Lab 11. Scaling out

This lab 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

Now that you have a good understanding of what Elasticsearch is capable of, you're ready to hear about Elasticsearch's next killer feature: the ability to scale—that is, to be able to handle more indexing and searching or to handle indexing and searching faster. These days, scaling is an important factor when dealing with mil- lions or billions of documents. You won't always be able to support the amount of traffic you'd like to on a single running instance of Elasticsearch, or *node*, without scaling in some form. Fortunately, Elasticsearch is easy to scale. In this lab we'll take a look at the scaling capabilities that Elasticsearch has at its disposal and how you can use those features to give Elasticsearch more performance and, at the same time, more reliability.

Having already seen how Elasticsearch handles the get-together data we introduced in labs 2 and 3, we're now ready to talk about how to scale your search sys-tem to handle all the traffic you can throw at it. Imagine you're sitting in your office,and in comes your boss to announce that your site has been featured in *Wired* maga-zine as the hot new site everyone should use for booking social get-togethers. Your job: make sure Elasticsearch can handle the influx of new groups and events, as well as allthe new searches expected to hit the site once that *Wired* article gets published! Youhave 24 hours. How are you going to scale up your Elasticsearch server to handle thistraffic in this time frame? Thankfully, Elasticsearch makes scaling a breeze by addingnodes to your existing Elasticsearch cluster.

9.1 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 par-allelization; you've run out of disk space on your machine, or perhaps your Elastic- search node is now running out of memory when performing queries against your data. In these cases, the easiest way to add performance to your Elasticsearch node is usually to turn it into an Elasticsearch cluster by adding more nodes, which you first learned about in lab 2. Elasticsearch makes it easy to scale horizontally by adding nodes to your cluster so they can share the indexing and searching workload. By add- ing nodes to your Elasticsearch cluster, you'll soon be able to handle indexing and searching the millions of groups and events headed your way.

9.1.1 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 develop-ment environment is as simple as extracting the Elasticsearch distribution to a sepa-rate directory, entering the directory, and running the `bin/elasticsearch` command, as the following code snippet shows. Elasticsearch will automatically pick the next port available to bind to—in this case, 9201—and automatically join the existing node like magic! If you want to go one step further, there's no need to even extract the Elastic-search distribution again; multiple instances of Elasticsearch can run from the same directory without interfering with one another:

```
% bin/elasticsearch

[in another terminal window or tab]

% cd /elasticstack/
% mkdir elasticsearch2
% cp -r elasticsearch-7.12.1/
./elasticsearch2/
% cd elasticsearch2/elasticsearch-
7.12.1/bin/
% ./elasticsearch
```

The originally runningElasticsearch node from lab 2

The newly started Elasticsearch node

Now that you have a second Elasticsearch node added to the cluster, you can run the `health` command from before and see how the status of the cluster has changed, as shown in the following listing.

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.

There are now no unassigned shards in this cluster, as you can see from the `unassigned_shards` count, which is zero. How exactly did the shards end up on the other node? Take a look at figure 9.1 and see what happens to the test index before and after adding a node to the cluster. On the left side, the primary shards for the testindex have all been assigned to `Node1`, whereas the replica shards are unassigned. In

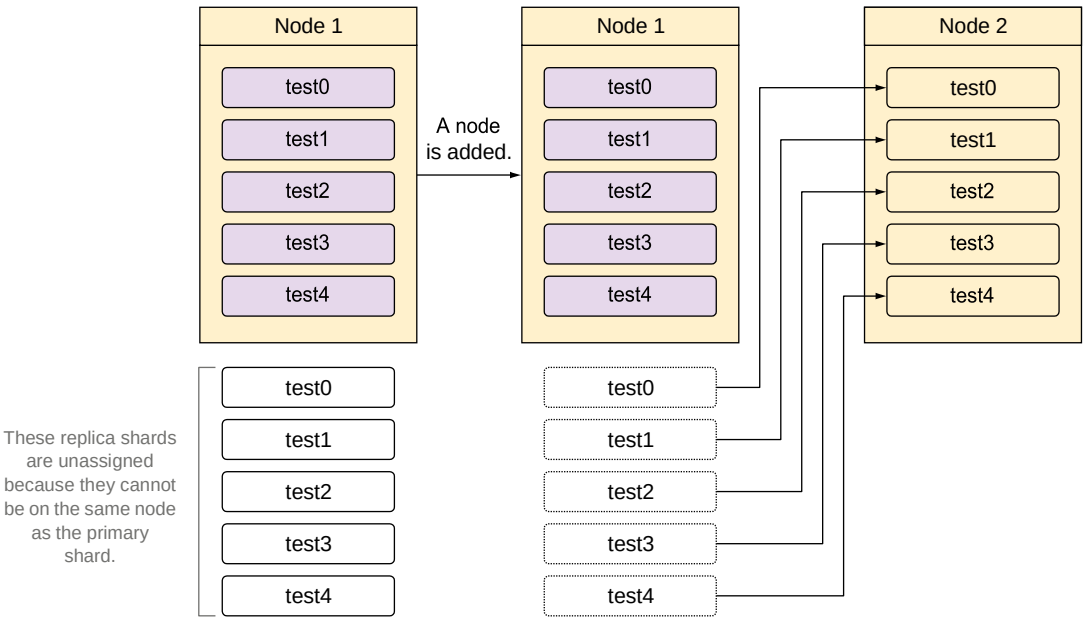

Figure 9.1 Shard allocation for the test index for one node transitioning to two nodes

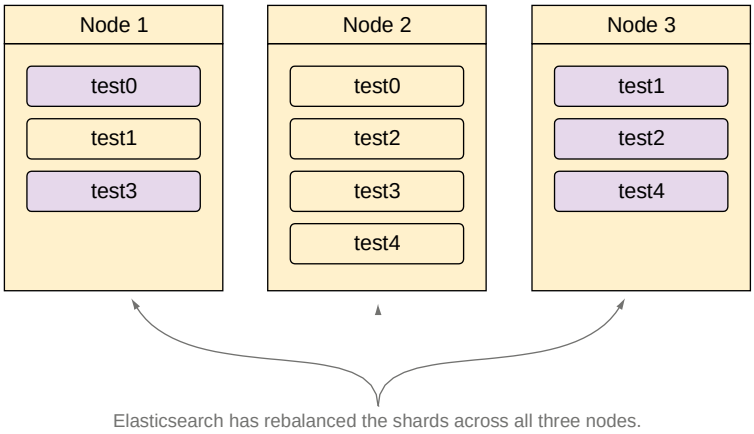

Figure 9.2 Shard allocation for the test index with three Elasticsearch nodes

this state, the cluster is yellow because all primary shards have a home, but the replica shards don't. Once a second node is added, the unassigned replica shards are assigned to the new `Node2`, which causes the cluster to move to the green state.

When another node is added, Elasticsearch will automatically try to balance out the shards among all nodes. Figure 9.2 shows how the same shards are distributed across three Elasticsearch nodes in the same cluster. Notice that there's no ban on having primary and replica shards on the same node as long as the primary and replica shards for the same shard number aren't on the same node.

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!

Adding nodes to your Elasticsearch cluster comes with substantial benefits, the primary being high availability and increased performance. When replicas are enabled (which they are by default), Elasticsearch will automatically promote a replica shard to a primary in the event the primary shard can't be located, so even if you lose the node where the primary shards for your index are, you'll still be able to access the data in your indices. This distribution of data among nodes also increases performance because search and get requests can be handled by both primary and replica shards, as you'll recall from figure 2.9. Scaling this way also adds more memory to the cluster as a whole, so if memory-intensive searches and aggregations are taking too long or causing your cluster to run out of memory, adding more nodes is almost always an easy way to handle more numerous and complex operations.

Now that you've turned your Elasticsearch node into a true cluster by adding a node, you may be wondering how each node was able to discover and communicate

with the other node or nodes. In the next section, we'll talk about Elasticsearch's node discovery methods.

9.2 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, Elastic- search 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.

9.2.1 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, so 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 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.

Generally, multicast discovery is a decent option when dealing with very flexible clusters on the same network, where the IP address of nodes being added changes fre- quently. Think of *multicast discovery* as shouting "Hey, are there any other nodes out there running an Elasticsearch cluster named 'xyz'?" and then waiting for a response. Figure 9.3 shows what multicast discovery looks like graphically.

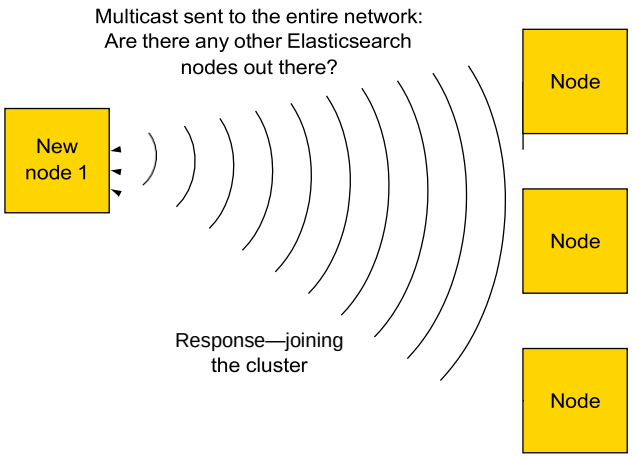

Figure 9.3 Elasticsearch using multicast discovery to discover other nodes in the cluster

Although multicast discovery is great for local development and a quick proof-of-concept test, when developing a production cluster, a more stable way of having Elasticsearch discover other nodes is to use some or all of the nodes as "gossip routers" to discover more information about the cluster. This can prevent the situation where nodes accidentally connect to a cluster they shouldn't have when someone connects a laptop to the same network. Unicast helps combat this by not sending a message to everyone on a network but connecting to a specific list of nodes.

9.2.2 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. Uni-cast is used by telling Elasticsearch the IP address and, optionally, the port or range of ports for other nodes in the cluster. An example of a unicast configuration would be setting `discovery.zen.ping.unicast.hosts: ["10.0.0.3", "10.0.0.4:9300", "10.0.0.5[9300-9400]"]` inside elasticsearch.yml for the Elasticsearch nodes on your network. Not all of the Elasticsearch nodes in the cluster need to be present in the unicast list to discover all the nodes, but enough addresses must be configured for each node to know about a gossip node that's available. For example, if the first node in the unicast list knows about three out of seven nodes in a cluster, and the second node in the unicast list knows about the other four out of the seven nodes, the node performing the discovery will still be able to find all seven nodes in the cluster. Figure 9.4 shows a graphical representation of unicast discovery.

There's no need to disable unicast discovery. If you'd like to use only multicast discovery to find other Elasticsearch nodes, leave the list unset (or empty) in the configuration file. After discovering other nodes that are part of the cluster, the Elasticsearch nodes will hold a master election.

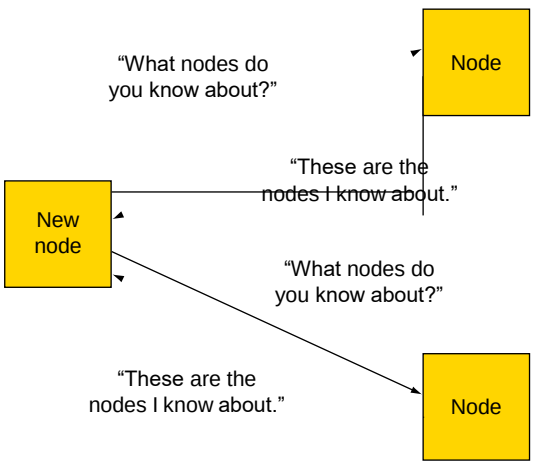

Figure 9.4 Elasticsearch using unicast discovery to discover other nodes in the cluster

9.2.3 Electing a master node and detecting faults

Once the nodes in your cluster have discovered each other, they'll negotiate whobecomes the master. The master node is in charge of managing the *state* of the clus- ter—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*, which we'll talk more about at the end of this section. Elasticsearch consid- ers all nodes eligible to become the master node unless the `node.master` setting is set to `false`. We'll talk more in this lab about why you may want to set the `node.master` setting, and the different types of Elasticsearch nodes, when we talk about how to search faster. In the event that your cluster has only a single node, that node will elect itself as the master after a timeout period if it doesn't detect anyother nodes in the cluster.

For production clusters with more than a couple of nodes, it's a good idea to set the minimum number of master nodes. Although this setting may make it seem like Elasticsearch can have multiple master nodes, it actually tells Elasticsearch how many nodes in a cluster must be eligible to become a master before the cluster is in a healthy state. Setting the minimum number of eligible master nodes can be helpful in making sure your cluster doesn't try to perform potentially dangerous operations without first having a complete view of the state of your cluster. You can either set the minimum number to the total number of nodes in your cluster if the number of nodes doesn't change over time or set it according to a common rule, which is the number of nodes in your cluster divided by 2, plus 1. Setting the `minimum_master_nodes` setting to a number higher than 1 can help prevent what's called a *split brain* in the cluster. Following the common rule for a three-node cluster, you'd set `minimum_master_nodes` to `2`, or for a 14-node cluster, you'd set the value to `8`. To change this setting, change `discovery.zen.minimum_master_nodes` in elasticsearch.yml to the number that fits your cluster.

What's a split brain?

The term *split brain* describes a scenario where (usually under heavy load or network issues) one or more of the nodes in your Elasticsearch cluster loses communication to the master node, elects a new master, and continues to process requests. At this point, you may have two different Elasticsearch clusters running independently of each other—hence the term *split brain*, because a single cluster has split into two distinct parts, similar to the hemispheres in a brain. To prevent this from happening, you should set `discovery.zen.minimum_master_nodes` depending on the number of nodes in your cluster. If the number of nodes won't change, set it to the total num- ber of nodes in the cluster; otherwise the number of nodes divided by 2 plus 1 is a good setting, because that means that if one or two nodes lose communication to the other nodes, they won't be able to elect a new master and form a new cluster because they don't meet the required number of master-eligible nodes.

Once your nodes are up and have discovered each other, you can see what node your cluster has elected as master by using the `curl` command shown in the following listing.

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

9.2.4 Fault detection

Now that your cluster has two nodes in it, as well as an elected master node, it needs to communicate with all nodes in the cluster to make sure everything is okay within the cluster; this is called the *fault detection* process. The master node pings all other nodes in the cluster and each node pings the master to make sure an election doesn't need to be held, as shown in figure 9.5.

As the figure shows, each node sends a ping every `discovery.zen.fd.ping_interval` (defaulting to `1s`), waits for `discovery.zen.fd.ping_timeout` (defaulting to `30s`), and

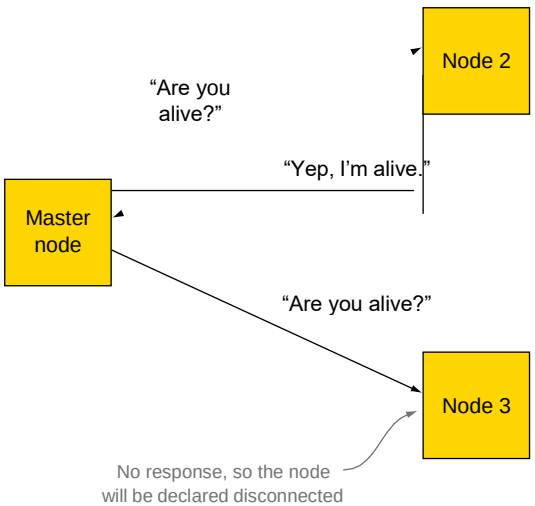

Figure 9.5 Cluster fault detection by the master node

tries a maximum number of `discovery.zen.fd.ping_retries` times (defaulting to `3`) before declaring a node disconnected and routing shards or holding a master election as necessary. Be sure to change these values if your environment has higher latency—say, when running on ec2 nodes that may not be in the same Amazon AWS zone.

Inevitably, one of the nodes in your cluster will go down, so in the next section, we'll talk about what happens when nodes are removed from the cluster and how to remove nodes without causing data loss in a distributed system.

9.3 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 in figure 9.2, containing the test index with five primary shards and one replica spread across the three nodes.

Let's say Joe, the sys admin, accidentally trips over the power cord for `Node1`; what happens to the three shards currently on `Node1`? The first thing that Elasticsearch does is automatically turn the `test0` and `test3` replica shards that are on `Node2` into primary shards, as shown in figure 9.6. This is because indexing first goes to the pri- mary shards, so Elasticsearch tries hard to make sure there are always primaries assigned for an index.

> **NOTE** Elasticsearch can choose any of the replicas to turn into a primary shard. It just so happens in this example that there's only one replica for each primary shard to choose from: the replicas on `Node2`.

After Elasticsearch turns the replicas for the missing primary shards into primaries, the cluster looks like figure 9.6.

After turning the replica shards into primaries, the cluster is now in a yellow state, meaning that some replica shards aren't allocated to a node. Elasticsearch next needs

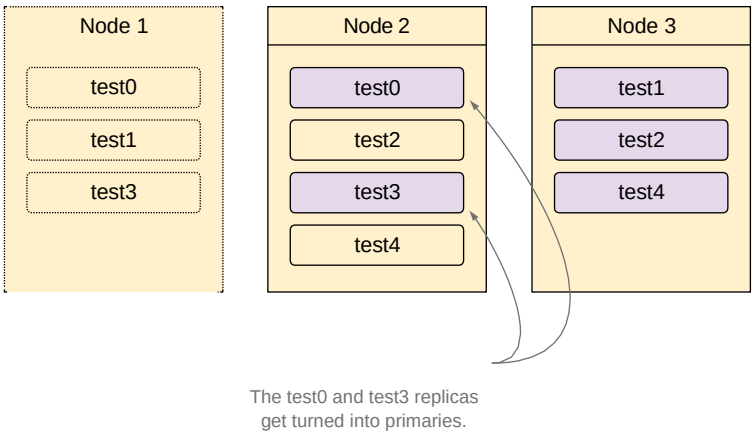

Figure 9.6 Turning replica shards into primaries after node loss

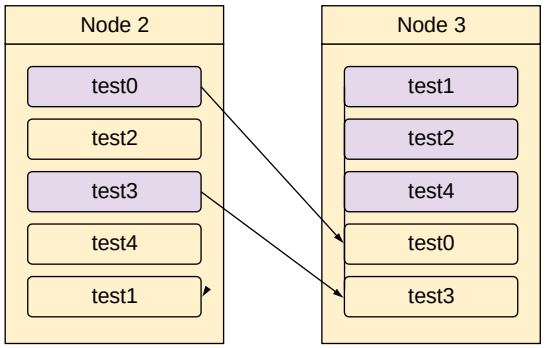

Figure 9.7 Re-creating replica shards after losing a node

to create more replica shards to maintain the high-availability setup for the test index. Because all the primaries are available, the data from the `test0` and `test3` primary shards on `Node2` will be replicated into replicas on `Node3`, and the data from the `test1` primary shard on `Node3` will be replicated onto `Node2`, as shown in figure 9.7.

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. If more than a single node is lost or a shard with no replicas is lost, the cluster will be in a red state, meaning that some amount of data has been lost permanently, and you'll need to either reconnect the node that has the data to the cluster or re-index the data that's missing.

It's important to understand how much risk you're willing to take with regard to the number of replica shards. Having a single replica means that one node can disappear from the cluster without data loss; if you use two replicas, two nodes can be lost without data loss, and so on, so make sure you choose the appropriate number of replicas for your usage. It's also always a good idea to back up your indices, which is a subject we'll cover in lab 11 when we talk about administering your cluster.

You've seen what adding and removing a node looks like, but what about shutting down a node without having the cluster go into a yellow state? In the next section we'll talk about decommissioning nodes so that they can be removed from the cluster with no interruption to the cluster users.

9.3.1 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. You could always stop the node by killing the Java process, and Elasticsearch would recover the data to the other nodes, but

what about when you have zero replicas for an index? That means you could lose data if you were to shut down a node without moving the data off first!

Thankfully, Elasticsearch has a way to decommission a node by telling the cluster not to allocate any shards to a node or set of nodes. In our three-node example, let's assume that `Node1`, `Node2`, and `Node3` have the IP addresses of 192.168.1.10, 192.168.1.11, and 192.168.1.12, respectively. If you wanted to shut down `Node1` while keeping the cluster in a green state, you could decommission the node first, which would move all shards on the node to other nodes in the cluster. 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.

Once you run this command, Elasticsearch will start moving all the shards from the decommissioned node to other nodes in the cluster. You can check where shards are located in the cluster by first determining the ID of the nodes in the cluster with the `_nodes` endpoint and then looking at the cluster state to see where each shard in the cluster is currently allocated. See the next listing for example output of these commands.

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" : {
          ...
        }
      }
    }
  },

  "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"
```

Retrieving a filtered cluster state

Shortened to fit on this page

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" : [ ]
}.
```

This is a long and ugly listing! Don't worry, though; later on this lab we'll talk about a more human-readable version of this API called the _cat API.

Here you can see that there are no shards on the `lFd3ANXiQlug-0eJztvaeA` node, which is the 192.168.1.10 node that was decommissioned, so it's now safe to stop ES on that node without causing the cluster to leave a green state. This process can be repeated one node at a time to decommission each node you want to stop, or you can use a list of comma-separated IP addresses instead of 192.168.1.10 to decommission multiple nodes at once. Keep in mind, however, that the other nodes in the cluster must be able to handle allocating the shard in terms of disk and memory use, so plan accordingly to make sure you have enough headroom before decommissioning nodes!

How much data can an Elasticsearch index handle?

Good question! Unfortunately, the limitations of a single index depend on the type of machine used to store the index, what you're planning to do with the data, and how many shards the index is backed by. Generally, a Lucene index (also known as an Elasticsearch shard) can't have more than 2.1 billion documents or more than 274 billion distinct terms (see https://lucene.apache.org/core/4_9_0/core/org/apache/lucene/codecs/lucene49/package-summary.html#Limitations), but you may be limited in disk space before this point. The best way to tell whether you'll be able to store your data in a single index is to try it out in a nonproduction system, adjusting settings as needed to get the performance characteristics desired. You can't change the number of primary shards once an index has been created; you can only change the number of replica shards, so plan accordingly!

Now that you've seen how nodes are added and removed from the cluster, let's talk about how to upgrade Elasticsearch nodes.

9.4 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.

Upgrade caveats

Before we get to upgrading instructions, it's important to understand that there are some limitations when upgrading Elasticsearch instances. Once you've upgraded an Elasticsearch server, the server can't be downgraded if any new documents have been written. When you perform upgrades to a production instance, you should always back up your data before performing an upgrade. We'll talk more about backing up your data in lab 11.

Another important thing to consider is that although Elasticsearch can handle a mixed-version environment easily, there have been cases where different JVM versions serialize information differently, so we recommend that you not mix different JVM versions within the same Elasticsearch cluster.

The simplest way to upgrade an Elasticsearch cluster is to shut down all nodes and then upgrade each Elasticsearch installation with whatever method you originally used—for example, extracting the distribution if you used the .tar.gz distribution or installing the .deb package using dpkg if you're using a Debian-based system. Once each node has been upgraded, you can restart the entire cluster and wait for Elastic- search to reach the green state. Voila, upgrade done!

This may not always be the case, though; in many situations downtime can't be tolerated, even during off-peak hours. Thankfully, you can perform a rolling restart to upgrade your Elasticsearch cluster while still serving indexing and searching requests.

9.4.1 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. This process is slightly more involved than a full restart because of the multiple steps required.

The first step in performing a rolling restart is to decide if you want Elasticsearch to automatically rebalance shards while each individual node is not running. The majority of people don't want Elasticsearch to start its automatic recovery in the event a node leaves the cluster for an upgrade because it means that they'll be rebalancing

every single node. In reality the data is still there; the node just needs to be restarted and to rejoin the cluster in order for it to be available.

For most people, it makes sense not to shift data around the cluster while performing the upgrade. You can accomplish this by setting the `cluster.routing.allocation.enable` setting to `none` while performing the upgrade. To clarify, the entire process looks like this:

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.

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.

Once you run this command, Elasticsearch will no longer rebalance shards around the cluster. For instance, if a primary shard is lost for an index because the node it resided on is shut down, Elasticsearch will still turn the replica shard into a new pri- mary, but a new replica won't be created. While in this state, you can safely shut down the single Elasticsearch node and perform the upgrade.

After upgrading the node, make sure that you reenable allocation for the cluster; otherwise you'll be wondering why Elasticsearch doesn't automatically replicate your data in the future! 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.

You need to perform these two book-ending steps, disabling allocation and reenabling allocation, for every node in the cluster being upgraded. If you were to perform them only once at the beginning and once at the end, Elasticsearch wouldn't allocate the shards that exist on the upgraded node every time you upgraded a node, and your cluster would be red once you upgraded multiple nodes. By reenabling allocation and waiting for the cluster to return to a green state after each node is

upgraded, your data is allocated and available when you move to the next node that needs to be upgraded. Repeat these steps for each node that needs to be upgraded until you have a fully upgraded cluster.

There's one more thing to mention in this section, and that's indices that don't have replicas. The previous examples all take into account the data having at least a single replica so that a node going down doesn't remove access to the data. If you have an index that has no replicas, you can use the decommissioning steps we covered in section 9.3.1 to decommission the node by moving all the data off it before shutting it down to perform the upgrade.

9.4.2 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. A larger amount of data may be copied in the event that the documents are the same. Until Elasticsearch has a way of verifying the last document written in a segment file, it has to copy over any differing files when replicating data between the primary shard and the replica shard.

There are two different ways to make segment files identical on the primary and replica shards. The first is using the optimize API that we'll talk about in lab 10 tocreate a single, large segment for both the primary and the replica. The second is to tog-gle the number of replicas to 0 and then back to a higher number; this ensures that allreplica copies have the same segment files as the primary shard. This means that for ashort period you'll have only a single copy of the data, so beware of doing this in a pro- duction environment!

Finally, in order to minimize recovery time, you can also halt indexing data into the cluster while you're performing the node upgrade.

Now that we've covered upgrading a node, let's cover a helpful API for getting information out of the cluster in a more human-friendly way: the _cat API.

9.5 Using the _cat API

Using the `curl` commands in sections 9.1, 9.2, and 9.3 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!). This is where the handy _cat API comes in. The _cat API provides helpful diagnostic and debugging tools that print data in a more human-readable way, ratherthan trying to page through a giant JSON response. The following listing shows

two of its commands for the equivalent health and node listing cURL statements we already covered.

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

% 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" : { }
    }
  }
}

% 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
```

Checking cluster health using the cluster health API

Checking cluster health using the _cat API

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

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

In addition to the `health` and `nodes` endpoints, the _cat API has many other features, all of which are useful for debugging different things your cluster may be undergoing. You can see the full list of supported _cat APIs by running `curl 'localhost:9200/_cat'`.

_cat APIs

At the time of this writing, here are some of the most useful _cat APIs and what they do. Be sure to check out the others!

- allocation—Shows the number of shards allocated to each node
- count—Counts the number of documents in the entire cluster or index
- health—Displays the health of the cluster
- indices—Displays information about existing indices
- master—Shows what node is currently elected master
- nodes—Shows various information about all nodes in the cluster
- recovery—Shows the status of ongoing shard recoveries in the cluster
- shards—Displays count, size, and names of shards in the cluster
- plugins—Displays information about installed plugins

While we're looking at adding nodes to a cluster, why not look at how the shards are distributed across each node using the _cat API in the following code listing. This is a much easier way to see how shards are allocated in your cluster as opposed to the `curl` command in listing 9.2.

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
```

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

```
% 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
```

Using the `_cat/allocation` and `_cat/shards` APIs is also a great way to determinewhen a node can be safely shut down after performing the decommission we dis- cussed in section 9.3.1. Compare the output of the `curl` command from listing 9.2 to the output from the commands in listing 9.6; it's much easier to read the _cat API output!

Now that you can see where the shards are located in your cluster, we should spend some more time discussing how you should plan your Elasticsearch cluster to make the most of your nodes and data.

9.6 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. In general, though, there are at least three things you'll want to consider when planning for a production Elasticsearch cluster: over-sharding, splitting data between indices and shards, and maximizing throughput.

9.6.1 Over-sharding

Let's start by talking about over-sharding. *Over-sharding* is the process whereby you intentionally create a larger number of shards for an index so you have room to add nodes and grow in the future; this is best illustrated by a diagram, so take a look at figure 9.8.

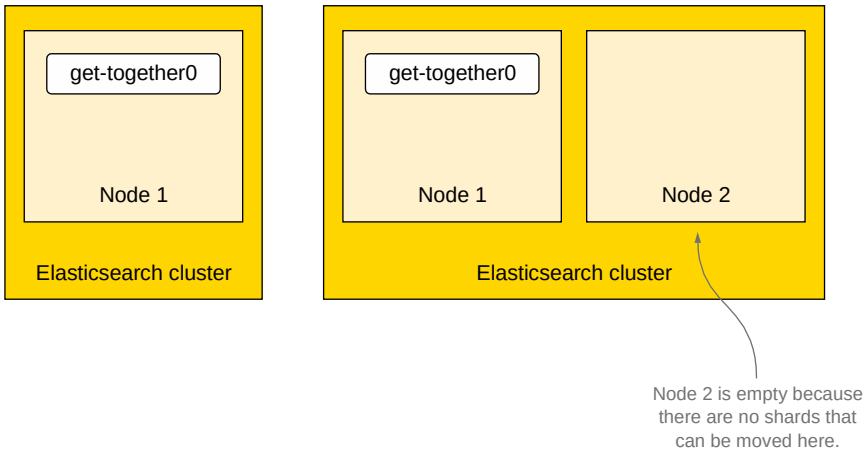

Figure 9.8 A single node with a single shard and two nodes trying to scale a single shard

In figure 9.8, you've created your get-together index with a single shard and no replicas. But what happens when you add another node?

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. Because a shard is the smallest thing that Elasticsearch can move around, it's a good idea to always make sure you have at least as many primary shards in your cluster asyou plan to have nodes; if you currently have a 5-node cluster with 11 primary shards, you have room to grow when you need to add more nodes to handle additional

requests. Using the same example, if you suddenly need more than 11 nodes, you won't be able to distribute the primary shards across nodes because you'll have more nodes than shards.

That's easy to fix, you might say: "I'll just create an index with 100 primary shards!" It may seem like a good idea at first, but there's a hidden cost to each shard Elasticsearch has to manage. Because each shard is a complete Lucene index, as you learned in lab 1, each shard requires a number of file descriptors for each segment of theindex, as well as a memory overhead. By creating too large a number of shards for an index, you may be using memory that could be better served to bolster perfor- mance, or you could end up hitting the machine's file descriptor or RAM limits. In addition, when compressing your data, you'll end up splitting the data across 100 dif-ferent things, lowering the compression rate you would have gotten if you had picked a more reasonable size.

It's worth noting that there is no perfect shard-to-index ratio for all use cases; Elasticsearch picks a good default of five shards for the general case, but it's always important to think about how you plan on growing (or shrinking) in the future with regard to the number of shards you create and index with. Don't forget: once an index has been created with a number of shards, the number of primary shards can never be changed for that index! You don't want to be in the position of having to re-index a large portion of your data six months down the line because there wasn't enough planning up front. We'll also talk more about this in the next lab when we discuss indexing in depth.

Along the same lines as choosing the number of shards to create an index with, you'll also need to decide on how exactly to split your data across indices in Elasticsearch.

9.6.2 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. For example, if you expect to have a larger number of events in New York than Sacramento, you could create a sacramento index with two primary shards and a newyork index with four primary shards, or you could segment the data by date, creating an index for each year an event occurs or is created: 2014, 2015, 2016, and so on. Segmenting data in this way can also be helpful when searching because the segmentation is handled by putting the right data in the right place; if the customer wants to search only for events or groups from the year 2014 or 2015, you'll have to search only those indices rather than the entire get-together index.

Another way to plan using indices is with aliases. An *alias* acts like a pointer to an index or a set of indices. An alias also allows you to change the indices that it points to at any time. This is incredibly useful for segmenting your data in a semantic way; you

could create an alias called last-year that points to 2015; then, when January 1, 2016 rolls around, you can change the alias to point to the 2015 index. This technique is commonly used when indexing date-based information (like log files) so that data can be segmented by date on a monthly/weekly/daily basis and an alias named current can be used to always point to the data that should be searched without having to change the name of the index being searched every time the segment rolls over. Again, aliases allow an incredible level of flexibility and have almost zero overhead, so experimentation is encouraged. We'll talk in more depth about aliases later on in this lab.

When creating indices, don't forget that because each index has its own shards, you'll still incur the overhead of creating a shard, so make sure not to create too many shards by creating too many indices and using resources that could be better spent handling requests. Once you know how your data will be laid out in the cluster, you can work on tweaking the node configuration to maximize your throughput.

9.6.3 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 Elastic- search 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? One way to make indexing faster is to temporarily reduce the number of replica shards in your cluster. When indexing data, by default the request won't complete until the data exists on the primary shard as well as all replicas, so it may be advantageous to reduce the number of replicas to one (or zero if you're okay with the risk) while indexing and then increase the number back to one or more once the period of heavy indexing has completed.

What about searches? Searches can be made faster by adding more replicas because either a primary or a replica shard can be used to search on. To illustrate this, check out figure 9.9, which shows a three-node cluster where the last node can't help with search requests until it has a copy of the data.

But don't forget that creating more shards in an Elasticsearch cluster does come with the small overhead in file descriptors and memory. If the volume of searches is getting too high for the nodes in the cluster to keep up, consider adding nodes with `node.data` and `node.master` both set to `false`. These nodes can then be used to handle incoming requests, distribute the request to the data nodes, and collect the results for responses. This way, the nodes searching the shards don't have to handle connec- tions from search clients; they only need to search shards. We'll talk more about dif- ferent ways of speeding up both indexing and searching in the next lab.

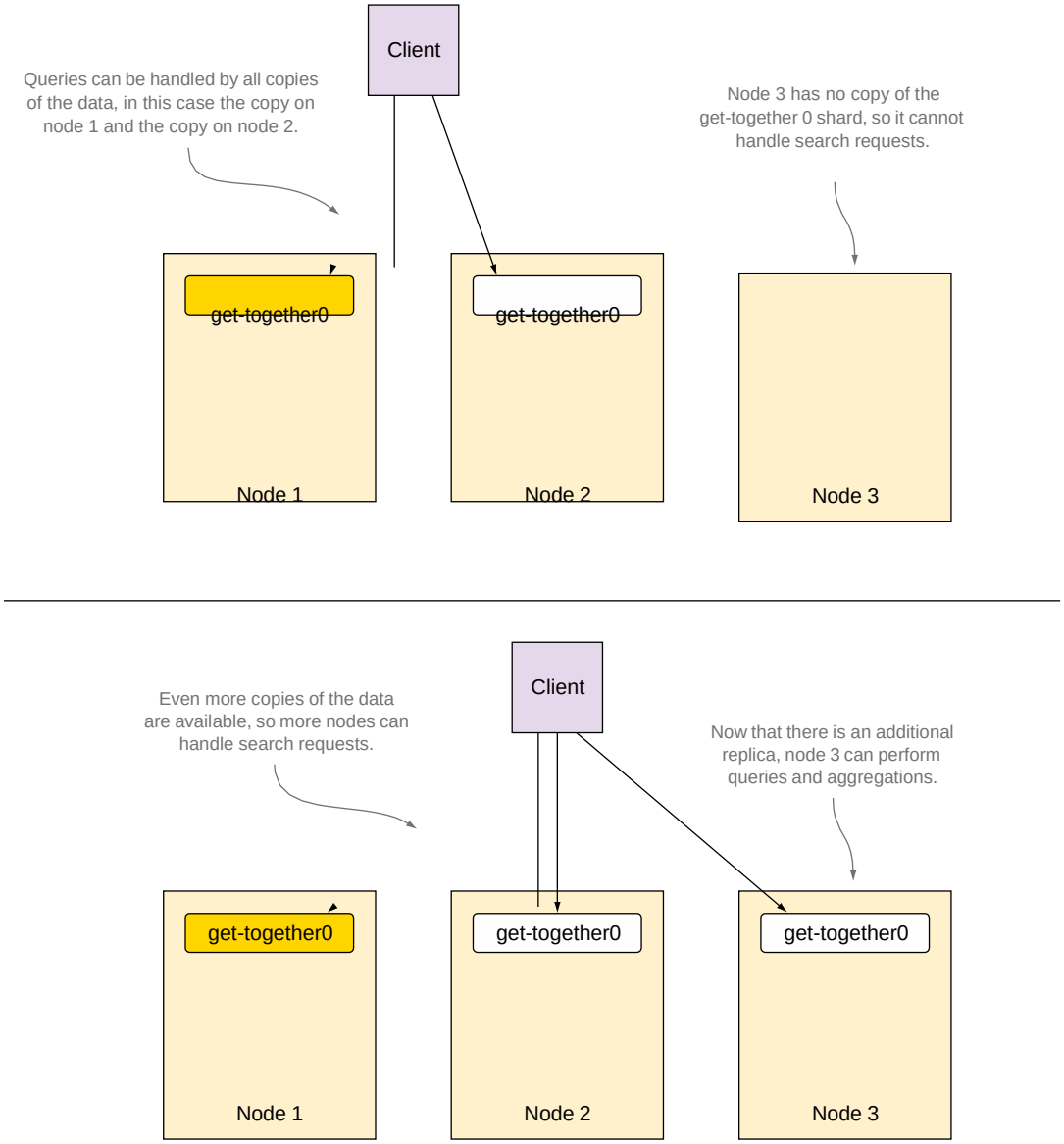

Figure 9.9 Additional replicas handling search and aggregations

9.7 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. Even when using an Elasticsearch cluster with only a single index, use an alias. You'll thank us later for the flexibility it will give you.

9.7.1 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. This means that compared to additional indices, aliases are much lighter in weight; thousands of them can be main- tained without negatively impacting your cluster. That said, we would caution against creating hundreds of thousands or millions of aliases because at that point, even the minimal overhead of a single entry in a map can cause the cluster state to grow to alarge size. This means operations that create a new cluster state will take longer becausethe entire cluster state is sent to each node every time it changes.

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. If you were using an alias for the original index, you can now change that alias to point to an additionally created index without having to change the name of the index you're searching (assuming you're using an alias for searching from the beginning).

Another useful feature can be creating windows into different indices; for example, if you create daily indices for your data, you may want a sliding window of the last week's data by creating an alias called last-7-days; then every day when you create a new daily index, you can add it to the alias while simultaneously removing the eight- day-old index.

MANAGING ALIASES

Aliases are created using the dedicated aliases API endpoint and a list of actions. Each action is a map with either an add or remove action followed by the index and alias on which to apply the operation. This will be much clearer with the example shown in the next listing.

Listing 9.7 Adding and removing aliases

```
curl -XPOST 'localhost:9200/_aliases' -d'
{
  "actions": [
    {

      "add" : {
        "index": "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.

```
      }
    },
    {
      "remove": {
        "index": "old-get-together",
        "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.

In this 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. You can specify as many add and remove actions as you like. It's important to recognize that these actions will all occur atomically, which means in the previous example there'll be no moment of time in which the gt-alias alias points to both the get-together and old-get-together indices. Although the compound Alias API call we just discussed may suit your needs, it's important to note that individual actions can be performed on the Alias API, using the common HTTP methods that Elasticsearch has standardized on. For instance, the following series of calls would have the same effect as the compound actions call shown previously:

```
curl -XPUT 'http://localhost:9200/get-together/_alias/gt-alias'

curl -XDELETE 'http://localhost:9200/old-get-together/_alias/gt-alias'
```

While we're exploring single-call API methods, this section wouldn't be complete without covering the API in more detail, specifically those endpoints that can come in handy in creating and listing operations.

9.7.2 Alias creation

When creating aliases, there are many options available via the API endpoint. For instance, you can create aliases on a specific index, many indices, or a pattern that matches index names:

Index name, _all, a comma-delimited list of index names, or a pattern to match

The name of the alias you're creating

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

curl -XPUT 'http://localhost:9200/myindex/_alias/myalias'
curl -XPUT 'http://localhost:9200/_all/_alias/myalias'

curl -XPUT 'http://localhost:9200/logs-2013,logs-2014/_alias/myalias'
curl -XPUT 'http://localhost:9200/logs-*/_alias/myalias'
```

Create alias myalias on index myindex.

Create alias myalias on all indices.

Create alias myalias on both indices, logs-2013 and logs-2014.

Create alias myalias on all index names that match the pattern logs-*.

Alias deletion accepts the same path parameter format:

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

You can retrieve all of the aliases that a concrete index points to by issuing a `GET` request on an index with `_alias`, or you can retrieve all indices and the aliases that point to them by leaving out the index name. Retrieving the aliases for an index is shown in the next listing.

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.

In addition to the `_alias` endpoint on an index, you have a number of different ways to get the alias information from an index:

Index name, _all, a comma-delimited list of index names, a pattern to match, or can be left blank

The name of the alias you're retrieving. Can be either an alias name, a comma-delimited list, or a pattern to match against.

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

```
curl -XGET 'http://localhost:9200/myindex/_alias/myalias'
curl -XGET 'http://localhost:9200/myindex/_alias/*'
curl -XGET 'http://localhost:9200/_alias/myalias'
curl -XGET 'http://localhost:9200/_alias/logs-*'
```

Retrieve alias myalias for index myindex.

Retrieve all aliases for index myindex.

Retrieve all indices with alias myalias.

Retrieve all indices with aliases that match the pattern logs-*.

MASKING DOCUMENTS WITH ALIAS FILTERS

Aliases have some other neat features as well; they can be used to automatically apply a filter to queries that are executed. For example, with your get-together data it could be useful to have an alias that points only to the groups that contain the `elasticsearch` tag, so 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"}
        }
      }
    }
  ]
}'
```

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

```
{"acknowledged":true}

$ curl 'localhost:9200/get-together/group/_count' -d'
{

  "query": {
    "match_all": {}
  }
}'
{"count":5,"_shards":{"total":2,"successful":2,"failed":0}}
```

Counting all the groups in the get-together index

Five groups in the get-together index

```
$ curl 'localhost:9200/es-groups/group/_count' -d'
{
  "query": {
    "match_all": {}
  }
}'

{"count":2,"_shards":{"total":2,"successful":2,"failed":0}}
```

Counting all the groups in the es-groups alias

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

Here you can see that the es-groups alias contains only two groups instead of five. This is because it's automatically applying the `term` filter for groups that contain the tag `elasticsearch`. This has a lot of applications; if you're indexing sensitive data, for instance, you can create a filtered alias to ensure that anyone using that alias can't see data they're not meant to see.

There's one more feature that aliases can provide, routing, but before we talk about using it with an alias, we'll talk about using it in general.

9.8 Routing

In lab 8, we talked about how documents end up in a particular shard; this pro- cess 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. Elasticsearch also allows you to manually specify the routing of a document when indexing, which is what you do when using parent-child relationships because the child document has to be in the same shard as the parent document.

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"
}'
```

In this example, `denver` is the value that's hashed to determine which shard the document ends up in, instead of `9`, the document's ID. Routing can be useful for scaling strategies, which is why we talk about it in detail in this lab.

9.8.1 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 rout- ing value into a single shard, and once these documents are in the same index, it allows you to route certain queries so that they are executed on a subset of the shards for an index. Sound confusing? We'll go over it in more detail to clarify what we mean.

9.8.2 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. This is a good choice for a routing value because the cities vary widely enough that you have quite a few values to pick from, and each event and group are already associated with a city, so it's easy to extract that from a document before indexing. If you were to pick something that had only a few different values, you could easily end up with unbalanced shards for the index. If there are only three possible routing values for all documents, all documents will end up routed between a maximum of three shards. It's important to pick a value that will have enough cardinality to spread data among shards in an index.

Now that you've picked what you want to use for the routing value, you need to specify this routing value when indexing documents, as shown in the listing that follows.

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

In this example, you use three different routing values—`denver`, `boulder`, and `amsterdam`—for three different documents. This means that instead of hashing the IDs `10`, `11`, and `12` to determine which shard to put the document in, you use the routing values instead. On the index side, this doesn't help you much; the real benefit comes when you combine routing on the query side, as the next listing shows. On the query side, you can combine multiple routing values with a comma.

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

Interesting! Instead of returning all three groups, only two were returned. So what actually happened? Internally, when Elasticsearch received the request, it hashed the values of the two provided routing values, `denver` and `amsterdam`, and then executed

the query on all the shards they hashed to. In this case `denver` and `amsterdam` both hash to the same shard, and `boulder` hashes to a different shard.

Extrapolate this to hundreds of thousands of groups, in hundreds of cities, by specifying the routing for each group both while indexing and while querying, and you're able to limit the scope of where a search request is executed. This can be a great scaling improvement for an index that might have 100 shards; instead of running the query on all 100 shards, it can be limited and thus run faster with less impact to your Elasticsearch cluster.

In the previous example, `denver` and `amsterdam` happen to route to the same shard value, but they could have just as easily hashed to different shard values. How can you tell which shard a request will be executed on? Thankfully, Elasticsearch has an API that can show you the nodes and shards a search request will be performed on.

9.8.3 Using the _search_shards API to determine where a search is performed

Let's take the prior example and use the search shards API to see which shards the request is going to be executed on, with and without the routing values, as shown in the following listing.

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

```
{
  "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.

You can see that even though there are two shards in the index, when the routing value `denver` is specified, only shard 1 is going to be searched. You've effectively cut the amount of data the search must execute on by half!

Routing can be useful when dealing with indices that have a large number of shards, but it's definitely not required for regular usage of Elasticsearch. Think of it as a way to scale more efficiently in some cases, and be sure to experiment with it.

9.8.4 Configuring routing

It can also be useful to tell Elasticsearch that you want to use custom routing for all documents and to refuse to allow you to index a document without a custom routing value. You can configure this through the mapping of a type. For example, to create an index called routed-events and required routing for each event, you can use the code in the following listing.

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"}'
```

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

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

Elasticsearch returns an error because required routing value is missing

There's one more way to use routing, and that's by associated a routing value with an alias.

9.8.5 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, Elasticsearch will return an error because it doesn't know which concrete index the document should be indexed into.

Reusing the previous example, you can create an alias called denver-events that automatically filters out events with "denver" in the name and adds "denver" to the routing when searching and indexing to limit where queries are executed, as shown in the next listing.

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"
      }
    }
  ]
}'
```

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.

```
{"acknowledged":true}

% 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" ]
      }
    } ]
  }
}
```

You can also use the alias you just created for indexing. When indexing with the denver-events alias, it's the same as if documents were indexed with the `routing=denver` query string parameter. Because aliases are lightweight, you can create as many as you need when using custom routing in order to scale out better.

9.9 Summary

You should now have a better understanding of how Elasticsearch clusters are formed and how they're made of multiple nodes, each containing a number of indices, which in turn are made up of a number of shards. Here are some of the other things we talked about in this lab:

- 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

In lab 10 we'll continue talking about scaling from the perspective of improving performance in your Elasticsearch cluster.