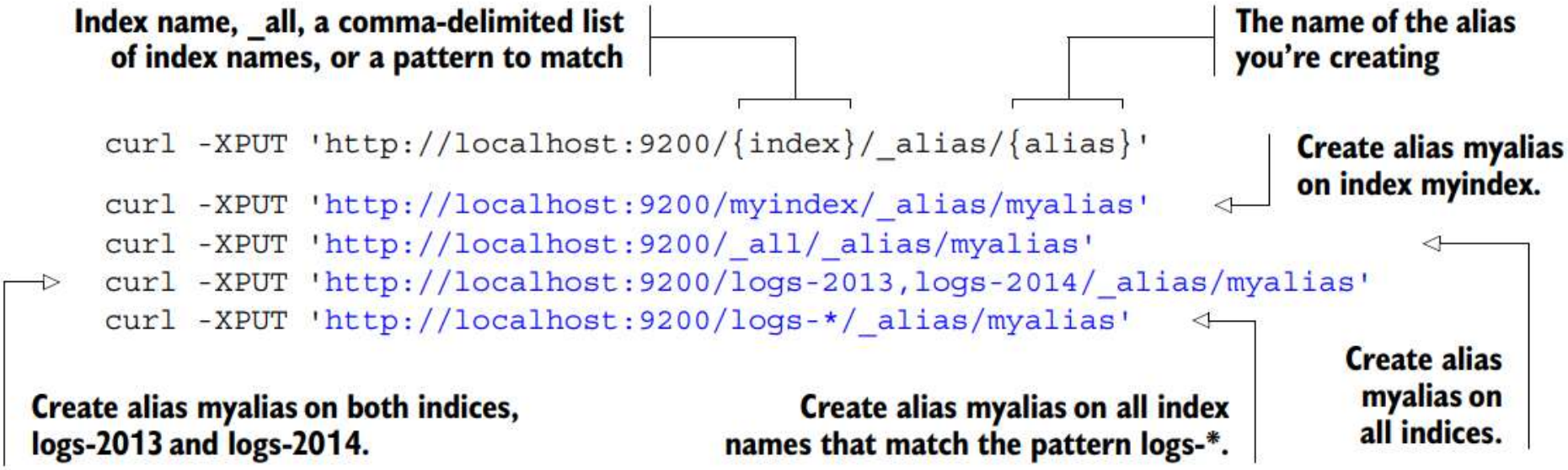
A remove operation to remove an index from an alias

The index old-get-together will be removed from the alias gt-alias.

Maximizing throughput

- In previous listing the get-together index is being added to an alias named gt-alias, and the made-up index old-get-together is being removed from the alias gt-alias.
- The act of adding an index to an alias creates it and removing all indices that an alias points to removes the alias; there's no manual alias creation and deletion.
- But the alias operations will fail if the index doesn't exist, so keep that in mind.

Alias creation



Alias creation

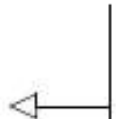
- Alias deletion accepts the same path parameter format:

```
curl -XDELETE 'localhost:9200/{index}/_alias/{alias}'
```

Alias creation

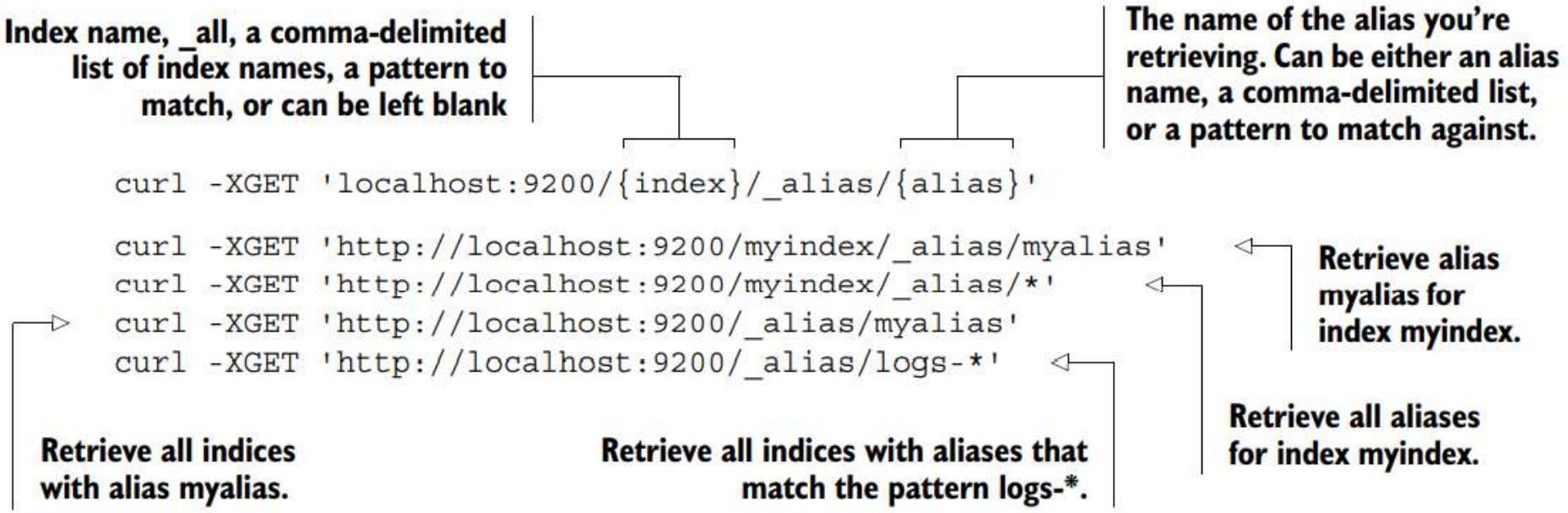
Listing 9.8 Retrieving the aliases pointing to a specific index

```
curl 'localhost:9200/get-together/_alias?pretty'
{
  "get-together" : {
    "aliases" : {
      "gt-alias" : { }
    }
  }
}
```



The gt-alias alias points to the get-together index.

Alias creation



MASKING DOCUMENTS WITH ALIAS FILTERS

- You can create an alias that does this filtering automatically, as shown in the following listing.

Listing 9.9 Creating a filtered alias

```
$ curl -XPOST 'localhost:9200/_aliases' -d'
{
  "actions": [
    {
      "add": {
        "index": "get-together",
```

```
        "alias": "es-groups",
        "filter": {
          "term": {"tags": "elasticsearch"}
        }
      }
    }
  ]
}'
{"acknowledged":true}

$ curl 'localhost:9200/get-together/group/_count' -d'
{
  "query": {
    "match_all": {}
  }
}'
{"count":5,"_shards":{"total":2,"successful":2,"failed":0}}

$ curl 'localhost:9200/es-groups/group/_count' -d'
{
  "query": {
    "match_all": {}
  }
}'
{"count":2,"_shards":{"total":2,"successful":2,"failed":0}}
```

Adding a filter for the es-groups alias for the elasticsearch tag

Counting all the groups in the get-together index

Five groups in the get-together index

Counting all the groups in the es-groups alias

Two groups in the es-groups alias; the results have been filtered automatically.

Routing

- We talked about how documents end up in a particular shard; this process is called routing the document.

- To refresh your memory, routing a document occurs when Elasticsearch hashes the ID of the document, either specified by you or generated by Elasticsearch, to determine which shard a document should be indexed into.

Routing

- Routing can also use a custom value for hashing, instead of the ID of the document.
- By specifying the routing query parameter on the URL, that value will be hashed and used instead of the ID:

```
curl -XPOST 'localhost:9200/get-together/group/9?routing=denver' -d'{
  "title": "Denver Knitting"
}'
```

Why use routing?

- If you don’t use routing at all, Elasticsearch will ensure that your documents are distributed in an even manner across all the different shards, so why would you want to use routing?

- Custom routing allows you to collect multiple documents sharing a routing value into a single shard, and once these documents are in the same index.

Routing strategies

- Routing is a strategy that takes effort in two areas: you'll need to pick good routing values while you're indexing documents, and you'll need to reuse those values when you perform queries.
- With our get-together example, you first need to decide on a good way to separate each document.
- In this case, pick the city that a get-together group or event happens to use as the routing value.

Routing strategies

Listing 9.10 Indexing documents with custom routing values

```
% curl -XPOST 'localhost:9200/get-together/group/10?routing=denver' -d'
{
  "name": "Denver Ruby",
  "description": "The Denver Ruby Meetup"
}'

% curl -XPOST 'localhost:9200/get-together/group/11?routing=boulder' -d'
{
  "name": "Boulder Ruby",
  "description": "Boulderites that use Ruby"
}'

% curl -XPOST 'localhost:9200/get-together/group/12?routing=amsterdam' -d'
{
  "name": "Amsterdam Devs that use Ruby",
  "description": "Mensen die genieten van het gebruik van Ruby"
}'
```

Indexing a document with a routing value of denver

Indexing a document with the routing value boulder

Listing 9.11 Specifying routing when querying

```
% curl -XPOST 'localhost:9200/get-together/group/
     _search?routing=denver,amsterdam' -d'
{
  "query": {
    "match": {
      "name": "ruby"
    }
  }
}'
{
    ...
    "hits": {
        "hits": [
            {
                "_id": "10",
                "_index": "get-together",
                "_score": 1.377483,
                "_source": {
                    "description": "The Denver Ruby Meetup",
                    "name": "Denver Ruby"
                },
                "_type": "group"
            },
            {
                "_id": "12",
                "_index": "get-together",
                "_score": 0.9642381,
                "_source": {
                    "description": "Mensen die genieten van het gebruik van
                    Ruby",
                    "name": "Amsterdam Devs that use Ruby"
                },
                "_type": "group"
            }
        ],
        "max_score": 1.377483,
        "total": 2
    }
}
```

Executing a query with a routing value of denver and amsterdam

Using the _search_shards API to determine where a search is performed

Listing 9.12 Using the _search_shards API with and without routing

Executing the search shards API without a routing value

```
% curl -XGET 'localhost:9200/get-together/_search_shards?pretty'
{
  "nodes" : {
    "aEFYkvsUQku4PTzNzTuuxw" : {
      "name" : "Captain Atlas",
      "transport_address" : "inet[/192.168.192.16:9300]"
    }
  },
  "shards" : [ [ {
    "state" : "STARTED",
    "primary" : true,
    "node" : "aEFYkvsUQku4PTzNzTuuxw",
    "relocating_node" : null,
    "shard" : 0,
    "index" : "get-together"
  } ], [ {
    "state" : "STARTED",
    "primary" : true,
    "node" : "aEFYkvsUQku4PTzNzTuuxw",
    "relocating_node" : null,
    "shard" : 1,
    "index" : "get-together"
  } ] ]
}

% curl -XGET 'localhost:9200/get-together/
    _search_shards?pretty&routing=denver'
```

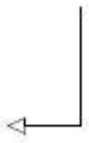
Nodes the request will be performed on

Both shard 0 and shard 1 will perform the request and return results.

Using the search shards API with the routing value denver

Routing strategies

```
{
  "nodes" : {
    "aEFYkvsUQku4PTzNzTuuxw" : {
      "name" : "Captain Atlas",
      "transport_address" : "inet[/192.168.192.16:9300]"
    }
  },
  "shards" : [ [ {
    "state" : "STARTED",
    "primary" : true,
    "node" : "aEFYkvsUQku4PTzNzTuuxw",
    "relocating_node" : null,
    "shard" : 1,
    "index" : "get-together"
  } ] ]
}
```



Only shard 1 will perform the request.

Listing 9.13 Defining routing as required in a type's mapping

```
% curl -XPOST 'localhost:9200/routed-events' -d'
{
  "mappings": {
    "event" : {
      "_routing" : {
        "required" : true
      },
      "properties": {
        "name": {
          "type": "string"
        }
      }
    }
  }
}'
{"acknowledged":true}

% curl -XPOST 'localhost:9200/routed-events/event/1' -d'
{"name": "my event"}'

  {"error":"RoutingMissingException[routing is required for [routed-events]/
  [event]/[1]]","status":400}
```

Creating an index named routed-events

Specifying that all documents for the event type require routing

Attempted indexing of a document without a routing value

Elasticsearch returns an error because required routing value is missing

Combining routing with aliases

- As you saw in the previous section, aliases are a powerful and flexible abstraction on top of indices.
- They can also be used with routing to automatically apply routing values when querying or when indexing, assuming the alias points to a single index.
- If you try to index into an alias that points to more than a single index.

Listing 9.14 Combining routing with an alias

```
% curl -XPOST 'localhost:9200/_aliases' -d'
{
  "actions" : [
    {
      "add" : {
        "index": "get-together",
        "alias": "denver-events",
        "filter": { "term": { "name": "denver" } },
        "routing": "denver"
      }
    }
  ]
}'
{"acknowledged":true}
```

Add an alias to the get-together index.

The alias will be called denver-events.

Filter results by documents whose names contain "denver".

Automatically use the routing value denver.

```
% curl -XPOST 'localhost:9200/denver-events/_search?pretty' -d'
{
  "query": {
    "match_all": {}
  },
  "fields": ["name"]
}'
{
  ...
  "hits" : {
    "total" : 3,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "get-together",
```

Query for all documents, using the denver-events alias.

```
      "_type" : "group",
      "_id" : "2",
      "_score" : 1.0,
      "fields" : {
        "name" : [ "Elasticsearch Denver" ]
      }
    }, {
      "_index" : "get-together",
      "_type" : "group",
      "_id" : "4",
      "_score" : 1.0,
      "fields" : {
        "name" : [ "Boulder/Denver big data get-together" ]
      }
    }, {
      "_index" : "get-together",
      "_type" : "group",
      "_id" : "10",
      "_score" : 1.0,
      "fields" : {
        "name" : [ "Denver Ruby" ]
      }
    } ]
  }
}
```

Summary

Things we talked about in this lesson:

- What happens when nodes are added to an Elasticsearch cluster
- How master nodes are elected
- Removing and decommissioning nodes
- Using the _cat API to understand your cluster
- Over-sharding and how it can be applied to plan for future growth of a cluster
- How to use aliases and routing for cluster flexibility and scaling

2. Improving performance

LEARNING VOYAGE

Improving performance

This lesson 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

Grouping requests

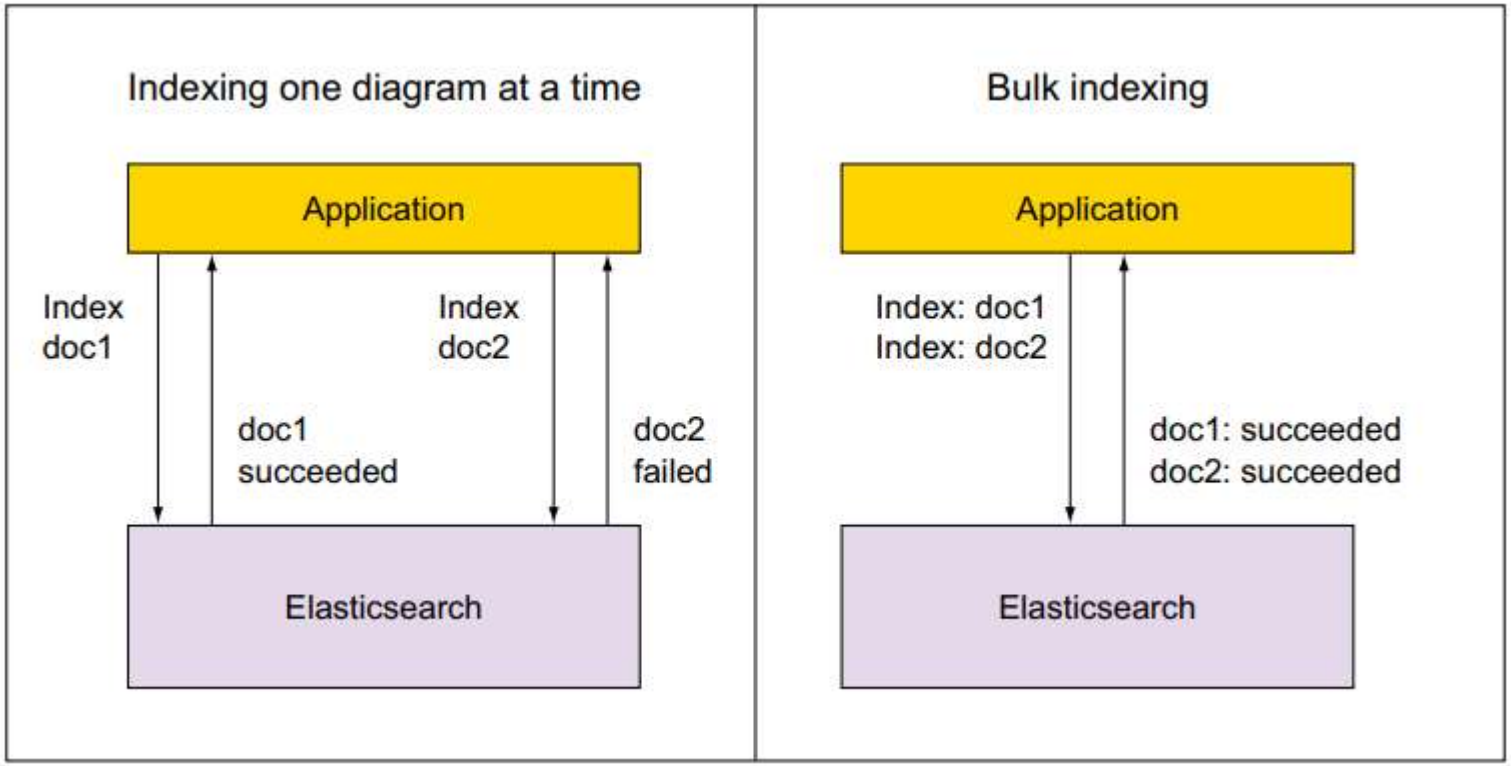
- The single best thing you can do for faster indexing is to send multiple documents to be indexed at once via the bulk API.
- This will save network round-trips and allow for more indexing throughput.
- A single bulk can accept any indexing operation; for example, you can create documents or overwrite them.

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.

Bulk indexing, updating, and deleting



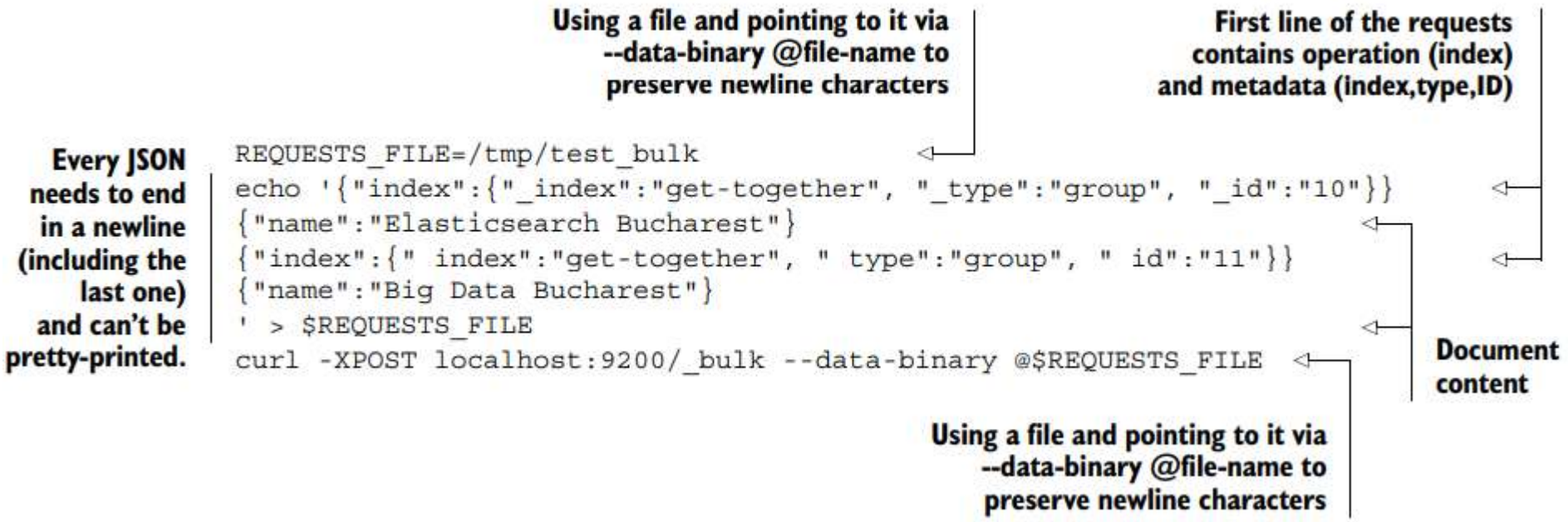
INDEXING IN BULKS

- Each indexing request is composed of two JSON documents separated by a newline: 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.

INDEXING IN BULKS

Listing 10.1 Indexing two documents in a single bulk



INDEXING IN BULKS

- _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
```

INDEXING IN BULKS

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

INDEXING IN BULKS

```
{
  "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
    }
  } ]
}
```

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 when you did individual updates.

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

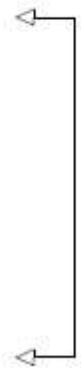
```
echo '{"index":{}}
{"title":"Elasticsearch Bucharest"}
{"create":{}}
{"title":"Big Data in Romania"}
{"update":{"_id": "11"}}
{"doc":{"created_on" : "2014-05-06"} }
{"delete":{"_id": "10"}}
' > $REQUESTS_FILE
URL='localhost:9200/get-together/group'
curl -XPOST $URL/_bulk?pretty --data-binary @$REQUESTS_FILE
# expected reply
  "took" : 37,
  "errors" : false,
  "items" : [ {
    "create" : {
      "_index" : "get-together",
      "_type" : "group",
      "_id" : "rVPtooieSxqfM6_JX-UCkg",
      "_version" : 1,
      "status" : 201
    }
  }, {
    "create" : {
      "_index" : "get-together",
      "_type" : "group",
      "_id" : "8w3GoNg5T_WEIL5jSTz_Ug",
      "_version" : 1,
      "status" : 201
    }
  }, {
```

Update operation: specify the ID and the partial document.

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

UPDATING OR DELETING IN BULKS

```
  "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.

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.

Listing 10.4 Multisearch request for events and groups about Elasticsearch

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.

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

```
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",
[…]
```

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.

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.

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"
        },
        {
            "_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 docs array identifies all documents that you want to retrieve.

The reply also contains a docs array.

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

MULTIGET

- 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" ]
}
```

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 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

Refresh and flush thresholds

- Recall 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.

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 runtime.
- 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"
}'
```

WHEN TO REFRESH

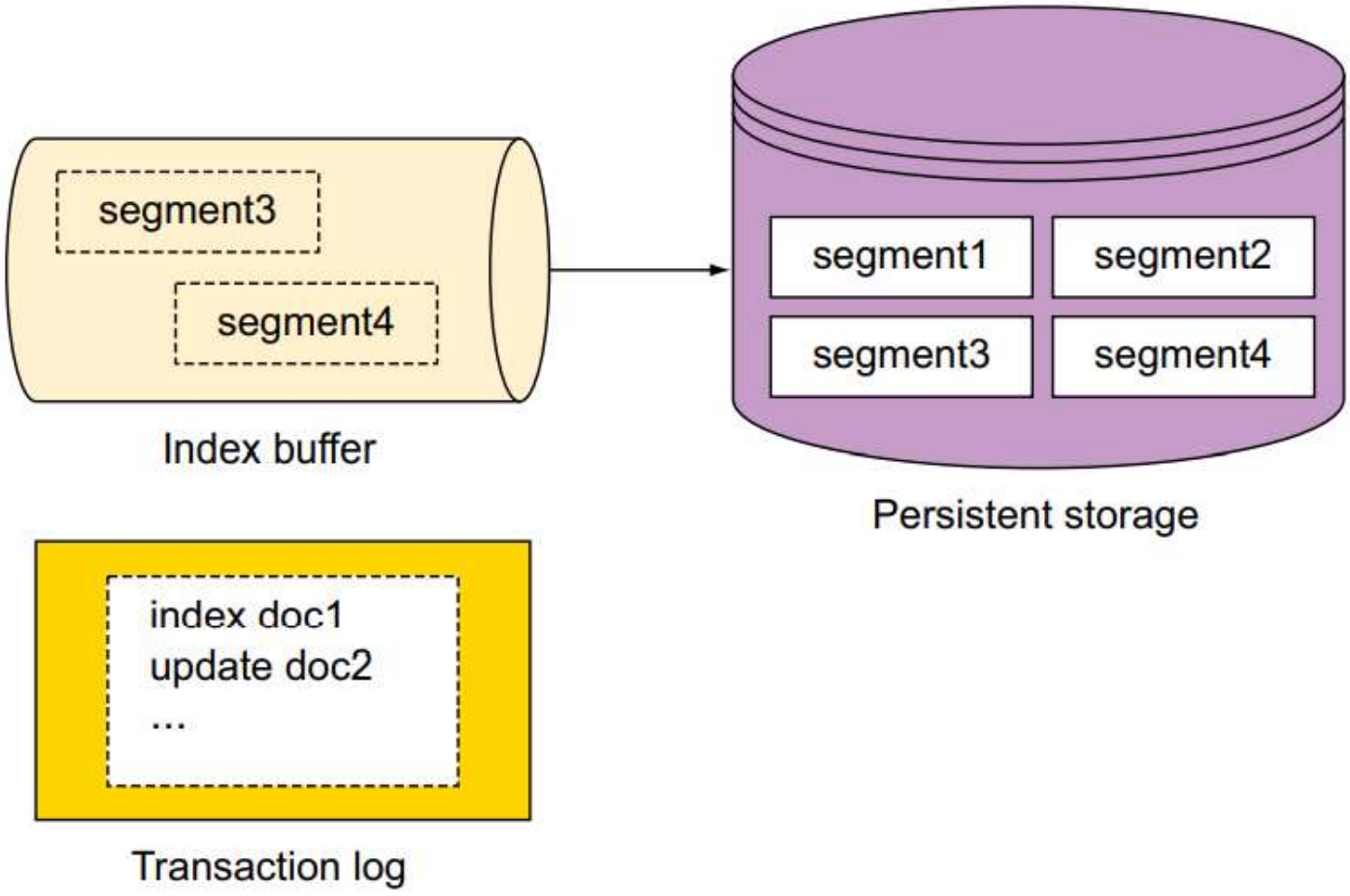
- To refresh manually, hit the _refresh endpoint of the index (or indices) you want to refresh:

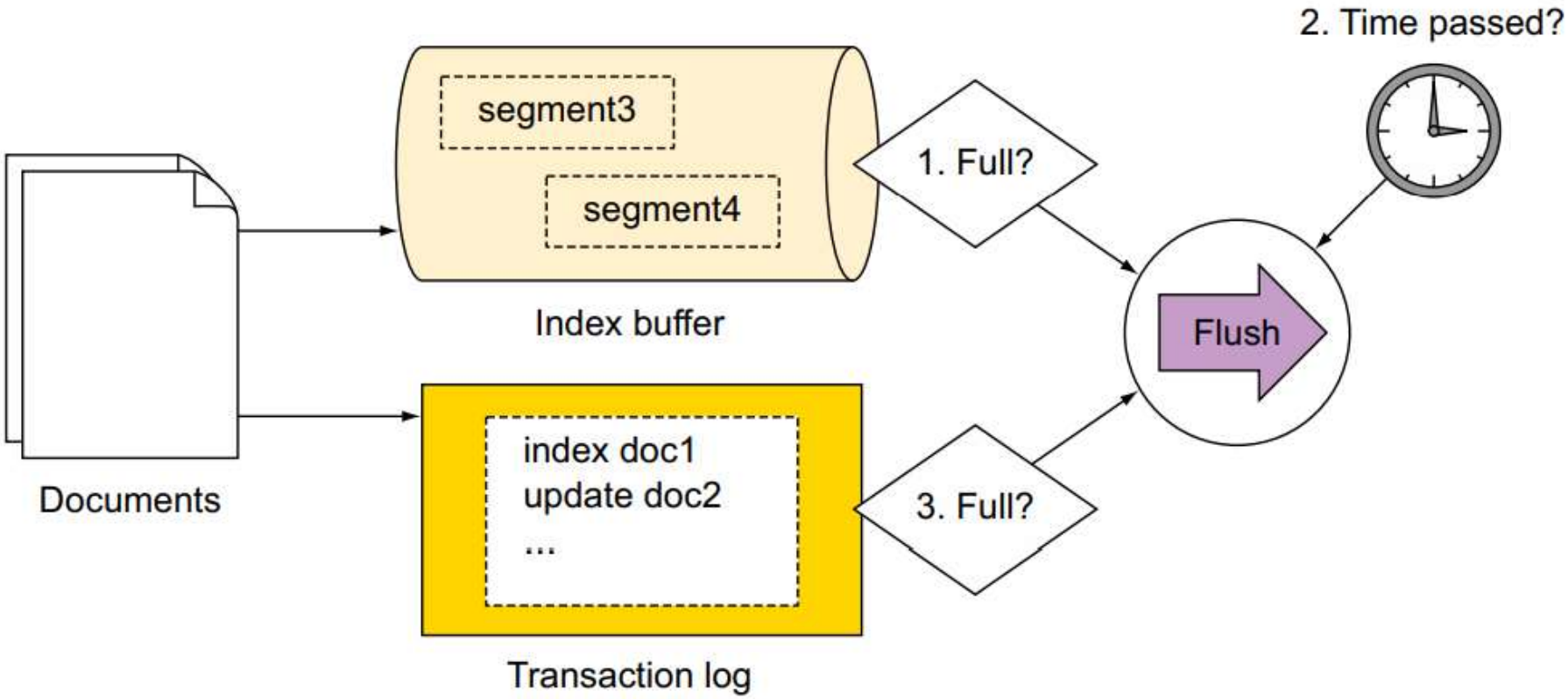
```
% curl localhost:9200/get-together/_refresh
```

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.


segment3
segment4
Index buffer
segment1
segment2
segment3
segment4
Persistent storage
index doc1
update doc2
...
Transaction log


2. Time passed?
segment3
segment4
1. Full?
Index buffer
Flush
Documents
index doc1
update doc2
...
3. Full?
Transaction log

WHEN TO FLUSH

- 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"
  }
}'
```

Merges and merge policies

- We first introduced segments 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.

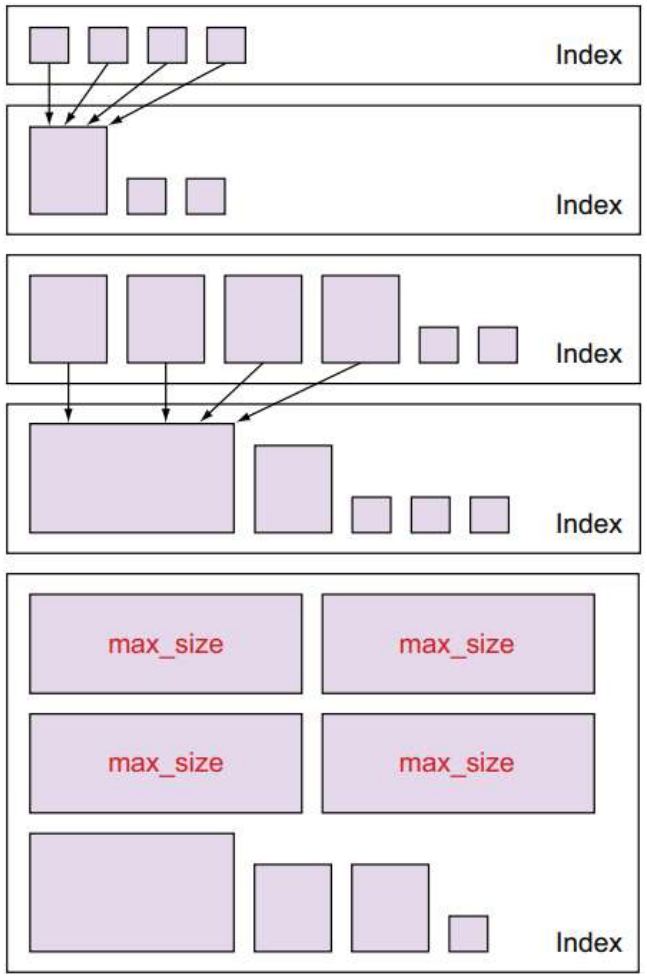
1. Flush operations add segments in the first tier, until there are too many. Let's say four are too many.

2. Small segments are merged into bigger ones. Flushing continues to add new small segments.

3. Eventually, there will be four segments on the bigger tier.

4. The four bigger segments get merged into an even bigger segment, and the process continues...

5. ...until a tier hits a set limit. Only smaller segments get merged; max segments stay the same.



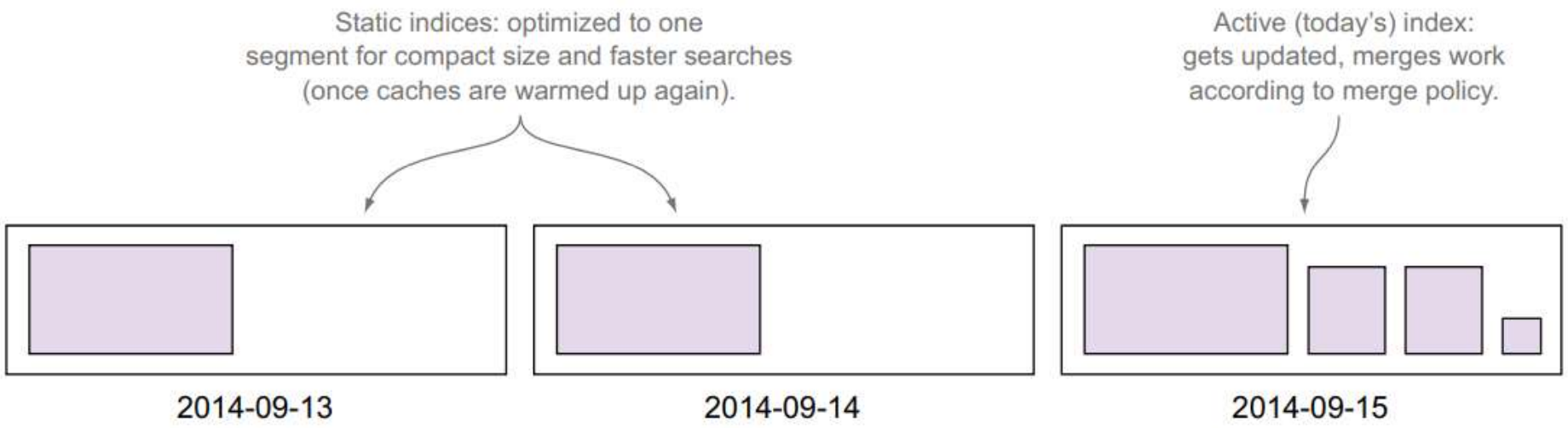
TUNING MERGE POLICY OPTIONS

- 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.

TUNING MERGE POLICY OPTIONS

```
% 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



OPTIMIZING INDICES

- 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
```

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.

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.

CHANGING STORE THROTTLING LIMITS

- 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:

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

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.

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.

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.

NIOFSDIRECTORY

- 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"
}'
```

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.

Store and store throttling

- Filters and filter caches 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.
- We said that of the two, we recommend using the filter, because it's cacheable.

Store and store throttling

- For 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
        }
      }
    }
  }
}'
```

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%

FILTER CACHE

- 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"
}'
```

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.

COMBINING FILTERS

- Table shows which of the important filters use bitsets and which don't.

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

COMBINING FILTERS

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

```
"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
          }
        }
      }
  ]
}
```

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
              }
            }
          }
        ]
      }
    }
  }
}'
```

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

The AND filter will run the bool filter first.

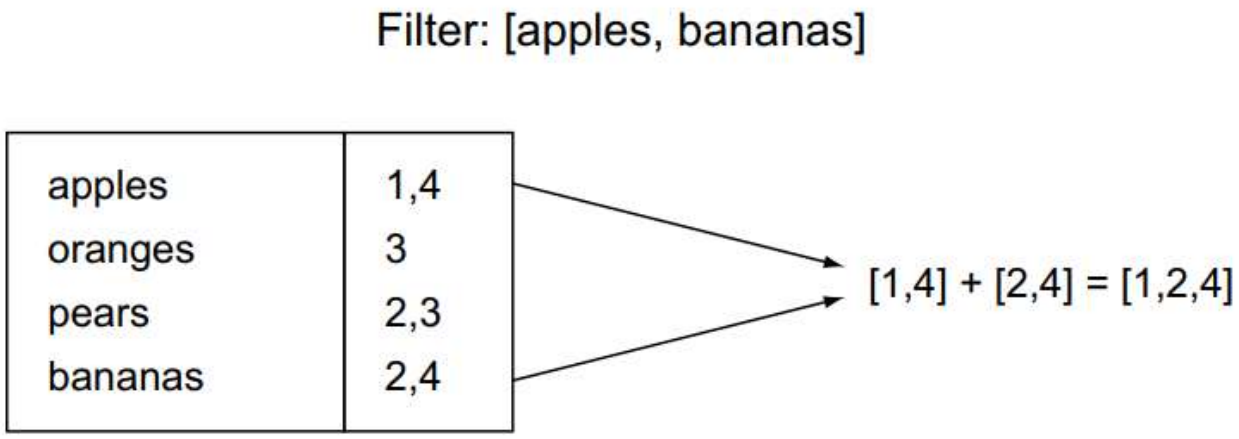
bool is fast when cached because it makes use of the two bitsets of the term filters.

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

RUNNING FILTERS ON FIELD DATA

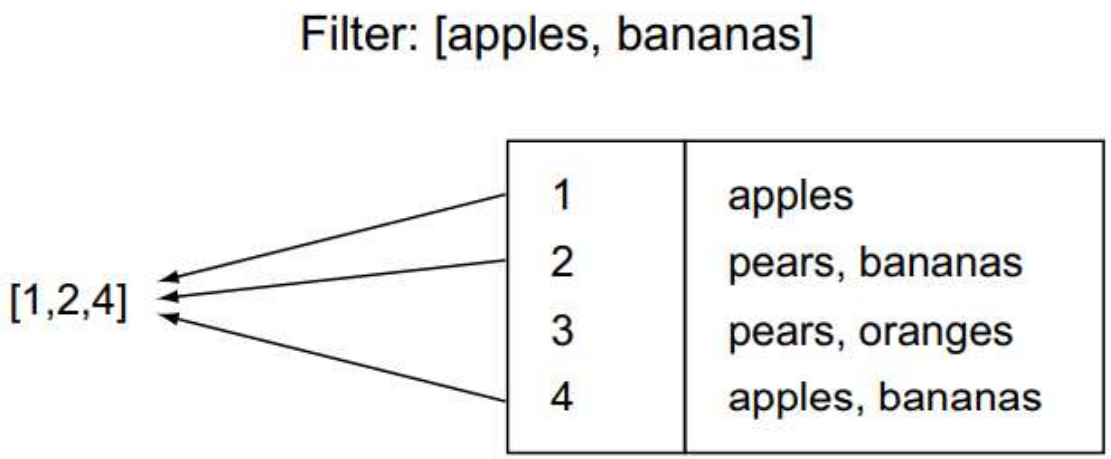
- We've discussed how bitsets and cached results make your filters faster.
- Some filters use bitsets; some can cache the overall results.
- Some filters can also run on field data. We first discussed field data as an in-memory structure that keeps a mapping of documents to terms.

RUNNING FILTERS ON FIELD DATA



By default, the `terms` filter is checking which documents match each term, and it intersects the lists.

RUNNING FILTERS ON FIELD DATA



Field data execution means iterating through documents but no list intersections.

RUNNING FILTERS ON FIELD DATA

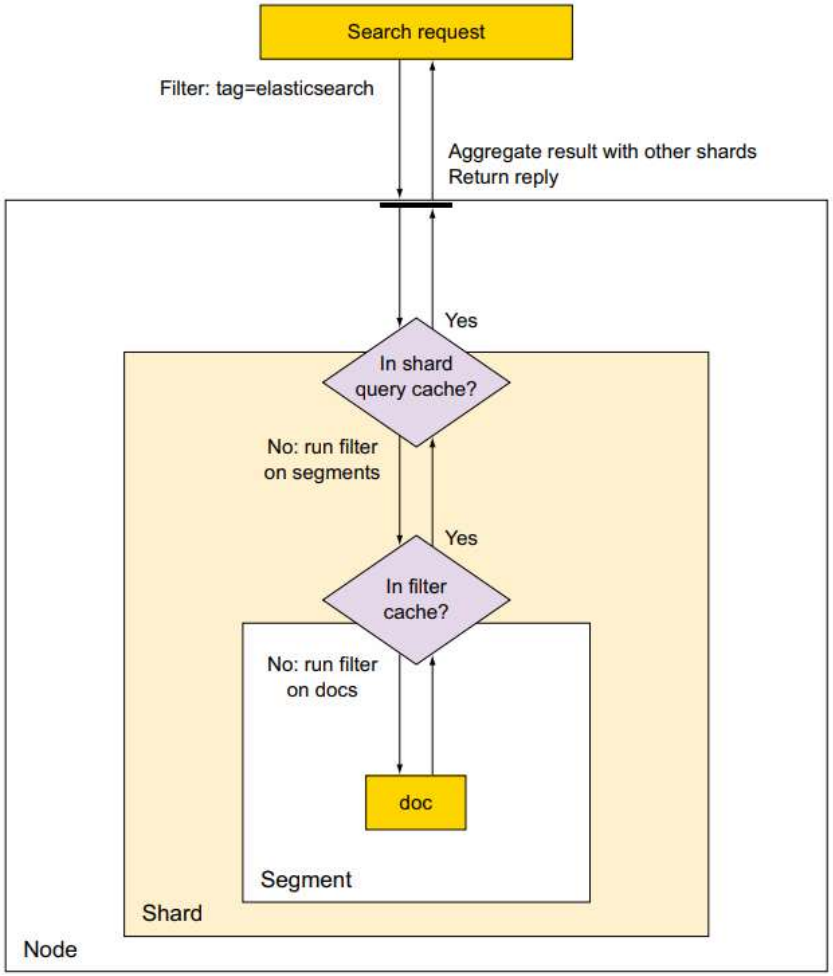
- For example, the following range filter will get events that happened in 2013 and will be executed on field data:

```
"filter": {
  "range": {
    "date": {
      "gte": "2013-01-01T00:00",
      "lt": "2014-01-01T00:00"
    },
    "execution": "fielddata"
  }
}
```

RUNNING FILTERS ON FIELD DATA

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


Search request
Filter: tag=elasticsearch
Aggregate result with other shards
Return reply
Yes
In shard query cache?
No: run filter on segments
Yes
In filter cache?
No: run filter on docs
doc
Segment
Shard
Node

Shard query cache

- 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.

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
}'
```

Shard query cache

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

JVM heap and OS caches

- If Elasticsearch doesn't have enough heap to finish an operation, it throws an out-ofmemory 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.

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

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, a good heap size is just large enough to accommodate the regular usage plus any spikes you might expect

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.

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}
```

Keeping caches up with warmers

- 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
```

Keeping caches up with warmers

Listing 10.8 Register warmer at index creation time

```
curl -XPUT 'localhost:9200/hot_index' -d '{
"warmers": {
  "date sorting": {
      "types": [],
      "source": {
        "sort": [{
          "date": {
            "order": "desc"
          }
        }]
      }
    }
}}'
```

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

Which types this warmer should run on. Empty means all types.

Under this key define the warmer itself.

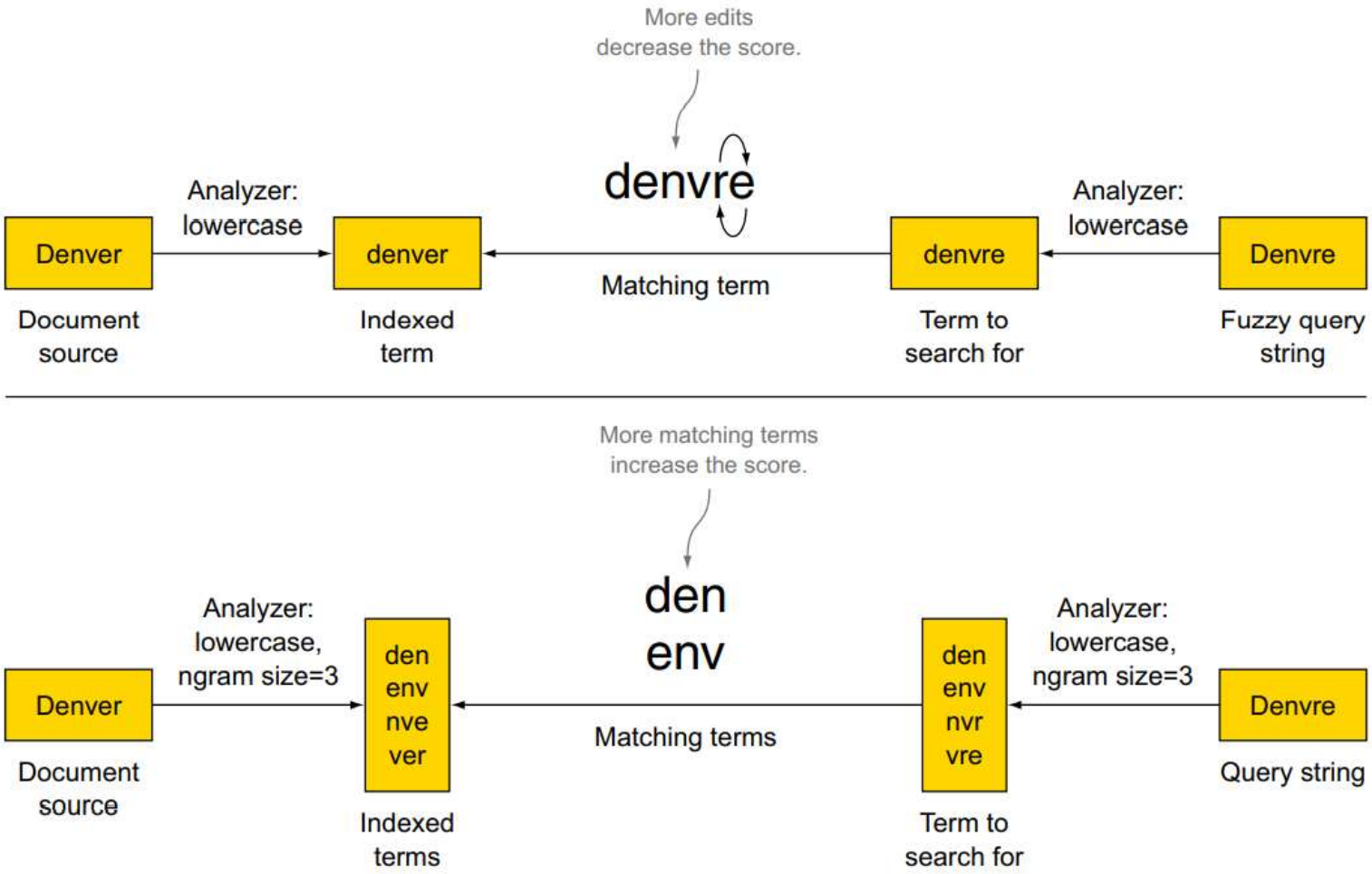
This warmer sorts by date.

Other performance tradeoffs

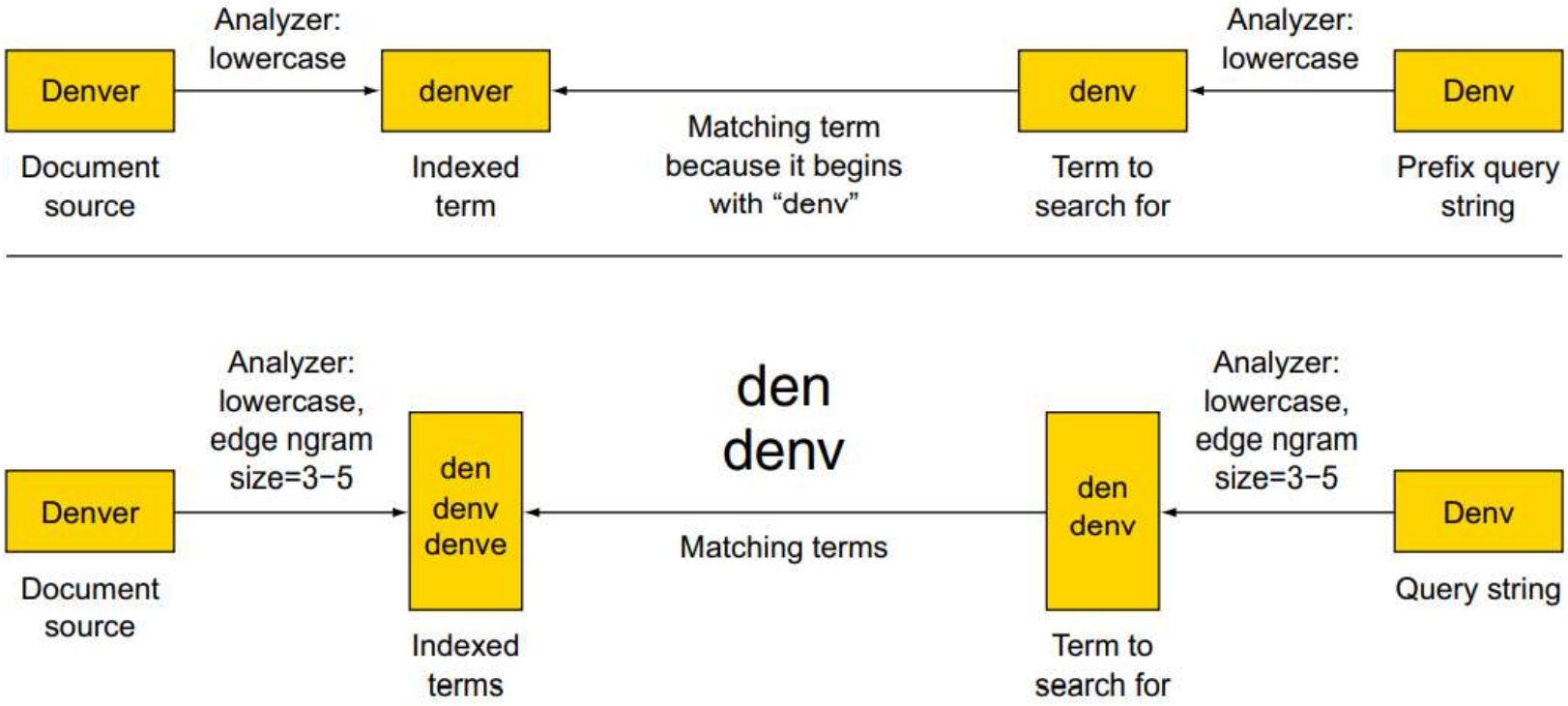
- In previous sections, you might have noticed that to make an operation fast, you need to pay with something.
- For example, if you make indexing faster by refreshing less often, you pay with searches that may not “see” recently indexed data.

Big indices or expensive searches

- 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.”


More edits
decrease the score.
denvre
Analyzer:
lowercase
Analyzer:
lowercase
Denver
denver
denvre
Denvre
Matching term
Document
source
Indexed
term
Term to
search for
Fuzzy query
string
More matching terms
increase the score.
den
env
Analyzer:
lowercase,
ngram size=3
Analyzer:
lowercase,
ngram size=3
Denver
den
env
nve
ver
den
env
nvr
vre
Denvre
Matching terms
Document
source
Indexed
terms
Term to
search for
Query string

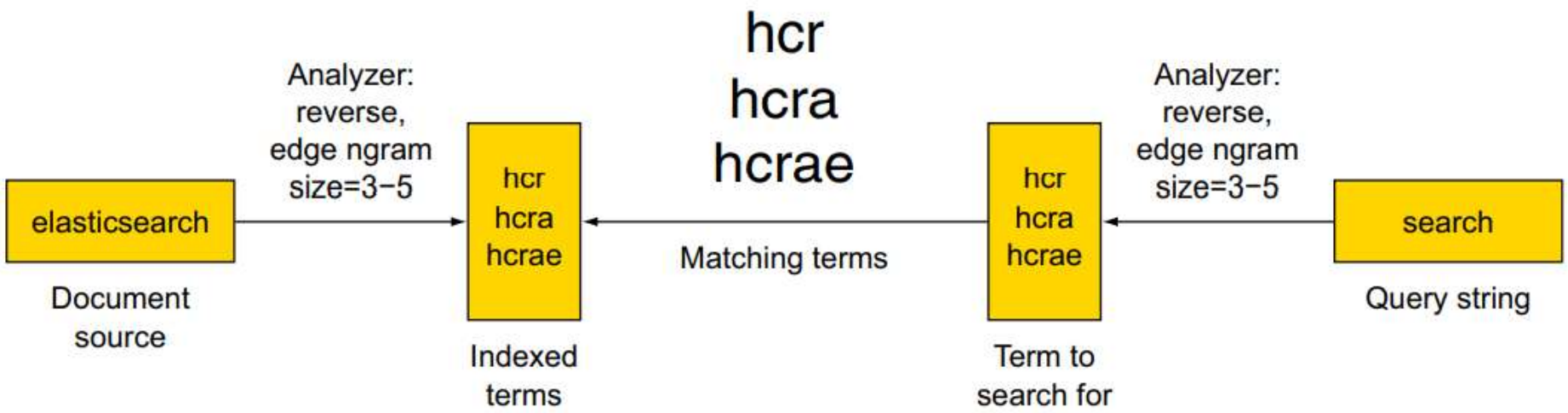
PREFIX QUERIES AND EDGE NGRAMS



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.

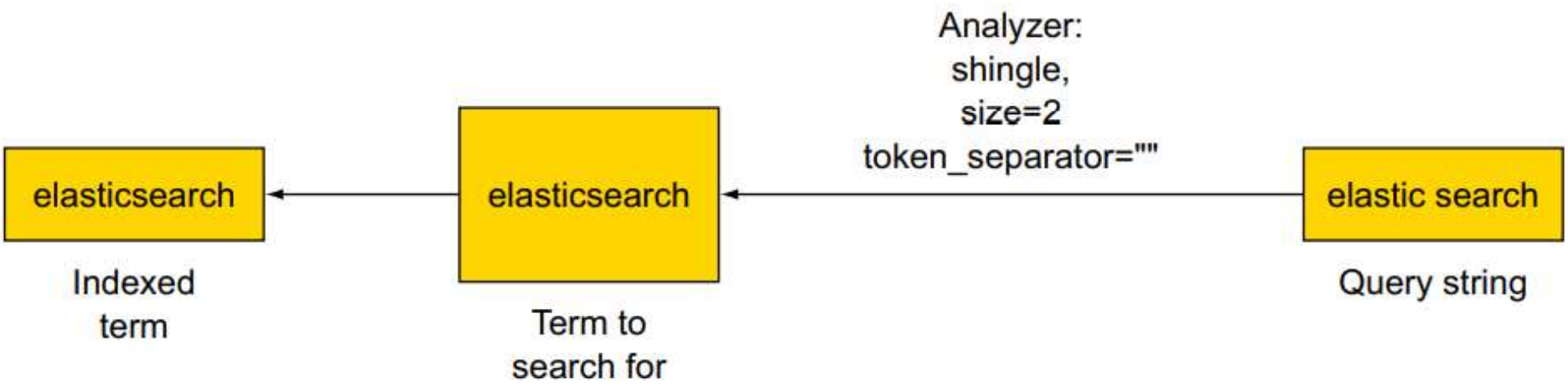
PREFIX QUERIES AND EDGE NGRAMS



PREFIX QUERIES AND EDGE NGRAMS

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

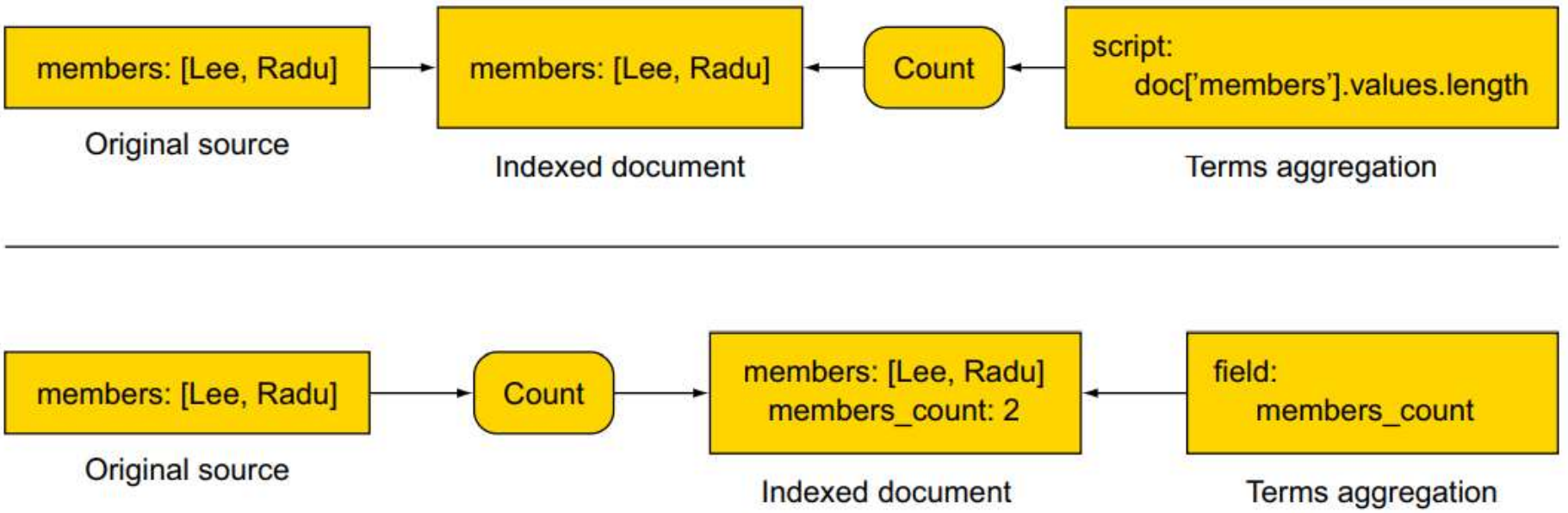
PREFIX QUERIES AND EDGE NGRAMS



Tuning scripts or not using them at all

- We first introduced scripts because they can be used for updates.
- You saw them, where you used them for sorting, 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

AVOIDING THE USE OF SCRIPTS



AVOIDING THE USE OF SCRIPTS

- 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.

```
"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.

NATIVE SCRIPTS

```
"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.

LUCENE EXPRESSIONS

- You want to calculate some stats based on that number:

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

TERM STATISTICS

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'].
- 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'].

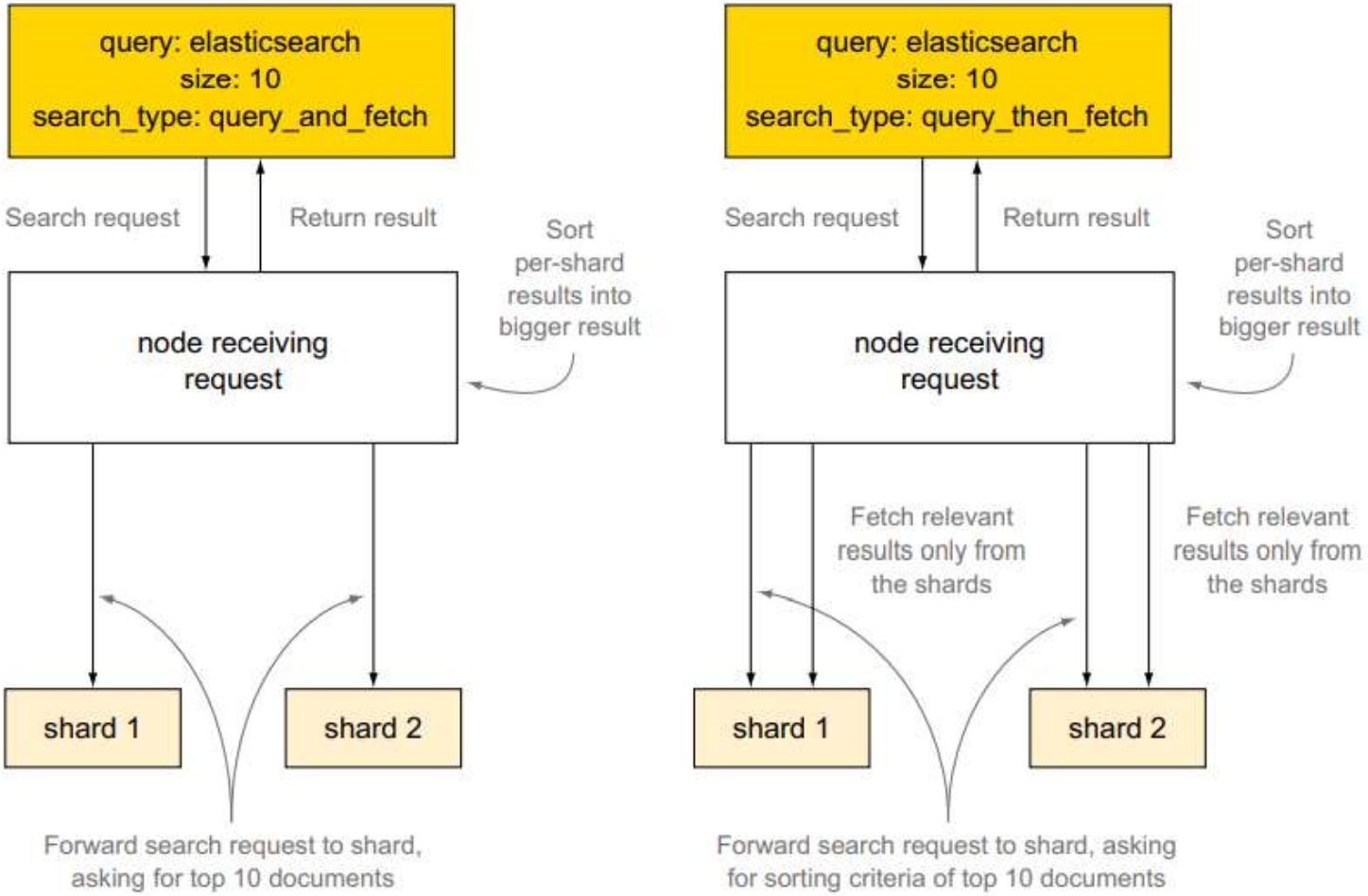
ACCESSING FIELD DATA

- 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:

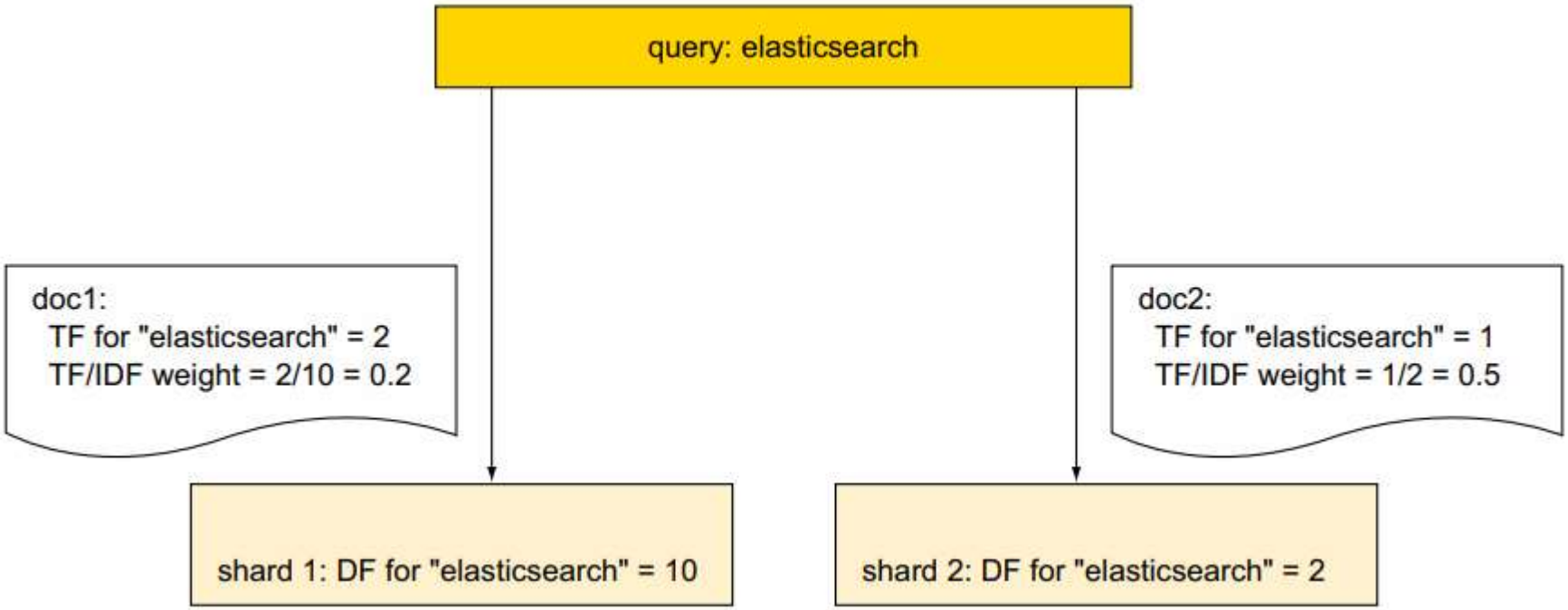
```
% curl 'localhost:9200/get-together/group/_search?pretty' -d '{
  "query": {
    "filtered": {
      "filter": {
        "script": {
          "script": "return
     doc.organizer.values.intersect(doc.members.values).isEmpty()",
        }
      }
    }
  }
}'
```

Trading network trips

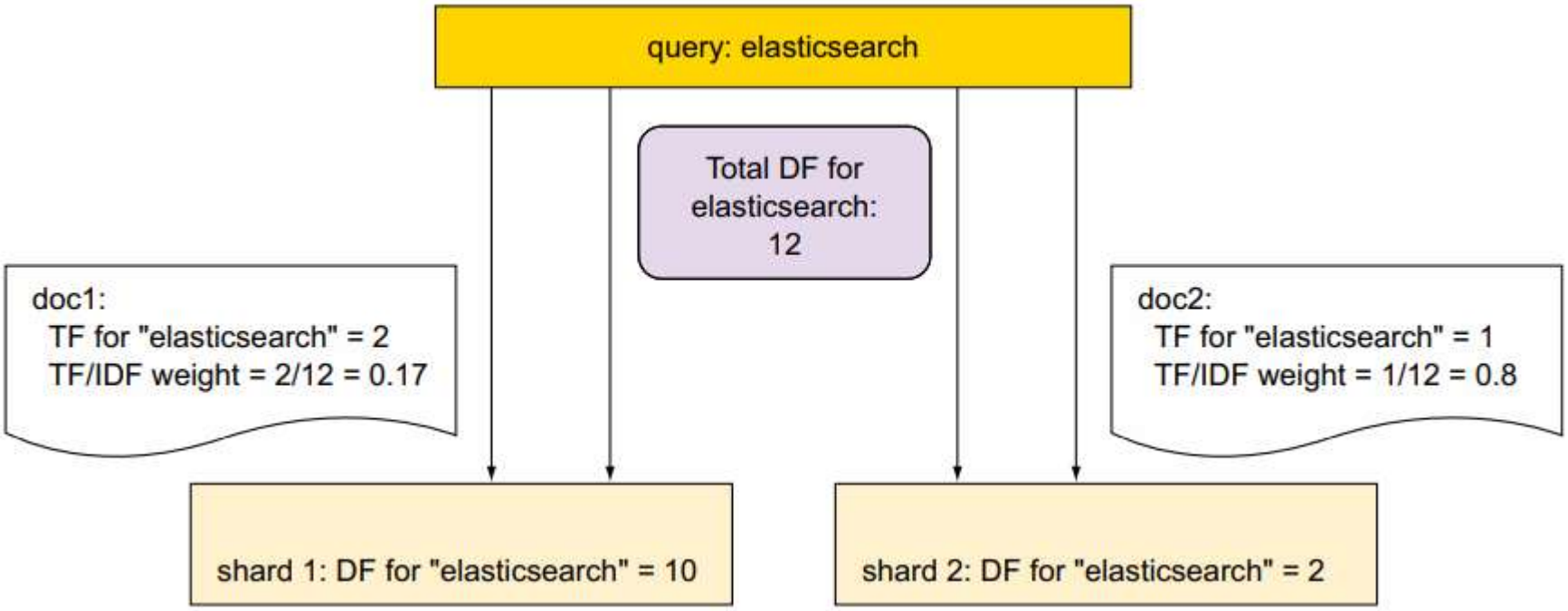
- Let's take a deeper look at how this works.
- The naïve approach would be to get N documents from all shards involved (where N is the value of size), sort them on the node that received the HTTP request (let's call it the coordinating node), pick the top N documents, and return them to the application.


query: elasticsearch
size: 10
search_type: query_and_fetch
Search request
Return result
Sort per-shard results into bigger result
node receiving request
shard 1
shard 2
Forward search request to shard, asking for top 10 documents
query: elasticsearch
size: 10
search_type: query_then_fetch
Search request
Return result
Sort per-shard results into bigger result
node receiving request
Fetch relevant results only from the shards
Fetch relevant results only from the shards
shard 1
shard 2
Forward search request to shard, asking for sorting criteria of top 10 documents

DISTRIBUTED SCORING



DISTRIBUTED SCORING



Trading memory for better deep paging

- 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
}'
```

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":
     "c2NhbjsxOzk2OjdZdkdQOTJLU1NpNGpxRWh4S0RWUVE7MTt0b3RhbF9oaXRzOjc7",
[...]
   "hits": {
      "total": 7,
      "max_score": 0,
      "hits": []
[...]
curl 'localhost:9200/_search/scroll?scroll=1m&pretty' -d
     'c2NhbjsxOzk2OjdZdkdQOTJLU1NpNGpxRWh4S0RWUVE7MTt0b3RhbF9oaXRzOjc7'
# reply
{
  "_scroll_id" : "c2NhbjswOzE7dG90YWxfaGl0czo3Ow==",
[...]
  "hits" : {
    "total" : 7,
    "max_score" : 0.0,
    "hits" : [ {
      "_index" : "get-together",
[...]
curl 'localhost:9200/_search/scroll?scroll=1m&pretty' -d
     'c2NhbjswOzE7dG90YWxfaGl0czo3Ow=='
```

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

The size of each page

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

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

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.

Trading memory for better deep paging

- 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"
    }
  }
}'
```

Summary

In this lesson 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 metasearch API, respectively.


3. Administering your cluster
LEARNING VOYAGE

Administering your cluster

This lesson covers

- Improving default configuration settings
- Creating default index settings with templates
- Monitoring for performance
- Using backup and restore

Improving defaults

- Although the out-of-the-box Elasticsearch configuration will satisfy the needs of most users
- It's important to note that it's a highly flexible system that can be tuned beyond its default settings for increased performance.

Index templates

Creating new indices and associated mappings in Elasticsearch is normally a simple task once the initial design planning has been completed. But there are some scenarios in which future indices must be created with the same settings and mappings as the previous ones. These scenarios include the following:

- Log aggregation
- Regulatory compliance
- Multi-tenancy

CREATING A TEMPLATE

- The index creation event will have to match the template pattern for the template to be applied.

- There are two ways to apply index templates to newly created indices in Elasticsearch:

1. By way of the REST API
2. By a configuration file

CREATING A TEMPLATE

```
curl -XPUT localhost:9200/_template/logging_index -d '{
    "template" : "logstash-*",
    "settings" : {
        "number_of_shards" : 2,
        "number_of_replicas" : 1
    },
    "mappings" : { … },
    "aliases" : { "november" : {} }
}'
```

PUT command

Applies this template to any index name that matches the pattern

TEMPLATES CONFIGURED ON THE FILE SYSTEM

- Template configurations must be in JSON format. For convenience, name them with a .json extension: .json.
- Template definitions should be located in the Elasticsearch configuration location under a templates directory. This path is defined in the cluster's configuration file (elasticsearch.yml) as path.conf; for example, <ES_HOME>/config/templates/*.
- Template definitions should be placed in the directories of nodes that are eligible to be elected as master.

TEMPLATES CONFIGURED ON THE FILE SYSTEM

- Using the previous template definition, your template.json file will look like this:

```
{
    "template" : "logstash-*",
    "settings" : {
        "number_of_shards" : 2,
        "number_of_replicas" : 1
    },
    "mappings" : { … },
    "aliases" : { "november" : {} }
}
```

MULTIPLE TEMPLATE MERGING

Listing 11.1 Configuring multiple templates

```
curl -XPUT localhost:9200/_template/logging_index_all -d '{
    "template" : "logstash-09-*",
    "order" : 1,
    "settings" : {
        "number_of_shards" : 2,
        "number_of_replicas" : 1
    },
    "mappings" : {
        "date" : { "store": false }
},
    "alias" : { "november" : {} }
}'

curl -XPUT http://localhost:9200/_template/logging_index -d '{
    "template" : "logstash-*",
    "order" : 0,
    "settings" : {
        "number_of_shards" : 2,
        "number_of_replicas" : 1
    },
    "mappings" : {
     "date" : { "store": true }
    }
}'
```

Apply this template to any index beginning with "logstash-09-".

Highest order number will override the lowest order number setting

Apply this template to any index beginning with "logstash-*" and store the date field.

RETRIEVING INDEX TEMPLATES

To retrieve a list of all templates, a convenience API exists:

```
curl -XGET localhost:9200/_template/
```

Likewise, you're able to retrieve either one or many individual templates by name:

```
curl -XGET localhost:9200/_template/logging_index
curl -XGET localhost:9200/_template/logging_index_1,logging_index_2
```

Or you can retrieve all template names that match a pattern:

```
curl -XGET localhost:9200/_template/logging_*
```

DELETING INDEX TEMPLATES

Deleting a template index is achieved by using the template name. In the previous section, we defined a template as such:

```
curl -XPUT 'localhost:9200/_template/logging_index' -d '{ … }'
```

To delete this template, use the template name in the request:

```
curl -XDELETE 'localhost:9200/_template/logging_index'
```

Default mappings

- We just showed you how index templates can be used to save time and add uniformity across similar datatypes.
- Default mappings have the same beneficial effects and can be thought of in the same vein as templates for mapping types.
- Default mappings are most often used when there are indices with similar fields.
- Specifying a default mapping in one place removes the need to repeatedly specify it across every index.

DYNAMIC MAPPINGS

- By default, Elasticsearch employs dynamic mapping: the ability to determine the datatype for new fields within a document.

- You may have experienced this when you first indexed a document and noticed that Elasticsearch dynamically created a mapping for it as well as the datatype for each of the fields.

DYNAMIC MAPPINGS

- The next listing shows how to add a dynamic mapping

Listing 11.2 Adding a dynamic mapping

```
curl -XPUT 'localhost:9200/first_index' -d
'{
    "mappings": {
        "person": {
            "dynamic":      "strict",
            "properties": {
                "email":  { "type": "string"},
                "created_date":  { "type": "date" }
            }
        }
    }
}'
```

Throw exception if an unknown field is encountered at index time.

DYNAMIC MAPPINGS

```
curl -XPUT 'localhost:9200/second_index' -d
'{
    "mappings": {
        "person": {
            "dynamic":      "true",
            "properties": {
                "email":   { "type": "string"},
                "created_date":  { "type": "date" }
            }
        }
    }
}'
```

Allow the dynamic creation of new fields.

DYNAMIC MAPPINGS

- For instance, try to index a document with an additional first_name field added:

```
curl -XPOST 'localhost:9200/first_index/person' -d
'{
"email": "foo@bar.com",
"created_date" : "2014-09-01",
"first_name" : "Bob"
}'
```

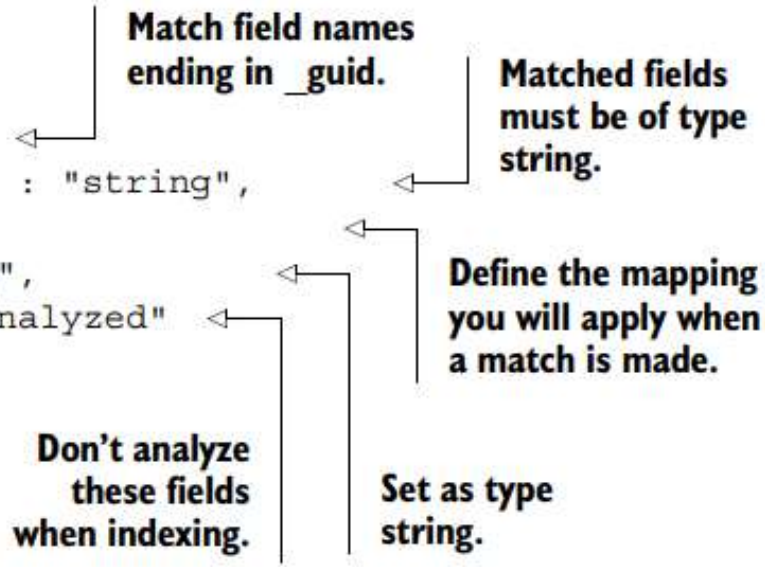
Here's the response:

```
{
error: "StrictDynamicMappingException[mapping set to strict, dynamic
     introduction of [first_name] within [person] is not allowed]"
status: 400
}
```

DYNAMIC MAPPING AND TEMPLATING TOGETHER

- Earlier we explored how index templates can be used to autodefine newly created indices for a uniform set of indices and mappings.
- We can expand on this idea now by incorporating what we've covered with dynamic mappings.
- The following example solves a simple problem when dealing with data comprising UUIDs

```
curl -XPUT 'http://localhost:9200/myindex' -d '
{
    "mappings" : {

        "my_type" : {
            "dynamic_templates" : [{
                "UUID" : {
                    "match" : "*_guid",
                    "match_mapping_type" : "string",
                    "mapping" : {
                        "type" : "string",
                        "index" : "not_analyzed"
                    }
                }
            }]
        }
    }
};
```



DYNAMIC MAPPING AND TEMPLATING TOGETHER

- If you wanted to match something like person.*.email. Using this logic, you can see a match on a data structure such as this:

```
{
 "person" : {
 "user" : {
 "email": { "bob@domain.com" }
 }
}
```

Allocation awareness

- This section covers the concept of laying out cluster topology to reduce central points of failure and improve performance by using the concept of allocation awareness.

- Allocation awareness is defined as knowledge of where to place copies (replicas) of data.

Shard-based allocation

- Allocation awareness allows you to configure shard allocation using a self-defined parameter.
- This is a common best practice in Elasticsearch deployments because it reduces the chances of having a single point of failure by making sure data is evened out among the network topology.

Shard-based allocation

- Enabling allocation awareness is achieved by defining a grouping key and setting it in the appropriate nodes.

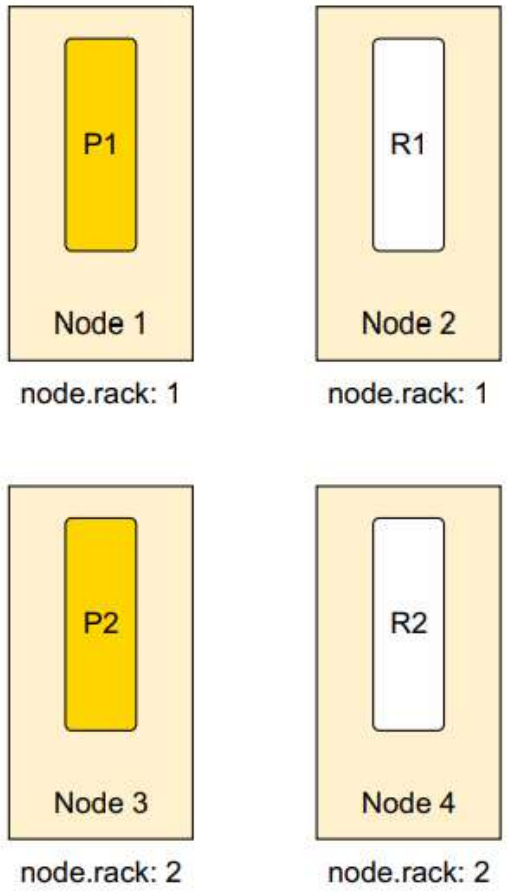
- For instance, you can edit elasticsearch.yml as follows:

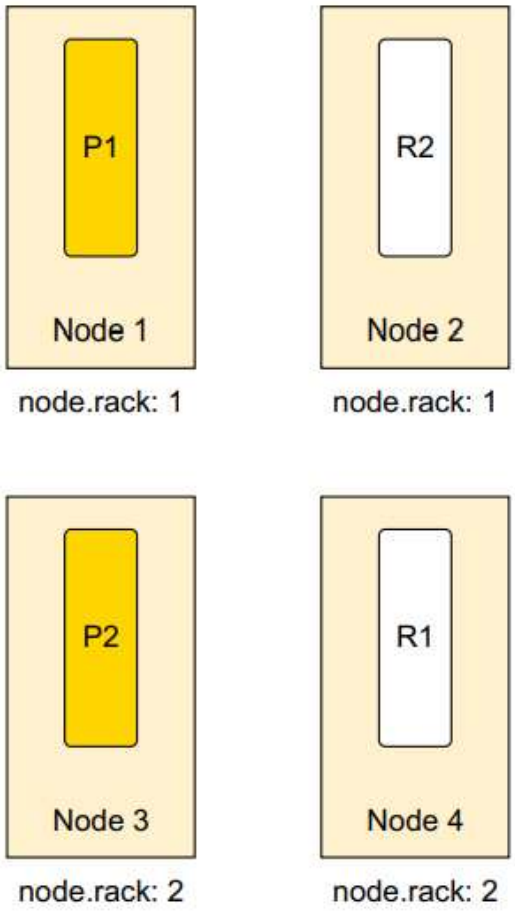
```
cluster.routing.allocation.awareness.attributes: rack
```

Shard-based allocation

- Note that Elasticsearch allows you to set metadata on nodes. In this case, the metadata key will be your allocation awareness parameter:

```
node.rack: 1
```


P1
Node 1
node.rack: 1
R1
Node 2
node.rack: 1
P2
Node 3
node.rack: 2
R2
Node 4
node.rack: 2


P1
Node 1
node.rack: 1
R2
Node 2
node.rack: 1
P2
Node 3
node.rack: 2
R1
Node 4
node.rack: 2

Forced allocation awareness

- Forced allocation awareness is useful when you know in advance the group of values and want to limit the number of replicas for any given group.

- A real-world example of where this is commonly used is in cross-zone allocation on Amazon Web Services or other cloud providers with multizone capabilities.

Forced allocation awareness

- For example, in this use case you want to enforce allocation at a zone level.
- First you specify your attribute, zone, as you did before.
- Next, you add dimensions to that group: us-east and us-west.
- In your elasticsearch.yml, you add the following:

```
cluster.routing.allocation.awareness.attributes: zone
cluster.routing.allocation.force.zone.values: us-east, us-west
```

Forced allocation awareness

- Elasticsearch applies the settings even after a restart, or temporary (transient):

```
curl -XPUT localhost:9200/_cluster/settings -d '{
        "persistent" : {
        "cluster.routing.allocation.awareness.attributes": zone
        "cluster.routing.allocation.force.zone.values": us-east, us-west
        }
}
```

Monitoring for bottlenecks

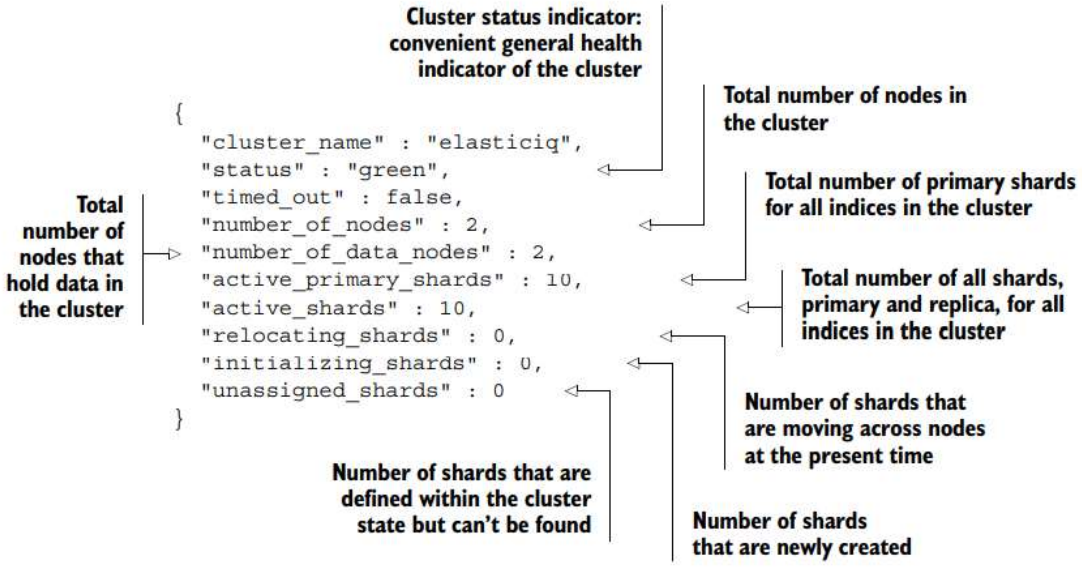
- Elasticsearch provides a wealth of information via its APIs: memory consumption, node membership, shard distribution, and I/O performance.
- The cluster and node APIs help you gauge the health and overall performance metrics of your cluster.

Checking cluster health

Listing 11.3 Cluster health API request

```
curl -XGET 'localhost:9200/_cluster/health?pretty';
```

And the response:



Checking cluster health

- Armed with this knowledge, you can now take a look at a cluster with a yellow status and attempt to track down the source of the problem:

```
curl -XGET 'localhost:9200/_cluster/health?pretty';
{
  "cluster_name" : "elasticiq",
  "status" : "yellow",
  "timed_out" : false,
  "number_of_nodes" : 1,
  "number_of_data_nodes" : 1,
  "active_primary_shards" : 10,
  "active_shards" : 10,
  "relocating_shards" : 0,
  "initializing_shards" : 0,
  "unassigned_shards" : 5
}
```

```
curl -XGET 'localhost:9200/_cluster/health?level=indices&pretty';
{
  "cluster_name" : "elasticiq",
  "status" : "yellow",
  "timed_out" : false,
  "number_of_nodes" : 1,
  "number_of_data_nodes" : 1,
  "active_primary_shards" : 10,
  "active_shards" : 10,
  "relocating_shards" : 0,
  "initializing_shards" : 0,
  "unassigned_shards" : 5,
  "indices" : {
    "bitbucket" : {
      "status" : "yellow",
      "number_of_shards" : 5,
      "number_of_replicas" : 1,
      "active_primary_shards" : 5,
      "active_shards" : 5,
      "relocating_shards" : 0,
      "initializing_shards" : 0,
      "unassigned_shards" : 5
    }...
```

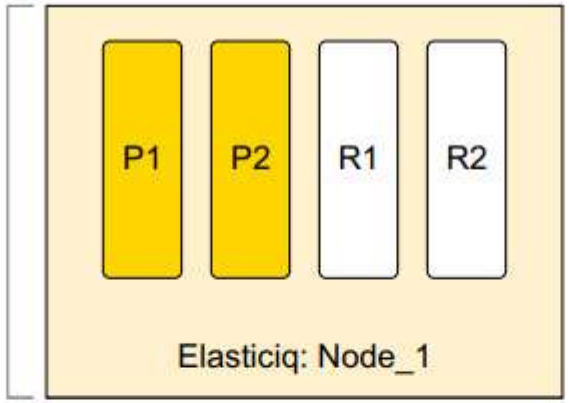
Note that the cluster has only one node running.

The primary shards

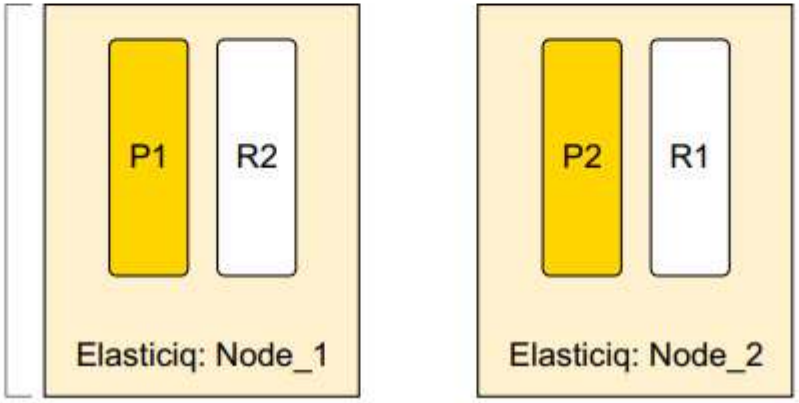
Here you tell Elasticsearch to allocate one replica per primary shard.

Unassigned shards caused by a lack of available nodes to support the replica definition

Yellow status: Single-node cluster with all shards confined to one node



Green status: New node added, causing even distribution of replicas



Yellow status solved by making nodes accessible

CPU: slow logs, hot threads, and thread pools

- Monitoring your Elasticsearch cluster may from time to time expose spikes in CPU usage or bottlenecks in performance caused by a constantly high CPU utilization or blocked/waiting threads.

SLOW LOGS

- It's important to note at this point that these settings can also be modified using the '{index_name}/_settings' endpoint:

```
index.search.slowlog.threshold.query.warn: 10s
index.search.slowlog.threshold.query.info: 1s
index.search.slowlog.threshold.query.debug: 2s
index.search.slowlog.threshold.query.trace: 500ms

index.search.slowlog.threshold.fetch.warn: 1s
index.search.slowlog.threshold.fetch.info: 1s
index.search.slowlog.threshold.fetch.debug: 500ms
index.search.slowlog.threshold.fetch.trace: 200ms
```

SLOW LOGS

- The actual log file you'll be outputting to is configured in your logging.yml file, along with other logging functionality, as shown here:

```
index_search_slow_log_file:
  type: dailyRollingFile
  file: ${path.logs}/${cluster.name}_index_search_slowlog.log
  datePattern: "'.'yyyy-MM-dd"
  layout:
    type: pattern
    conversionPattern: "[%d{ISO8601}][%-5p][%-25c] %m%n"
```

SLOW LOGS

- The typical output on a slow log file will appear as this:

```
[2014-11-09 16:35:36,325][INFO ][index.search.slowlog.query] [ElasticIQ-
Master] [streamglue][4] took[10.5ms], took_millis[10], types[], stats[],
search_type[QUERY_THEN_FETCH], total_shards[10],
source[{"query":{"filtered":{"query":{"query_string":{"query":"test"}}}}},...]

[2014-11-09 16:35:36,339][INFO ][index.search.slowlog.fetch] [ElasticIQ-
Master] [streamglue][3] took[9.1ms], took_millis[9], types[], stats[],
search_type[QUERY_THEN_FETCH], total_shards[10], ...
```

SLOW INDEX LOG

- Equally useful in discovering bottlenecks during index operations is the slow index log.
- Its thresholds are defined in your cluster configuration file, or via the index update settings API, much like the previous slow log:

```
index.indexing.slowlog.threshold.index.warn: 10s
index.indexing.slowlog.threshold.index.info: 5s
index.indexing.slowlog.threshold.index.debug: 2s
index.indexing.slowlog.threshold.index.trace: 500ms
```

SLOW INDEX LOG

- As before, the output of any index operation meeting the threshold values will be written to your log file, and you'll see the [index][shard_number] ([bitbucket][2]) and duration (took[4.5ms]) of the index operation:

```
[2014-11-09 18:28:58,636][INFO ][index.indexing.slowlog.index] [ElasticIQ-
Master] [bitbucket][2] took[4.5ms], took_millis[4], type[test],
id[w0QyH_m6Sa2P-juppUy3Tw], routing[], source[] ...
```

HOT_THREADS API

- Note that unlike other APIs, hot_threads doesn't return JSON but instead returns formatted text:

```
curl -XGET 'http://127.0.0.1:9200/_nodes/hot_threads';
```

Here's the sample output:

```
::: [ElasticIQ-Master][AtPvr5Y3ReW-ua7ZPtPfuQ][loki.local][inet[/
127.0.0.1:9300]]{master=true}
    37.5% (187.6micros out of 500ms) cpu usage by thread
'elasticsearch[ElasticIQ-Master][search][T#191]
10/10 snapshots sharing following 3 elements
```

HOT_THREADS API

The output of the hot_threads API requires some parsing to understand correctly, so let's have a look at what information it provides on CPU performance:

```
::: [ElasticIQ-Master][AtPvr5Y3ReW-ua7ZPtPfuQ][loki.local][inet[/
127.0.0.1:9300]]{master=true}
```

The top line of the response includes the node identification. Because the cluster presumably has more than one node, this is the first indication of which CPU the thread information belongs to:

```
    37.5% (187.6micros out of 500ms) cpu usage by thread
'elasticsearch[ElasticIQ-Master][search][T#191]
```

HOT_THREADS API

- Other possible output identifiers here are block usage, which identifies threads that are blocked, and wait usage for threads in a WAITING state:

10/10 snapshots sharing following 3 elements

HOT_THREADS API

- You can tune the information-gathering process by adding parameters to the hot_threads API call:

```
curl -XGET 'http://127.0.0.1:9200/_nodes/
hot_threads?type=wait&interval=1000ms&threads=3';
```

THREAD POOLS

- Every node in a cluster manages thread pools for better management of CPU and memory usage.
- Elasticsearch will seek to manage thread pools to achieve the best performance on a given node.
- In some cases, you'll need to manually configure and override how thread pools are managed to avoid cascading failure scenarios.

THREAD POOLS

- It's also worth noting here that the Cluster Settings API allows you to update these settings on a running cluster as well:

```
# Bulk Thread Pool
threadpool.bulk.type: fixed
threadpool.bulk.size: 40
threadpool.bulk.queue_size: 200
```

Memory: heap size, field, and filter caches

- This section will explore efficient memory management and tuning for Elasticsearch clusters.

- Many aggregation and filtering operations are memory-bound within Elasticsearch, so knowing how to effectively improve the default memory-management settings in Elasticsearch and the underlying JVM will be a useful tool for scaling your cluster.

HEAP SIZE

- Elasticsearch is a Java application that runs on the Java Virtual Machine (JVM), so it's subject to memory management by the garbage collector.

- The concept behind the garbage collector is a simple one: it's triggered when memory is running low, clearing out objects that have been dereferenced and thus freeing up memory for other JVM applications to use.

FILTER AND FIELD CACHE

- Caches play an important role in Elasticsearch performance, allowing for the effective use of filters, facets, and index field sorting.

- This section will explore two of these caches: the filter cache and the field data cache

FILTER AND FIELD CACHE

Two types of filter caches are available in Elasticsearch:

- Index-level filter cache
- Node-level filter cache

FIELD-DATA CACHE

- The field-data cache is used to improve query execution times.
- Elasticsearch loads field values into memory when you run a query and keeps those values in the fielddata cache for subsequent requests to use.
- Because building this structure in memory is an expensive operation, you don’t want Elasticsearch performing this on every request, so the performance gains are noticeable.

FIELD-DATA CACHE

To retrieve the current state of the field-data cache, there are some handy APIs available:

- Per-Node:

```
curl -XGET 'localhost:9200/_nodes/stats/indices/
fielddata?fields=*&pretty=1';
```

- Per-Index:

```
curl -XGET 'localhost:9200/_stats/fielddata?fields=*&pretty=1';
```

- Per-Node Per-Index:

```
curl -XGET 'localhost:9200/_nodes/stats/indices/
fielddata?level=indices&fields =*&pretty=1';
```

FIELD-DATA CACHE

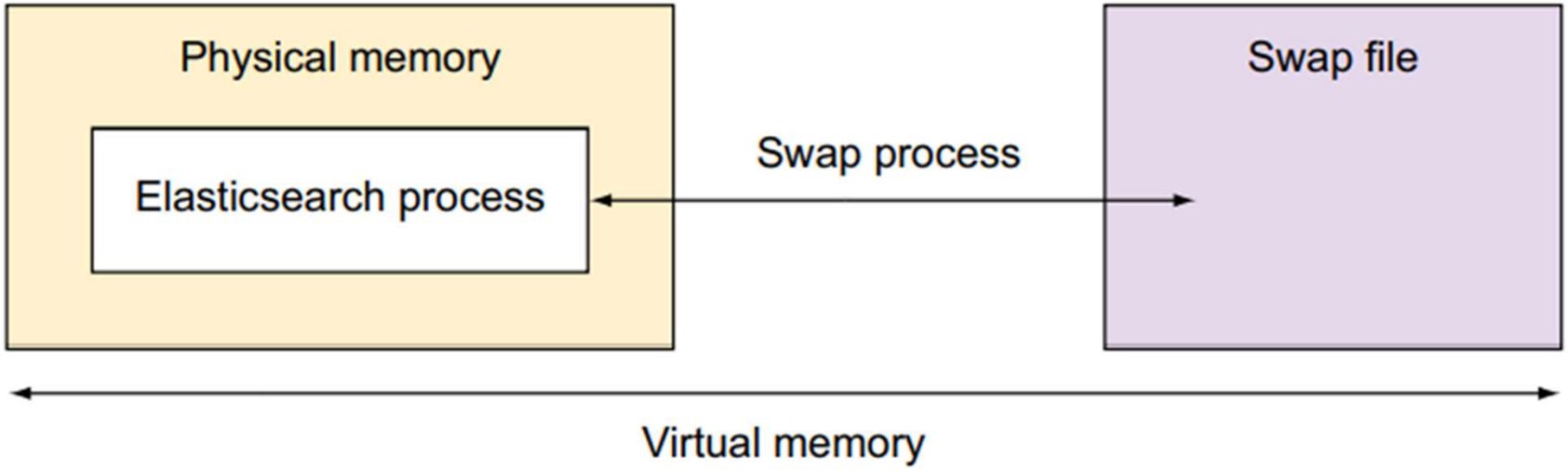
- Specifying fields=* will return all field names and values.
- The output of these APIs looks similar to the following:

```
"indices" : {
  "bitbucket" : {
    "fielddata" : {
      "memory_size_in_bytes" : 1024mb,
      "evictions" : 200,
      "fields" : { … }
    }
  }, ...
```

CIRCUIT BREAKER

- As mentioned in the previous section, the field-data cache may grow to the point that it causes an OutOfMemory exception.
- This is because the field-data size is calculated after the data is loaded.
- To avoid such events, Elasticsearch provides circuit breakers.
- Circuit breakers are artificial limits imposed to help reduce the chances of an OutOfMemory exception.

AVOIDING SWAP



- Sample error in the log:

```
[2014-11-21 19:22:00,612][ERROR][common.jna]
Unknown mlockall error 0
```

- API request:

```
curl -XGET 'localhost:9200/_nodes/process?pretty=1';
```

- Response:

```
...
 "process" : {
        "refresh_interval_in_millis" : 1000,
        "id" : 9809,
        "max_file_descriptors" : 10240,
        "mlockall" : false
      } ...
```

OS caches

- Elasticsearch and Lucene leverage the OS file-system cache heavily due to Lucene's immutable segments.
- Lucene is designed to leverage the underlying OS file-system cache for in-memory data structures.
- Lucene segments are stored in individual immutable files.
- Immutable files are considered to be cache-friendly, and the underlying OS is designed to keep "hot" segments resident in memory for faster access.

OS caches

- To achieve this, your new index, myindex, will now be created only on nodes that have tag set to mynode1 and mynode2, with the following command:

```
curl -XPUT localhost:9200/myindex/_settings -d '{
 "index.routing.allocation.include.tag" : "mynode1,mynode2"
}'
```

Store throttling

- Apache Lucene stores its data in immutable segment files on disk, Immutable files are by definition written only once by Lucene but read many times.
- Merge operations work on these segments because many segments are read at once when a new one is written.
- Although these merge operations normally don’t task a system heavily, systems with low I/O can be impacted negatively when merges, indexing, and search operations are all occurring at the same time.

Store throttling

- For node-level throttling, use indices.store.throttle.max_bytes_per_sec, and for index-level throttling, use index.store.throttle.max_bytes_per_sec.
- Note that the values are expressed in megabytes per second:

indices.store.throttle.max_bytes_per_sec : "50mb"

or

index.store.throttle.max_bytes_per_sec : "10mb"

Backing up your data

- Elasticsearch provides a full-featured and incremental data backup solution.

- The snapshot and restore APIs enable you to back up individual index data, all of your indices, and even cluster settings to either a remote repository or other pluggable backend systems and then easily restore these items to the existing cluster or a new one.

Snapshot API

- Using the snapshot API to back up your data for the first time, Elasticsearch will take a copy of the state and data of your cluster.
- All subsequent snapshots will contain the changes from the previous one.
- The snapshot process is nonblocking, so executing it on a running system should have no visible effect on performance.

Snapshot API

It’s important to note that snapshots are stored in repositories. A repository can be defined as either a file system or a URL.

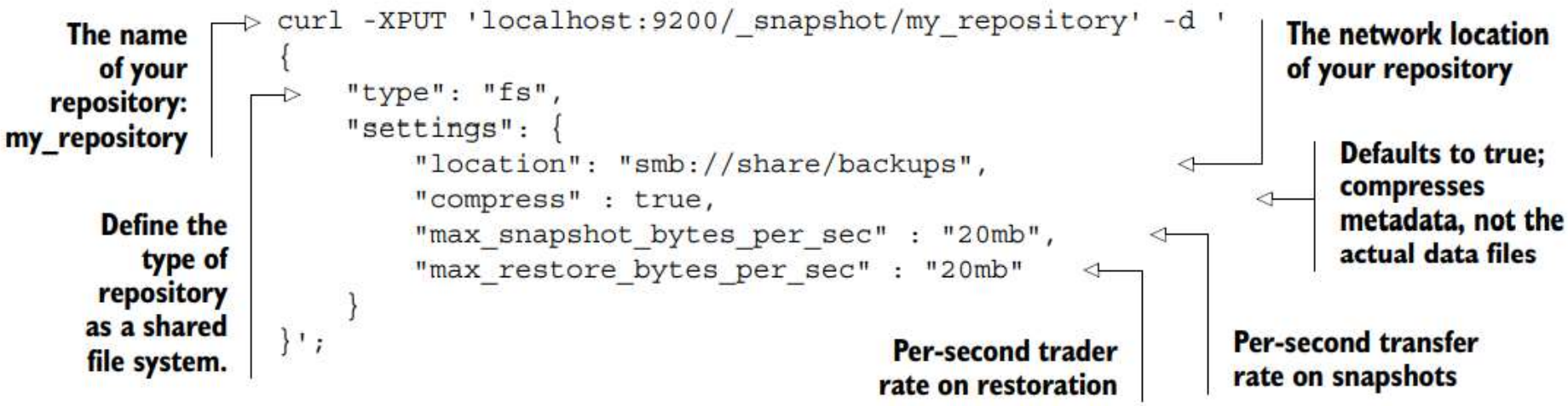
- A file-system repository requires a shared file system, and that shared file system must be mounted on every node in the cluster.
- URL repositories are read-only and can be used as an alternative way to access snapshots.

Backing up data to a shared file system

- Define a repository—Instruct Elasticsearch on how you want the repository structured.
- Confirm the existence of the repository—You want to trust but verify that the repository was created using your definition.
- Execute the backup—Your first snapshot is executed via a simple REST API command.

Backing up data to a shared file system

Listing 11.4 Defining a new repository



Backing up data to a shared file system

- Once the repository has been defined across your cluster, you can confirm its existence with a simple GET command:

```
curl -XGET 'localhost:9200/_snapshot/my_repository?pretty=1';
{
  "my_repository" : {
    "type" : "fs",
    "settings" : {
      "compress" : "true",
      "max_restore_bytes_per_sec" : "20mb",
      "location" : "smb://share/backups",
      "max_snapshot_bytes_per_sec" : "20mb"
    }
  }
}
```

Note that as a default action, you don't have to specify the repository name, and Elasticsearch will respond with all registered repositories for the cluster:

```
curl -XGET 'localhost:9200/_snapshot?pretty=1';
```

Once you've established a repository for your cluster, you can go ahead and create your initial snapshot/backup:

```
curl -XPUT 'localhost:9200/_snapshot/my_repository/first_snapshot';
```

This command will trigger a snapshot operation and return immediately. If you want to wait until the snapshot is complete before the request responds, you can append the optional `wait_for_completion` flag:

```
curl -XPUT 'localhost:9200/_snapshot/my_repository/
first_snapshot?wait_for_completion=true';
```

Backing up data to a shared file system

- Now take a look at your repository location and see what the snapshot command stored away:

```
./backups/index
./backups/indices/bitbucket/0/__0
./backups/indices/bitbucket/0/__1
./backups/indices/bitbucket/0/__10
./backups/indices/bitbucket/1/__c
./backups/indices/bitbucket/1/__d
./backups/indices/bitbucket/1/snapshot-first_snapshot
…
./backups/indices/bitbucket/snapshot-first_snapshot
./backups/metadata-first_snapshot
./backups/snapshot-first_snapshot
```

```
smb://share/backups/indices/bitbucket/0/snapshot-first_snapshot
{
  "name" : "first_snapshot",
  "index_version" : 18,
  "start_time" : 1416687343604,
  "time" : 11,
  "number_of_files" : 20,
  "total_size" : 161589,
  "files" : [ {
    "name" : "__0",
    "physical_name" : "_1.fnm",
    "length" : 2703,
    "checksum" : "1ot813j",
    "written_by" : "LUCENE_4_9"
  }, {
    "name" : "__1",
    "physical_name" : "_1_Lucene49_0.dvm",
    "length" : 90,
    "checksum" : "1h6yhga",
    "written_by" : "LUCENE_4_9"
  }, {
    "name" : "__2",
    "physical_name" : "_1.si",
    "length" : 444,
    "checksum" : "afusmz",
    "written_by" : "LUCENE_4_9"
  }
```

SECOND SNAPSHOT

- Because snapshots are incremental, only storing the delta between them, a second snapshot command will create a few more data files but won't recreate the entire snapshot from scratch:

```
curl -XPUT
'localhost:9200/_snapshot/my_repository/second_snapshot';
```

SECOND SNAPSHOT

- Analyzing the new directory structure, you can see that only one file was modified:

- the existing /index file in the root directory. Its contents now hold a list of the snapshots taken:

{"snapshots":["first_snapshot","second_snapshot"]}

SNAPSHOTS ON A PER-INDEX BASIS

- In the previous example, you saw how you can take snapshots of the entire cluster and all indices.
- It's important to note here that snapshots can be taken on a per-index basis, by specifying the index in the PUT command:

```
curl -XPUT 'localhost:9200/_snapshot/my_repository/third_snapshot' -d '
{
  "indices": "logs-2014,logs-2013"
};
```

Comma-separated list of index names to snapshot

SNAPSHOTS ON A PER-INDEX BASIS

- Retrieving basic information on the state of a given snapshot (or all snapshots) is achieved by using the same endpoint, with a GET request:

```
curl -XGET
'localhost:9200/_snapshot/my_repository/first_snapshot
?pretty';
```

```
{
  "snapshots": [
    {
      "snapshot": "first_snapshot",
      "indices": [
        "bitbucket"
      ],
      "state": "SUCCESS",
      "start_time": "2014-11-02T22:38:14.078Z",
      "start_time_in_millis": 1414967894078,
      "end_time": "2014-11-02T22:38:14.129Z",
      "end_time_in_millis": 1414967894129,
      "duration_in_millis": 51,
      "failures": [],
      "shards": {
        "total": 10,
        "failed": 0,
        "successful": 10
      }
    }
  ]
}
```

SNAPSHOTS ON A PER-INDEX BASIS

- Substituting the snapshot name for _all will supply you with information regarding all snapshots in the repository:

```
curl -XGET
'localhost:9200/_snapshot/my_repository/_all';
```

SNAPSHOTS ON A PER-INDEX BASIS

- Because snapshots are incremental, you must take special care when removing old snapshots that you no longer need.
- It's always advised that you use the snapshot API in removing old snapshots because the API will delete only currently unused segments of data:

```
curl -XDELETE
'localhost:9200/_snapshot/my_repository/first_snapshot';
```

Restoring from backups

- Snapshots are easily restored to any running cluster, even a cluster the snapshot didn't originate from. Using the snapshot API with an added _restore command, you can restore the entire cluster state:

```
curl -XPOST
'localhost:9200/_snapshot/my_repository/first_snapshot/
_restore';
```

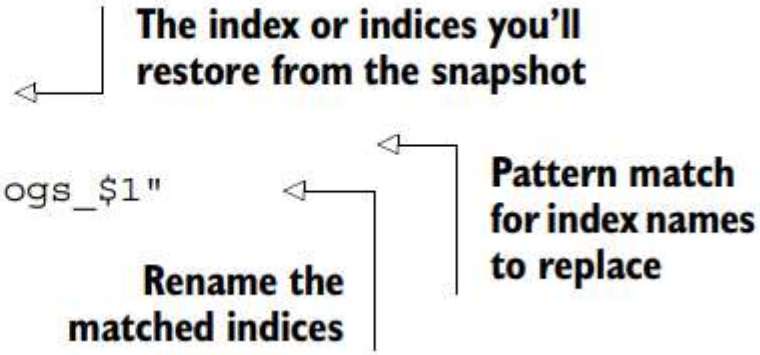
Restoring from backups

- By default, the restore HTTP request returns immediately, and the operation executes in the background:

```
curl -XPOST
'localhost:9200/_snapshot/my_repository/first_snapshot/
_restore?wait_for_completion=true';
```

Restoring from backups

```
curl -XPOST 'localhost:9200/_snapshot/my_repository/first_snapshot/_restore'
-d '
{
    "indices": "logs_2014",
    "rename_pattern": "logs_(.+)",
    "rename_replacement": "a_copy_of_logs_$1"
}';
```

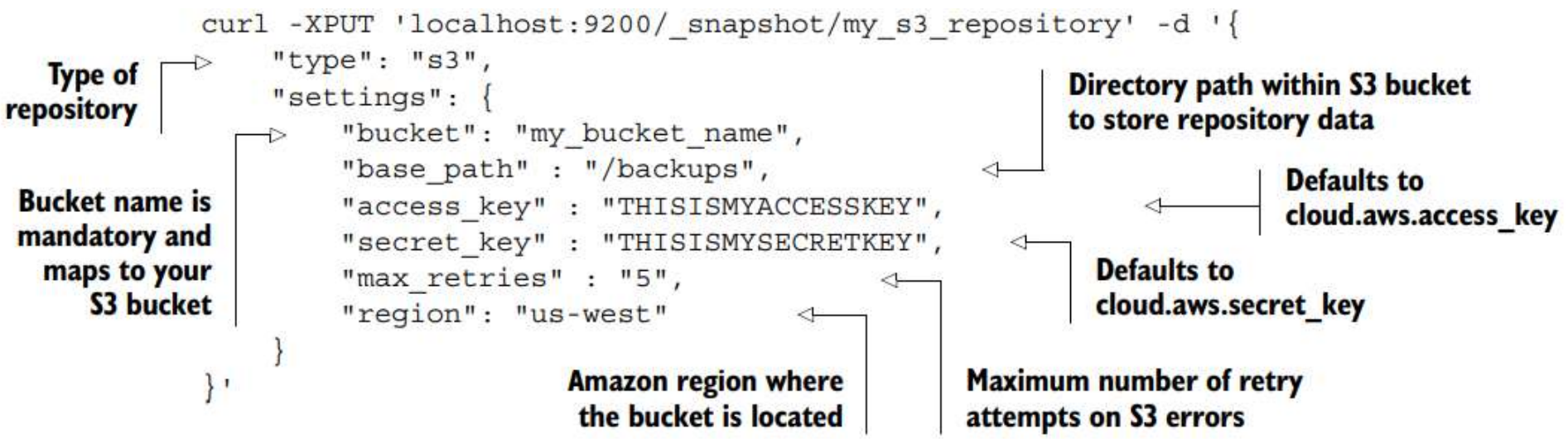


Using repository plugins

- Although snapshotting and restoring from a shared file system is a common use case, Elasticsearch and the community also provide repository plugins for several of the major cloud vendors.

- These plugins allow you to define repositories that use a specific vendor's infrastructure requirements and internal APIs.

AMAZON S3


curl -XPUT 'localhost:9200/_snapshot/my_s3_repository' -d '{
"type": "s3",
"settings": {
"bucket": "my_bucket_name",
"base_path" : "/backups",
"access_key" : "THISISMYACCESSKEY",
"secret_key" : "THISISMYSECRETKEY",
"max_retries" : "5",
"region": "us-west"
}
}'
Type of repository
Bucket name is mandatory and maps to your S3 bucket
Directory path within S3 bucket to store repository data
Defaults to cloud.aws.access_key
Defaults to cloud.aws.secret_key
Amazon region where the bucket is located
Maximum number of retry attempts on S3 errors

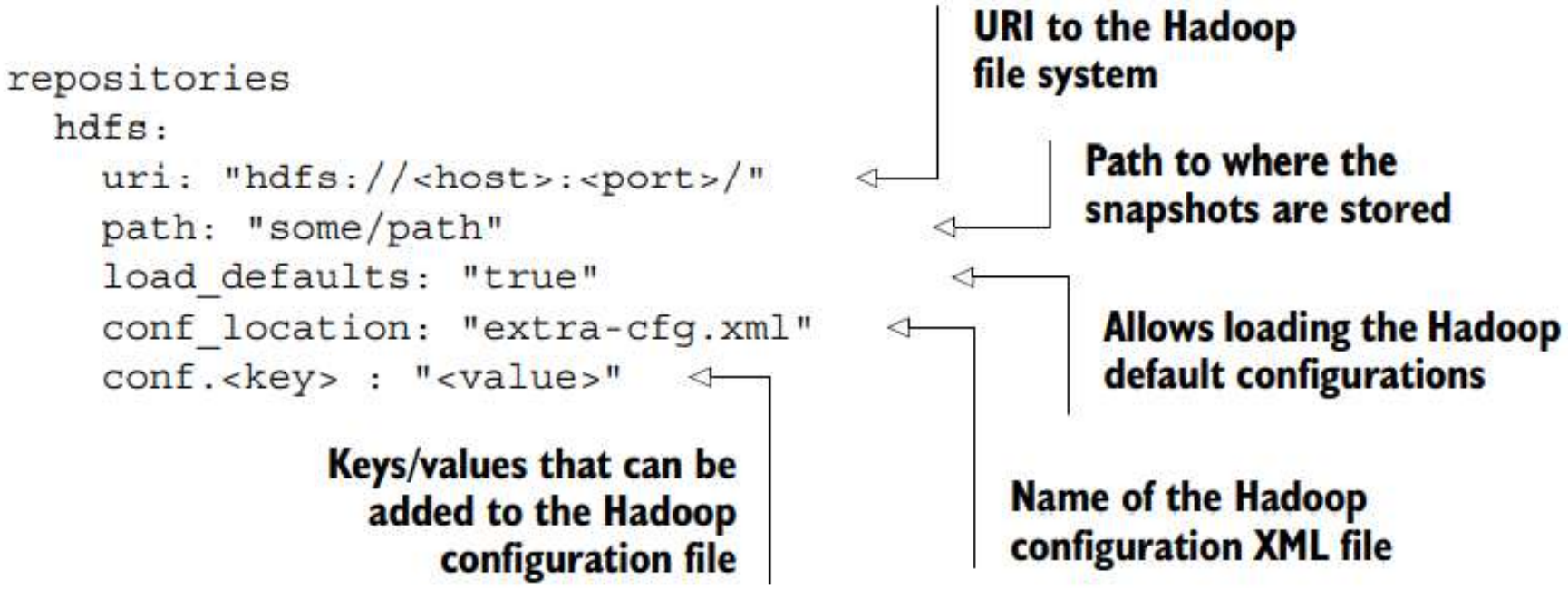
HADOOP HDFS

- You must install the latest stable release of this plugin on your Elasticsearch cluster.

- From the plugin directory, use the following command to install the desired version of the plugin directly from GitHub:

bin/plugin -i elasticsearch/elasticsearch-repository-hdfs/2.x.y

HADOOP HDFS


repositories
hdfs:
uri: "hdfs://<host>:<port>/"
path: "some/path"
load_defaults: "true"
conf_location: "extra-cfg.xml"
conf.<key> : "<value>"
URI to the Hadoop file system
Path to where the snapshots are stored
Allows loading the Hadoop default configurations
Name of the Hadoop configuration XML file
Keys/values that can be added to the Hadoop configuration file

Summary

- Index templates enable autocreation of indices that share common settings.
- Default mappings are convenient for the repetitive tasks of creating similar mappings across indices.
- Aliases allow you to query across many indices with a single name, thereby allowing you to keep your data segmented if needed.