Lab 13. Administering your cluster

This lab covers

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

We've covered a lot of material in this course, and we hope you now feel comfortable working with the Elasticsearch APIs. In this lab you'll augment the APIs you've learned thus far and use them with the goal of monitoring and tuning your Elastic- search cluster for increasing performance and implementing disaster recovery.

Both developers and administrators will eventually be faced with the prospect of having to monitor and administer their Elasticsearch cluster. Whether your system is under high or moderate use, it will be important for you to understand and identify bottlenecks and be prepared in the event of a hardware or system failure.

This lab covers Elasticsearch cluster administration operations using the REST API, which you should now feel comfortable with, given the exposure through-out this course. This will enable you to identify and address possible performance bottlenecks, using real-time monitoring and best practices.

To that end, we'll cover three overarching topics: improving defaults, monitoring for problems, and effectively using the backup system, with the simple premise that effective performance monitoring is necessary for system optimization and that understanding your system will aid in the planning of disaster scenarios.

11.1 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.

Most uses of Elasticsearch in production environments may fall into the category of occasional full-text search, but a growing number of deployments are pushing formerly edge-case uses into more common installations, such as the growing trends of using Elasticsearch as a sole source of data, logging aggregators, and even using it in hybrid storage architectures where it's used in conjunction with other database types. These exciting new uses open the door for us to explore interesting ways in which to tune and optimize the Elasticsearch default settings.

11.1.1 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*—In this situation a daily log index is needed for efficient querying and storage, much as rolling log file appenders work. A common example of this is found in cloud-based deployments, where distributed systems push their logs onto a central Elasticsearch cluster. Configuring the cluster to handle automatic templating of log data by day helps organize the data and eases searching for the proverbial needle in the haystack.
- *Regulatory compliance*—Here blocks of data must be either kept or removed after a certain time period to meet compliance standards, as in financial sector companies where Sarbanes-Oxley compliance is mandated. These sorts of mandates require organized record keeping where template systems shine.
- *Multi-tenancy*—Systems that create new tenants dynamically often have a need to compartmentalize tenant-specific data.

Templates have their uses when a proven and repeatable pattern is needed for homogenous data storage. The automated nature of how Elasticsearch applies tem- plates is also an attractive feature.

Creating a template

As the name suggests, an index template will be applied to any new index created. Indices that match a predefined naming pattern will have a template applied to them, ensuring uniformity in index settings across all of the matching indices. 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:

- By way of the REST API
- By a configuration file

The former assumes a running cluster; the latter does not and is often used in predeployment scenarios that a dev ops engineer or system administrator would employ in a production environment.

In this section we'll illustrate a simple index template used for log aggregation, so your log aggregation tool will have a new index created per day. At the time of this writing, Logstash was the most popular log-aggregation tool used alongside Elasticsearch, and its integration was seamless, so focusing on Logstash-to-Elasticsearch index template creation makes the most sense.

By default, Logstash makes API calls using the daily timestamp appended to the index name; for example, logstash-11-09-2014. Assuming you're using the Elasticsearch default settings, which allow for automatic index creation, once Logstash makes a call to your cluster with a new event, the new index will be created with a name of logstash-11-09-2014, and the document type will be automapped. You'll use the REST API method first, as shown here:

```
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

Using the PUT command, you instruct Elasticsearch to apply this template whenever an index call matching the logstash-* pattern is received. In this case, when Logstash posts a new event to Elasticsearch and an index doesn't exist by the name given, a new one will be created using this template.

This template also goes a bit further in applying an alias, so you can group all of these indices under a given month. You'll have to rename the index manually each month, but it affords a convenient way to group indices of log events by month.

TEMPLATES CONFIGURED ON THE FILE SYSTEM

If you want to have templates configured on the file system, which sometimes makes it easier to manage maintenance, the option exists. Configuration files must follow these simple rules:

- Template configurations must be in JSON format. For convenience, name them with a .json extension: <FILENAME>.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.

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" : {} }
}
```

Much like defining via the REST API, now every index matching the `logstash-*` pat- tern will have this template applied.

Multiple template merging

Elasticsearch also enables you to configure multiple templates with different settings. You can then expand on the previous example and configure a template to handle log events by month and another that will store all log events in one index, as the following listing shows.

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.

In the previous example, the topmost template will be responsible for November-specific logs because it matches on the pattern of index names beginning with `"logstash-09-"`. The second template acts as a catchall, aggregating all log stash indices and even containing a different setting for the `date` mapping.

One thing to note about this configuration is the `order` attribute. This attribute implies that the lowest order number will be applied first, with the higher order number then overriding it. Because of this, the two templates settings are merged, with the effect of all November log events *not* having the `date` field stored.

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'
```

11.1.2 Default mappings

As you learned in lab 2, mappings enable you to define specific fields, their types,and even how Elasticsearch will interpret and store them. Furthermore, you learned how Elasticsearch supports dynamic mapping in lab 3, removing the need to define your mappings at index-creation time; instead those mappings are dynamically generated based on the content of the initial document you index. This section, muchlike the previous one that covered default index templates, will introduce you to the concept of specifying default mappings, which act as a convenience utility for repeti-tive mapping creation.

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.

Mapping is not retroactive

Note that specifying a default mapping doesn't apply the mapping retroactively. Default mappings are applied only to newly created types.

Consider the following example, where you want to specify a default setting for how you store the `_source` for all of your mappings, except for a `Person` type:

```
curl -XPUT 'localhost:9200/streamglue/_mapping/events' -d ' {
    "Person" :
    {
        "_source" : {"enabled" : false}
    },
    "_default_" :
    {"_source" : {"enabled" : true }
    }
}'
```

In this case, all new mappings will by default store the document `_source`, but any mapping of type `Person`, by default, will not. Note that you can override this behavior in individual mapping specifications.

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. You can alter this behavior by instructing Elasticsearch to ignore new fields or even throw exceptions on unknown fields. You'd normally want to restrict the new addition of fields to prevent data pollution and help maintain control over the schema definition.

DISABLING DYNAMIC MAPPING Note also that you can disable the dynamic creation of new mappings for unmapped types by setting `index.mapper.dynamic` to `false` in your elasticsearch.yml configuration.

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",     <- Throw exception if an unknown field is encountered at index time.
            "properties": {
                "email":  { "type": "string"},

                "created_date":  { "type": "date" }
            }
        }
    }
}'
```

```
curl -XPUT 'localhost:9200/second_index' -d
'{
    "mappings": {
        "person": {
            "dynamic":      "true",        <-- Allow the dynamic creation of new fields.
            "properties": {
                "email":  { "type": "string"},

                “created_date":  { "type": "date" }
            }
        }
    }
}'
```

The first mapping restricts the creation of new fields in the `person` mapping. If you attempt to insert a document with an unmapped field, Elasticsearch will respond with an exception and not index the document. 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

This section wouldn’t be complete if we didn’t cover how dynamic mapping and dynamic templates work together, allowing you to apply different mappings depending on the field name or datatype.

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. These are unique alphanumeric strings that contain hyphen separators, such as `"b20d5470-d7b4-11e3-9fa6-25476c6788ce"`. You don’t want Elasticsearch analyzing them with a default analyzer because it would split the UUID by hyphen when building the index tokens. You want to be able to search by the complete string UUID, so you need Elasticsearch to store the entire string as a token. In this case, you need to instruct Elasticsearch to not analyze any `string` field whose name ends in `"_guid"`:

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

Match field names ending in _guid.

Matched fields must be of type string.

Define the mapping you will apply when a match is made.

Set as type string.

Don't analyze these fields when indexing.

In this example, the dynamic template is used to dynamically map fields thatmatched a certain name and type, giving you more control over how your data is stored and made searchable by Elasticsearch. As an additional note, you can use the `path_match` or `path_unmatch` keyword, which allows you to match or unmatch the dynamic template using dot notation—for instance, if you wanted to match some- thing like `person.*.email`. Using this logic, you can see a match on a data structure such as this:

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

Dynamic templates are a convenient method of automating some of the more tedious aspects of Elasticsearch management. Next, we'll explore allocation awareness.

11.2 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. You can arm Elasticsearch with this knowledge so it intelligently distributes replica data across a cluster.

11.2.1 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. You can also experience faster read operations, as nodes deployed on the same physical rack will potentially have a proximity advantage of not having to go over the network.

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
```

NOTE The awareness attribute can be assigned more than one value. `cluster.routing.allocation.awareness.attributes: rack, group, zone`

Using the previous definition, you'll segment your shards across the cluster using the awareness parameter `rack`. You alter the elasticsearch.yml for each of your nodes, setting the value the way you want your network configuration to be. 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
```

A simple before-and-after illustration may help in this case. Figure 11.1 shows a cluster with the default allocation settings.

This cluster suffers from primary and replica shard data being on the same rack. With the allocation awareness setting, you can remove the risk, as shown in figure 11.2.

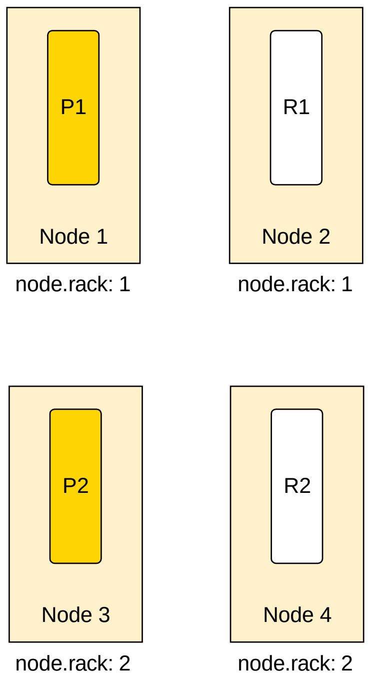

Figure 11.1 Cluster with default allocation settings

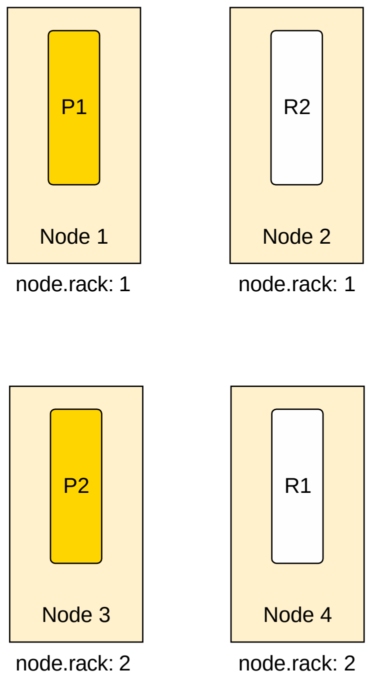

Figure 11.2 Cluster with allocation awareness

Wwith allocation awareness, the primary shards were not moved, but the replicas were moved to nodes with a different `node.rack` parameter value. Shard allocation is a convenient feature that insured against a central point of failure. A common use is separating cluster topology by location, racks, or even virtual machines.

Next, we'll take a look at forced allocation with a real-world AWS zone example.

11.2.2 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. The use case is simple: limit the number of replicas in one zone if another zone goes down or is unreachable. By doing this, you reduce the danger of overallocating replicas from another group.

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
```

Given these settings, let's play out this real-world scenario. Let's say you start a set of nodes in the East region with `node.zone: us-east`. You'll use the defaults here, leaving an index with five primary shards and one replica. Because there are no other zone values, only the primary shards for your indices will be allocated.

What you're doing here is limiting the replicas to balance only on nodes without your value. If you were to start up your West region cluster, with `node.zone: us-west`, replicas from `us-east` would be allocated to it. No replicas will ever exist for nodes defined as `node.zone: us-east`. Ideally, you'd do the same on `node.zone: us-west`, thereby ensuring that replicas never exist in the same location. Keep in mind that if you lose connectivity with `us-west`, no replicas will ever be created on `us-east`, or vice versa.

Allocation awareness does require some planning up front, but in the event that allocation isn't working as planned, these settings can all be modified at runtime using the Cluster Settings API. They can be persistent, where Elasticsearch applies thesettings 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
        }
}
```

Cluster allocation can make the difference between a cluster that scales and is resilient to failure and one that isn't.

Now that we've explored some of the finer adjustments that you can make to Elasticsearch default settings with shard allocation, let's look at how to monitor the general health of your cluster for performance issues.

11.3 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. Understanding cluster diagnostic data and being able to assess the overall status of the cluster will alert you to performance bottlenecks, such as unassigned shards and missing nodes, so you can easily address them.

11.3.1 Checking cluster health

The cluster health API provides a convenient yet coarse-grained view of the overall health of your cluster, indices, and shards. This is normally the first step in being alerted to and diagnosing a problem that may be actively occurring in your cluster. The next listing shows how to use the cluster health API to check overall cluster state.

Listing 11.3 Cluster health API request

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

And the response:

Cluster status indicator: convenient general health indicator of the cluster

Total number of nodes in the cluster

```
{
  "cluster_name" : "elasticiq",
  "status" : "green",
  "timed_out" : false,
  "number_of_nodes" : 2,
  "number_of_data_nodes" : 2,
  "active_primary_shards" : 10,
  "active_shards" : 10,
  "relocating_shards" : 0,
  "initializing_shards" : 0,
  "unassigned_shards" : 0
}
```

Total number of primary shards for all indices in the cluster

Total number of nodes that hold data in the cluster

Total number of all shards, primary and replica, for all indices in the cluster

Number of shards that are moving across nodes at the present time

Number of shards that are defined within the cluster state but can't be found

Number of shards that are newly created

Taking the response shown here at face value, you can deduce a lot about the general health and state of the cluster, but there's much more to reading this simple output than what's obvious at first glance. Let's look a little deeper into the meaning of the last three indicators in the code: `relocating_shards`, `initializing_shards`, and `unassigned_shards`.

- `relocating_shards`—A number above zero means that Elasticsearch is moving shards of data across the cluster to improve balance and failover. This ordinarily occurs when adding a new node, restarting a failed node, or removing a node, thereby making this a temporary occurrence.
- `initializing_shards`—This number will be above zero when you've just created a new index or restarted a node.
- `unassigned_shards`—The most common reason for this value to be above zero is having unassigned replicas. The issue is common in development environments, where a single-node cluster has an index defined as having the default, five shards and one replica. In this case, there'll be five unassigned replicas.

As you saw from the first line of output, the cluster status is green. There are times when this may not be so, as in the case of nodes not being able to start or falling away from the cluster, and although the status value gives you only a general idea of the health of the cluster, it's useful to understand what those status values mean for cluster performance:

- *Green*—Both primary and replica shards are fully functional and distributed.
- *Yellow*—Normally this is a sign of a missing replica shard. The `unassigned_shards` value is likely above zero at this point, making the distributed nature of the cluster unstable. Further shard loss could lead to critical data loss. Look for any nodes that aren't initialized or functioning correctly.
- *Red* —This is a critical state, where a primary shard in the cluster can't be found, prohibiting indexing operations on that shard and leading to inconsistentquery results. Again, likely a node or several nodes are missing from the cluster.

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
}
```

Given this API call and response, you see that the cluster is now in yellow status, and as you've already learned, the likely culprit is the `unassigned_shards` value being above `0`. The cluster health API provides a more fine-grained operation that will allow you to further diagnose the issue. In this case, you can look deeper at which indices are affected by the unassigned shards by adding the `level` parameter:

```
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

The single-node cluster is experiencing some problems because Elasticsearch is trying to allocate replica shards across the cluster, but it can't do so because there's only one node running. This leads to the replica shards not being assigned anywhere and therefore a yellow status across the cluster, as figure 11.3 shows.

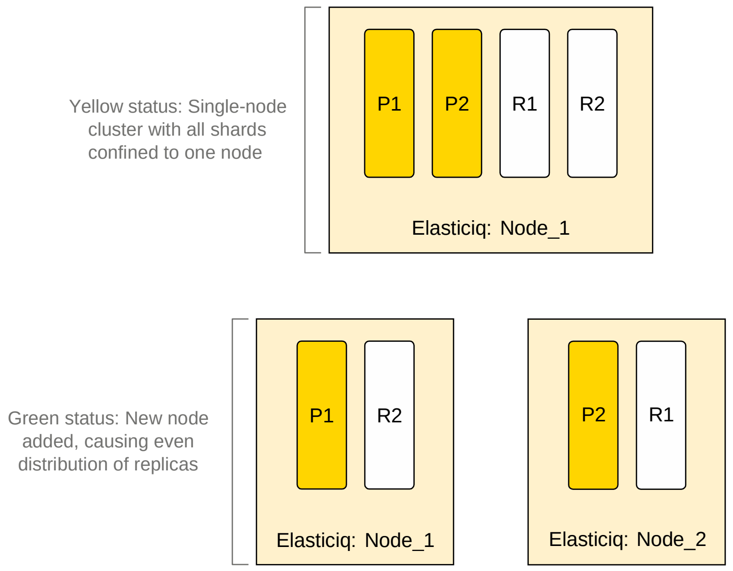
Yellow status: Single-node cluster with all shards confined to one node
P1
P2
R1
R2
Elasticiq: Node_1
Green status: New node added, causing even distribution of replicas
P1
R2
Elasticiq: Node_1
P2
R1
Elasticiq: Node_2

Figure 11.3 Yellow status solved by making nodes accessible

As you can see, an easy remedy is to add a node to the cluster so Elasticsearch can then allocate the replica shards to that location. Making sure that all of your nodes are running and accessible is the easiest way to solve the yellow status issue.

11.3.2 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. This section will help demystify some of these possible performance bottlenecks and provide you with the tools needed to identify and address these issues.

SLOW LOGS

Elasticsearch provides two logs for isolating slow operations that are easily configured within your cluster configuration file: slow query log and slow index log. By default, both logs are disabled. Log output is scoped at the shard level. That is, one operation can represent several lines in the corresponding log file. The advantage to shard-level logging is that you'll be better able to identify a problem shard and thereby a node with the log output, as shown here. 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
```

As you can see, you can set thresholds for both phases of a search: the query and the fetch. The log levels (warn, info, debug, trace) allow you finer control over which level will be logged, something that comes in handy when you simply want to grep your log file. 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"
```

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 QUERY LOG

The important parts you're interested in for identifying performance issues are the query times: `took[##ms]`. Additionally, it's helpful to know the shards and indices involved, and those are identifiable by the `[index][shard_number]` notation; in this case it's `[streamglue][4]`.

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
```

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[] ...
```

Discovering where your slow queries and index calls are happening will go a long way in helping remedy Elasticsearch performance problems. Allowing slow performance to grow unbounded can cause a cascading failure across your entire cluster, leading to it crashing entirely.

HOT_THREADS API

If you've ever experienced high CPU utilization across your cluster, you'll find the hot_threads API helpful in identifying specific processes that may be blocked and causing the problem. The hot_threads API provides a list of blocked threads for every node in your cluster. 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
...
```

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]
```

Here you can see that 37.5% of CPU processing is being spent on a search thread. This is key to your understanding, because you can then fine-tune your search queries that may be causing the CPU spike. Expect that the search value won't always be there. Elasticsearch may present other values here like `merge`, `index`, and the like that identify the operation being performed on that thread. You know this is CPU-related because of the `cpu usage` identifier. 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
```

The final line before the stack trace tells you that Elasticsearch found that this thread with the same stack trace was present in 10 out of the 10 snapshots it took within a few milliseconds.

Of course, it's worth learning how Elasticsearch gathers the hot_threads API information for presentation. Every few milliseconds, Elasticsearch collects information about thread duration, its state (WAITING/BLOCKED), and the duration of the wait or block for each thread. After a set interval (500 ms by default), Elasticsearch does a second pass of the same information-gathering operation. During each of these passes, it takes snapshots of each stack trace. 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';
```

- `type`—One of `cpu`, `wait`, or `block`. Type of thread state to snapshot for.
- `interval`—Time to wait between the first and second checks. Defaults to 500 ms.
- `threads`—Number of top "hot" threads to display.

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 per- formance on a given node. In some cases, you'll need to manually configure and over- ride how thread pools are managed to avoid cascading failure scenarios. Under aheavy load, Elasticsearch may spawn thousands of threads to handle requests, causing your cluster to fail. Knowing how to tune thread pools requires intimate knowledge of how your application is using the Elasticsearch APIs. For instance, an application that uses mostly the bulk index API should be allotted a larger set of threads. Otherwise, `bulk index` requests can become overloaded, and new requests will be ignored.

You can tune the thread pool settings within your cluster configuration. Thread pools are divided by operation and configured with a default value depending on the operation type. For brevity, we're listing only a few of them:

- `bulk`—Defaults to a fixed size based on the number of available processors for all bulk operations.
- `index`—Defaults to a fixed size based on the number of available processors for index and delete operations.
- `search`—Defaults to a fixed size that's three times the number of available processors for count and search operations.

Looking at your elasticsearch.yml configuration, you can see that you can increase the size of the thread pool queue and number of thread pools for all bulk operations. 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
```

Note that there are two thread pool types, `fixed` and `cache`. A `fixed` thread pool type holds a fixed number of threads to handle requests with a backing queue for pending requests. The `queue_size` parameter in this case controls the number of threads and defaults to the five times the number of cores. A `cache` thread pool type is unbounded, meaning that a new thread will be created if there are any pending requests.

Armed with the cluster health API, slow query and index logs, and thread information, you can diagnose CPU-intensive operations and bottlenecks more easily. The next section will cover memory-centric information, which can help in diagnosing and tuning Elasticsearch performance issues.

11.3.3 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. These garbage-collection operations are time consuming and cause system pauses. Loading too much data in memory can also lead to `OutOfMemory` exceptions, causing failures and unpredictable results—a problem that even the garbage collector can't address.

For Elasticsearch to be fast, some operations are performed in memory because of improved access to field data. For instance, Elasticsearch doesn't just load field data for documents that match your query; it loads values for all the documents in your index. This makes your subsequent query much faster by virtue of having quick access to in-memory data.

The JVM heap represents the amount of memory allocated to applications running on the JVM. For that reason, it's important to understand how to tune its performance to avoid the ill effects of garbage collection pauses and `OutOfMemory` exceptions. You set the JVM heap size via the `HEAP_SIZE` environment variable. The two golden rules to keep in mind when setting your heap size are as follows:

- *Maximum of 50% of available system RAM*—Allocating too much system memory to the JVM means there's less memory allocated to the underlying file-system cache, which Lucene makes frequent use of.
- *Maximum of 32 GB RAM*—The JVM changes its behavior at over 32 GB allocated by not using compressed ordinary object pointers (OOP). This means that setting the heap size under 32 GB uses approximately half the memory space.

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.

The filter cache stores the results of filters and query operations in memory. This means that an initial query with a filter applied will have its results stored in the filter cache. Every subsequent query with that filter applied will use the data from the cache and not go to disk for the data. The filter cache effectively reduces the impact on CPU and I/O and leads to faster results of filtered queries.

Two types of filter caches are available in Elasticsearch:

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

The node-level filter cache is the default setting and the one we'll be covering. The index-level filter cache isn't recommended because you can't predict where the index will reside inside the cluster and therefore can't predict memory usage.

The node-level filter cache is an LRU (least recently used) cache type. That means that when the cache becomes full, cache entries that are used the least amount of times are destroyed first to make room for new entries. Choose this cache type by setting `index.cache.filter.type` to `node`, or don't set it at all; it's the default value. Now you can set the size with the `indices.cache.filter.size` property. It will take either a percentage value of memory (20%) to allocate or a static value (1024 MB) within your elasticsearch.yml configuration for the node. Note that a percentage property uses the maximum heap for the node as the total value to calculate from.

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 field-data 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. By default, this is an unbounded cache, meaning that it will grow until it trips the field-data circuit breaker (covered in the next section). By specifying a value for the field-data cache, you tell Elasticsearch to evict data from the structure once the upper bound is reached.

Your configuration should include an `indices.fielddata.cache.size` property that can be set to either a percentage value (20%) or a static value (16 GB). These values represent the percentage or static segment of node heap space to use for the 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';
  ```

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" : { … }
    }
  }, ...
```

These operations will break down the current state of the cache. Take special note of the number of `evictions`. Evictions are an expensive operation and a sign that the field-data cache may be set to too small of a value.

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 `OutOf-Memory` exception. They work by introspecting data fields requested by a query to determine whether loading the data into the cache will push the total size over the cache size limit. Two circuit breakers are available in Elasticsearch, as well as a parent circuit breaker that sets a limit on the total amount of memory that all circuit breakers may use:

- `indices.breaker.total.limit`—Defaults to 70% of heap. Doesn't allow the field-data and request circuit breakers to surpass this limit.
- `indices.breaker.fielddata.limit`—Defaults to 60% of heap. Doesn't allow the field-data cache to surpass this limit.
- `indices.breaker.request.limit`—Defaults to 40% of heap. Controls the size fraction of heap allocated to operations like aggregation bucket creation.

The golden rule with circuit breaker settings is to be conservative in their valuesbecause the caches the circuit breakers control have to share memory space with memory buffers, the filter cache, and other Elasticsearch memory use.

AVOIDING SWAP

Operating systems use the swapping process as a method of writing memory pages to disk. This process occurs when the amount of memory isn't enough for the operating system. When the swapped pages are needed by the OS, they're loaded back in memory for use. Swapping is an expensive operation and should be avoided.

Elasticsearch keeps a lot of runtime-necessary data and caches in memory, as shown in figure 11.4, so expensive write and read disk operations will severely impact a running cluster. For this reason, we'll show how to disable swapping for faster performance.

The most thorough way to disable Elasticsearch swapping is to set `bootstrap .mlockall` to `true` in the elasticsearch.yml file. Next, you need to verify that the setting

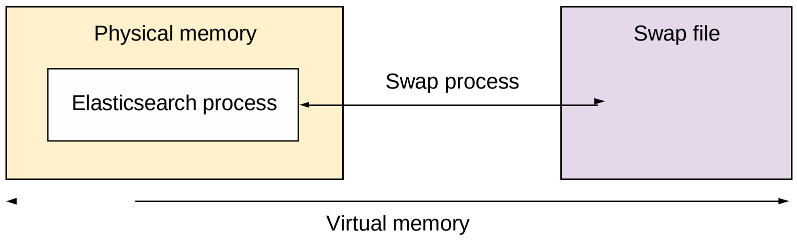

Figure 11.4 Elasticsearch keeps runtime data and caches in memory, so writes and reads can be expensive.

is working. Running Elasticsearch, you can either check the log for a warning or sim- ply query for a live status:

- 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
      } ...
```

If either the warning is visible in the log or the status check results in `mlockall` being set to `false`, your settings didn't work. Insufficient access rights on the user running Elasticsearch are the most common reason for the new setting not taking affect. This is normally solved by running `ulimit -l unlimited` from the shell as the root user. It will be necessary to restart Elasticsearch for these new settings to be applied.

11.3.4 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 immu- table 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. The endeffect is that smaller indices tend to be cached entirely in memory by your OS and become diskless and *fast*.

Because of Lucene's heavy use of the OS file-system cache and the previous recommendation to set the JVM heap at half the physical memory, you can count on Lucene using much of the remaining half for caching. For this simple reason, it's considered best practice to keep the indices that are most often used on faster machines. The idea is that Lucene will keep hot data segments in memory for really fast access, and this is easiest to accomplish on machines with more non-heap memory allocated. But to make this happen, you'll need to assign specific indices to your faster nodes using routing.

First, you need to assign a specific attribute, `tag`, to all of your nodes. Every node has a unique value assigned to the attribute `tag`; for instance, `node.tag: mynode1` or `node.tag: mynode2`. Using the node's individual settings, you can create an index that will deploy only on nodes that have specific tag values. Remember, the point of this exercise is to make sure that your new, busy index is created only on nodes with more non-heap memory that Lucene can make good use of. To achieve this, your newindex, `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"
}'
```

Assuming these specific nodes have a higher non-heap memory allocation, Lucene will cache segments in memory, resulting in a much faster response time for your index than the alternative of having to seek segments on disk.

11.3.5 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. Fortunately, Elasticsearch provides throttling features to help control how much I/O is used.

You can configure throttling at both the node level and the index level. At the node level, throttling configuration settings affect the entire node, but at the index level, throttling configuration takes effect only on the indices specified.

Node-level throttling is configured by use of the `indices.store.throre.throttle.type` property with possible values of `none`, `merge`, and `all`. The `merge` value instructs Elasticsearch to throttle I/O for merging operations across the entire node, meaning every shard on that node. The `all` value will apply the throttle limits to all operations for all of the shards on the node. Index-level throttling is configured muchthe same way but uses the `index.store.throttle.type` property instead. Addition- ally, it allows for a `node` value to be set, which means it will apply the throttling limits to the entire node.

Whether you're looking to implement node- or index-level throttling, Elasticsearch provides a property for setting the maximum bytes per second that I/O will use. 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"
```

We leave as an exercise for you to configure the correct values for your particular system. If the frequency of I/O wait on a system is high or performance is degrading, lowering these values may help ease some of the pain.

Although we've explored ways to curtail a disaster, the next section will look at how to back up and restore data from/to your cluster in the event of one.

11.4 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.

The typical use case for creating snapshots is, of course, to perform backups for disaster recovery, but you may also find it useful in replicating production data in development or testing environments and even as insurance before executing a large set of changes.

11.4.1 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. Furthermore, because every subsequent snapshot is the delta from the previous one, it makes for smaller and faster snapshots over time.

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.

In this section, we'll cover the more common and flexible file-system repository types, how to store snapshots in them, restoring from them, and leveraging common plugins for cloud vendor storage repositories.

11.4.2 Backing up data to a shared file system

Performing a cluster backup entails executing three steps that we'll cover in detail:

- *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.

The first step in enabling snapshots requires you to define a shared file-system repository. The `curl` command in the following listing defines your new repository on a network mounted drive.

Listing 11.4 Defining a new repository

```
curl -XPUT 'localhost:9200/_snapshot/my_repository' -d '
{
    "type": "fs",
    "settings": {
        "location": "smb://share/backups",
        "compress" : true,
        "max_snapshot_bytes_per_sec" : "20mb",
        "max_restore_bytes_per_sec" : "20mb"
    }

}';
```

The name of your repository: my_repository

Define the type of repository as a shared file system.

The network location of your repository

Defaults to true; compresses metadata, not the actual data files

Per-second transfer rate on snapshots

Per-second trader rate on restoration

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';
```

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
```

From this list, you can see a pattern emerging on what Elasticsearch backed up. The snapshot contains information for every index, shard, segment, and accompanying metadata for your cluster with the following file path structure: /<index_name>/<shard_number>/<segment_id>. A sample snapshot file may look similar to the following, which contains information about size, Lucene segment, and the files that each snapshot points to within the directory structure:

```
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" : "_l.fnm",
    "length" :  2703,
    "checksum" : "1ot813j",
    "written_by" : "LUCENE_4_9"
  }, {
    "name" : "__1",
    "physical_name" : "_l_Lucene49_0.dvm",
    "length" : 90,
    "checksum" : "1h6yhga",
    "written_by" : "LUCENE_4_9"
  }, {
    "name" : "_2",
    "physical_name" : "_l.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';
```

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

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';
```

The response contains which indices were part of this snapshot and the total duration of the entire snapshot operation:

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

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';
```

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';
```

Now that you have a solid understanding of the options available when backing up your cluster, let's have a look at restoring your cluster data and state from these snapshots, which you'll need to understand in the event of a disaster.

11.4.3 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';
```

This command will restore the data and state of the cluster captured in the given snapshots: `first_snapshot`. With this operation, you can easily restore the cluster to any point in time you choose.

Similar to what you saw before with the snapshot operation, the restore operation allows for a `wait_for_completion` flag, which will block the HTTP call you make until the restore operation is fully complete. 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';
```

Restore operations also have additional options available that allow you to restore an index to a newly named index space. This is useful if you want to duplicate an index or verify the contents of a restored index from backup:

```
curl -XPOST 'localhost:9200/_snapshot/my_repository/first_snapshot/_restore'
-d '
{
    "indices": "logs_2014",

    "rename_pattern": "logs_(.+)",
    "rename_replacement": "a_copy_of_logs_$1"

}';
```

The index or indices you'll restore from the snapshot

Pattern match for index names to replace

Rename the matched indices

Given this command, you'll restore only the index named `logs_2014` from the snapshot and ignore restoring any other indices found in the snapshot. Because the index name matches the pattern you defined as the `rename_pattern`, the snapshot data will reside in a new index named `a_copy_of_logs_2014`.

NOTE When restoring an existing index, the running instance of the index must be closed. Upon completion, the restore operation will open the closed indices.

Now that you understand how the snapshot API works to enable backups in a network-attached-storage environment, let's explore some of the many plugins available for performing backups in a cloud-based vendor environment.

11.4.4 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

For those deploying on an Amazon Web Services infrastructure, there's a freely available S3 repository plugin available on GitHub and maintained by the Elasticsearch team: https://github.com/elasticsearch/elasticsearch-cloud-aws#s3-repository.

The Amazon S3 repository plugin has a few configuration variables that differ from the norm, so it's important to understand what functionality each of them controls. An S3 repository can be created as such:

```
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

Maximum number of retry attempts on S3 errors

Amazon region where the bucket is located

Once enabled, the S3 plugin will store your snapshots in the defined bucket path. Because HDFS is compatible with Amazon S3, you may be interested in reading the next section, which covers the Hadoop HDFS repository plugin, as well.

Hadoop HDFS

The HDFS file system can be used as a snapshot/restore repository with this simple plugin, built and maintained by the Elasticsearch team that's part of the more general Hadoop plugin project: https://github.com/elasticsearch/elasticsearch-hadoop/tree/master/repository-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
```

Once it's installed, it's time to configure the plugin. The HDFS repository plugin configuration values should be placed within your elasticsearch.yml configuration file. Here are some of the important values:

```
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

Now, with your HDFS repository plugin configured, your snapshot and restore operations will execute using the same snapshot API as covered earlier. The only difference is that the method of snapshotting and restoring will be from your Hadoop file system.

In this section we explored various ways to back up and restore cluster data and state using the snapshot API. Repository plugins provide a convenience for those deploying Elasticsearch with public cloud vendors. The snapshot API provides a simple and automated way to store backups in a networked environment for disaster recovery.

11.5 Summary

We've covered a lot of information in this lab, with the main focus being adminis-tration and optimization of your Elasticsearch cluster. Now that you have a firm under- standing of these concepts, let's recap:

- 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.
- The cluster health API provides a simple way to gauge the general health of your cluster, nodes, and shards.
- Use the slow index and slow query logs to help diagnose index and query operations that can be affecting the performance of the cluster.
- Armed with a solid understanding of how the JVM, Lucene, and Elasticsearch allocate and use memory, you can prevent the operating system from swapping processes to disk.
- The snapshot API provides a convenient way to back up and restore your cluster with network-attached storage. Repository plugins expand this functionality to public cloud vendors.