sematext

Solr & Elasticsearch Consulting. Training. Support.

Elasticsearch DevOps Cheat Sheet

Allocation

Allocation awareness

Avoids putting two copies of the same shard on nodes with the same attribute (e.g. rack, availability zone). For example:

```
node.attr.availability_zone: us-east1  # in elasticsearch.yml
```

Awareness is enabled at the cluster level:

```
curl -XPUT localhost:9200/_cluster/settings?pretty -d '{
  "persistent" : {
    "cluster.routing.allocation.awareness.attributes" : "availability_zone"
  }
}'
```

Allocation filtering

Shards of an index can prefer/avoid nodes with certain attributes. Good for having hot/cold tiers:

```
node.attr.temperature: hot  # in elasticsearch.yml
```

At index creation, you can assign shards to the hot nodes:

```
curl -XPUT localhost:9200/logs01 -d '{
  "settings": {
    "index.routing.allocation.include.tag": "hot"
  }
}'
```

Later on, you can change this value to **cold** move the shards to nodes having **temperature** set to **cold**.

Delayed allocation

Avoids the domino effect of relocation when a node is restarted or temporarily unavailable:

```
curl -XPUT localhost:9200/$INDEX/_settings -d '{
  "settings": {
    "index.unassigned.node_left.delayed_timeout": "5m"
  }
}'
```

Caches

Query cache

Defaults to 10% of heap:

```
indices.queries.cache.size: 7%  # in elasticsearch.yml
```

By default, queries running in the **filter** context will be cached if they run repeatedly, and only on larger segments. You can override this and cache everything in **elasticsearch.yml**:

```
index.queries.cache.everything: true
```

Request cache

Caches results of aggregations on indices that haven't changed. Defaults to 1% of heap:

```
indices.requests.cache.size: 2%
```

Indexing buffer

A node-level buffer for indexing, before a flush will commit to disk. Defaults to 10% of heap:

```
indices.memory.index_buf er_size: 5%
```

Page recycler

Big arrays used by aggregations are put here so they can be reused. Defaults to 10% of heap:

```
cache.recycler.page.limit.heap: 5%
```

Field data

The only way to do sorting/aggregations on text fields. Avoid it if possible. If not, limit it through per-request circuit breakers:

```
indices.breaker.f elddata.limit: 10%
```

And by limiting the overall size:

```
indices.f elddata.cache.size: 20%
```

Merges

Force merge

Might be worth merging indices that don't change into a handful of big segments:

```
curl -XPOST localhost:9200/$INDEX/_forcemerge?max_num_segments=5
```

Segments need to be merged in order to change the compression level, so you can do that before force merging:

```
curl -XPUT localhost:9200/$INDEX/_settings -d '{
  "index.codec": "best_compression"
}'
```

Thread pools

size = number of parallel requests, and **queue_size** = number of waiting requests:

```
threadpool.search.size: 8
threadpool.search.queue_size: 5000
threadpool.bulk.size: 12
threadpool.bulk.queue_size: 500
```

Merge policy

There are multiple knobs here. Most importantly:

- Segments per tier. Defaults to 10. Higher values allow for more segments, giving better indexing throughput at the expense of search latency, disk space, memory and open file handles
- Max merge at once. Defaults to 10. Lower values lower the impact of merging, but will make the process slower (which can potentially throttle indexing)
- Max merged segment. Defaults to 5GB. Lower values result in less merges of large segments, but require more merges of small segments, trading spikes for overall load.

```
curl -XPUT localhost:9200/$INDEX/_settings -d '{
  "index.merge.policy": {
    "segments_per_tier": 50,
    "max_merge_at_once": 50,
    "max_merged_segment": "1gb"
  }
}'
```

Shrink index

Shrink an index into a new one with less shards (factor of the current number of shards):

```
curl -XPOST localhost:9200/logs01/_shrink/logs01_shrinked  -d '{
  "settings": {
    "index.number_of_shards": 1
  }
}'
```

Troubleshooting: get info

Cat health

```
curl localhost:9200/_cat/health?v
```

v is for "verbose", shows column headers. Gives number of nodes, shards (started, initializing, relocating) and cluster color:

- All primaries and replicas are up
- All primaries are up, but not all replicas
- Not all primaries are up

Cat nodes

```
curl localhost:9200/_cat/nodes?v
```

Shows figures like load and heap usage of nodes. You can **select columns** via the **help** parameter to get other metrics.

_cat/allocation

How many shards are on each node and how much disk space they take (vs free space).

_cat/indices

How big is each index; how many shards and replicas it has.

_cat/shards

How big is each shard and on which node it is. Shows whether a shard is STARTED, UNASSIGNED, INITIALIZING or RELOCATING. You can easily **grep** though those values when you have many shards

_cat/segments

How big each segment in each shard is (including memory usage). You can filter by index, for example:

```
curl localhost:9200/_cat/segments/$INDEX?v
```

If you look at the files, you'll see different extensions. Most importantly (in terms of memory and storage):

- .cfs, .cfe: These are **compound segments**
- .fdt: **Stored fields** (like **_source**)
- .tim: **Term dictionary**, used when searching in indexed fields
- .doc: **Frequency** of each term in each document (for scoring)
- .pos: **Positional** information (for phrase searches)
- .pay: **Payloads**, most notably character offsets (for the Postings-based highlighters)
- .nvd, .nvm: Field lengths (a.k.a. **norms** - also used for storing)
- .dvd, .dvm: **Doc values** (used for sorting and aggregations)
- .tv?: **Term vectors** (used for the term-vector-based highlighters)
- .dii, .dim: **Point values** (for geo fields as well as numerics)

More information can be found here (for Elasticsearch 5.x, which uses Lucene 6.x, you may need to change the version):
https://lucene.apache.org/core/6_0_0/core/org/apache/lucene/codecs/lucene60/package-summary.html

_cat/pending_tasks

In-progress operations in your cluster. You'd typically catch long-running ones (e.g. snapshot, force merge) or the ones that get queued up when the cluster is in trouble and the master gets overloaded (e.g. lots of mapping/cluster state updates).

_cat/thread_pool

How many threads are active (working on) searches, bulk indexing and so on. You can also see how many are enqueued (queue) compared to the queue.size and how many were rejected (usually because the queue was full).

_cat/fielddata

How much heap field data (the in memory equivalent of doc values) takes. Per field, per node.

Nodes stats

```
curl localhost:9200/_nodes/stats?pretty
```

Gives back statistics of all nodes in the cluster. You can filter nodes, too, like **_nodes/_local/stats** just for the current node. Relevant metrics include:

- How much time was spent in queries, fetches, indexing, merging, etc
- How much memory current segments take, broken by type (e.g. term dictionary, doc values) which is a good indicator of the live set
- Current and maximum amount of heap usage per pool. Good indicator of

Nodes hot threads

```
curl localhost:9200/_nodes/hot_threads
```

Tells you what's keeping Elasticsearch busy. Add **type=wait** or **type=block** to see what's keeping it from being busy. You can also filter nodes like **_nodes/_local/hot_threads**

OS stats

top, **iotop**, **dstat**, **iostat** help figure out what the bottleneck is. Usually:

- Aggregations are CPU-intensive and memory-intensive. The last part may translate into high GC (check the logs for longer GC events)
- Full-text search (without aggregations) is IO latency sensitive
- Indexing (especially merging) is CPU intensive and IO throughput intensive
- Snapshots, replication and replication are network and disk intensive

Cluster allocation explain

Shows all the decisions that make a particular shard not to be allocated on different nodes:

```
curl localhost:9200/_cluster/allocation/explain?pretty -d'{
  "index": "INDEX_NAME",
  "shard": 0,
  "primary": true
}'
```

Also accepts the node name as a **node** value in the body to show the explanation only for it.

Indices shard stores

```
curl localhost:9200/$INDEX/_shard_stores?pretty
```

Returns the last exception that occurred while opening shards of this index.

Troubleshooting Actions

Total shards per node

How many shards an index can have on each node (good for force-balancing the cluster):

```
curl -XPUT localhost:9200/$INDEX/_settings -d '{
  "index.routing.allocation.total_shards_per_node": 2
}'
```

Disk allocation thresholds

Prevents nodes from running out of disk.

Low watermark: when to stop allocating new shards.
High watermark: when to relocate existing shards.

```
curl -XPUT localhost:9200/_cluster/settings -d '{
   "persistent" : {
      "cluster.routing.allocation.disk.watermark.low" : "70%",
      "cluster.routing.allocation.disk.watermark.high" : "85%"
   }
}'
```

Shard reroute (allocate, move and cancel)

Allows you to try and allocate a shard manually, or cancel a replication/relocation, or to move a shard:

```
curl -XPOST localhost:9200/_cluster/reroute -d '{
   "commands" : [ {
      "move" :
         {
          "index" : "INDEX_NAME", "shard" : SHARD_NUMBER,
          "from_node" : "SOURCE_NODE", "to_node" : "DESTINATION_NODE"
         }
      }
   ]
}'
```

Concurrent replications, relocations and bandwidth

How many shards can be replicated from each node:

```
curl -XPUT localhost:9200/_cluster/settings?pretty -d '{
  "persistent" : {
    "cluster.routing.allocation.node_concurrent_recoveries": 2
  }
}'
```

How many shards can move around, cluster-wide:

```
curl -XPUT localhost:9200/_cluster/settings?pretty -d '{
  "persistent" : {
    "cluster.routing.allocation.cluster_concurrent_rebalance": 2
  }
}'
```

How much bandwidth can recovery/rebalancing take:

```
curl -XPUT localhost:9200/_cluster/settings?pretty -d '{
  "persistent" : {
    "indices.recovery.max_bytes_per_sec": "20mb"
  }
}'
```

Transaction log settings

Trade durability for performance (less IOPS):

```
curl -XPUT localhost:9200/$INDEX/_settings -d '{
  "index.translog": {
    "index.translog.durability": "async"
  }
}'
```

GC tuning

If survivor space is mostly full, you can increase it by lowering -XX:SurvivorRatio in jvm.options (default is 8 on most platforms).

If the whole young generation (survivor + eden) is mostly full, you can increase it via -XX:NewSize.

On large heaps (>30GB, usually you'd want to stay under 30GB to get compressed pointers, but 60-90GB may be needed on some big boxes), using G1 instead of CMS should help. To do that, replace:

```
-XX:+UseConcMarkSweepGC
-XX:CMSInitiatingOccupancyFraction=75
-XX:+UseCMSInitiatingOccupancyOnly
```

With:

```
-XX:+UseG1GC
```

Clear caches

Quick way to free some heap:

```
curl -XPOST localhost:9200/_cache/clear
```