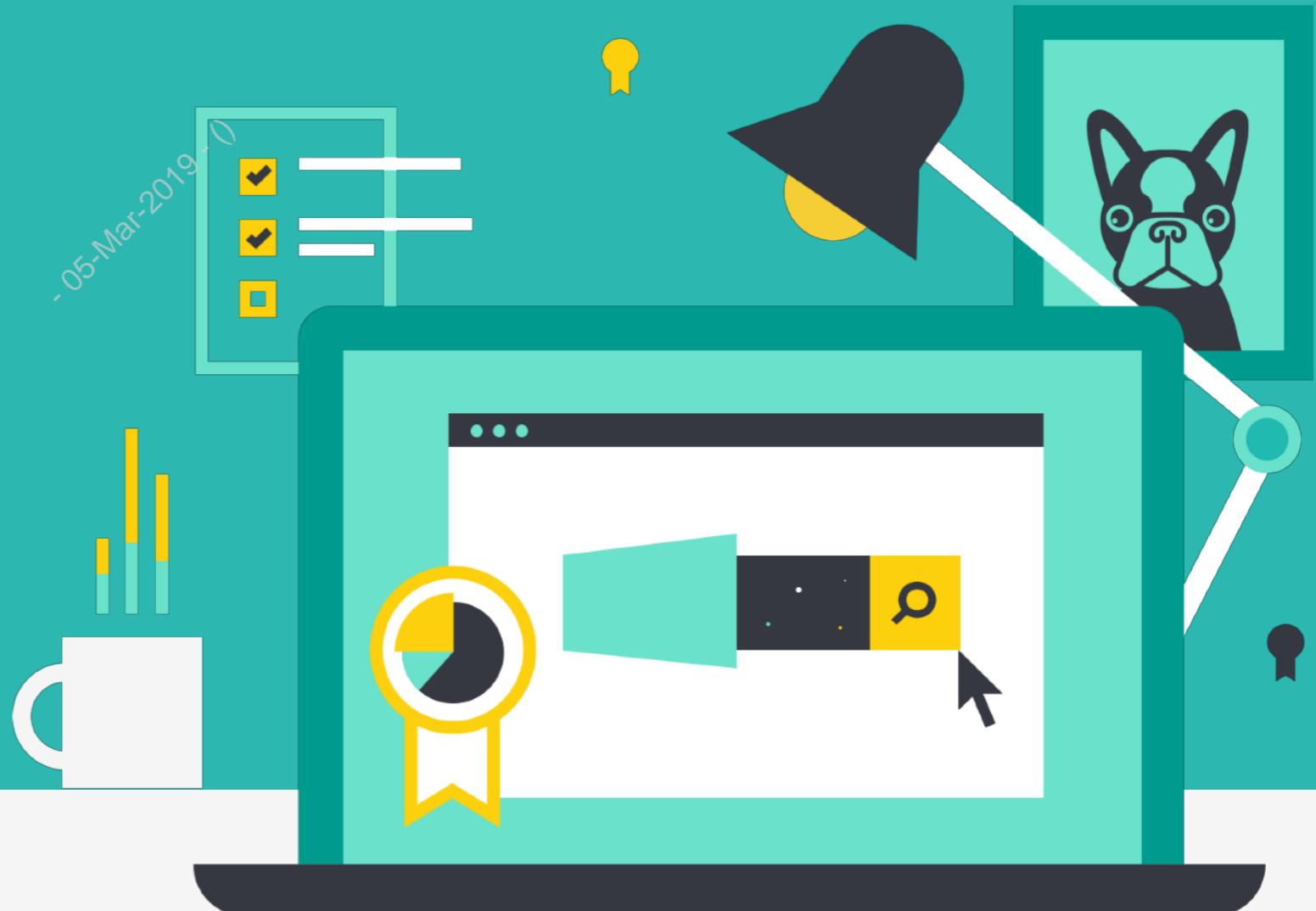




Elasticsearch Engineer II

An Elastic Training Course



6.5.1

elastic.co/training

6.x.x

Elasticsearch Engineer II

Course: Elasticsearch Engineer II

Version 6.5.1

© 2015-2019 Elasticsearch BV. All rights reserved. Decompiling, copying, publishing and/or distribution without written consent of Elasticsearch BV is strictly prohibited.

~05-Mar-2019~0

Welcome to This Virtual Training

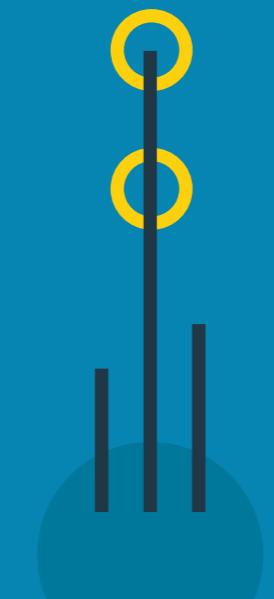
- We will start momentarily
- The training will start with an audio/video test, to make sure that everyone can hear and see the instructors
- To prevent any audio/video issues, please:
 - disable any ad blockers or script blockers
 - use a supported web browser: Chrome or Firefox
- In case of problems, try the following steps in order:
 - refresh this web page
 - open this page in an "incognito" or "private" window
 - try another web browser
 - as a last resort, restarting your computer sometimes helps too

Welcome to This Training

- Visit training.elastic.co and log in
 - follow instructions from registration email to get access
- Go to "**My Account**" and click on today's training
- Download the PDF file (this contains all the slides)
- Click on "**Virtual Link**" to access the Lab Environment
 - create an account
 - you will need an access token, which the instructor will provide

Agenda and Introductions

'05-Mar-2019-'0



About This Training

- Environment
- Introductions
- Agenda...

~05-Mar-2019~0

Course Agenda

1 Elasticsearch Internals

2 Field Modeling

3 Fixing Data

4 Advanced Search & Aggregations

5 Cluster Management

6 Capacity Planning

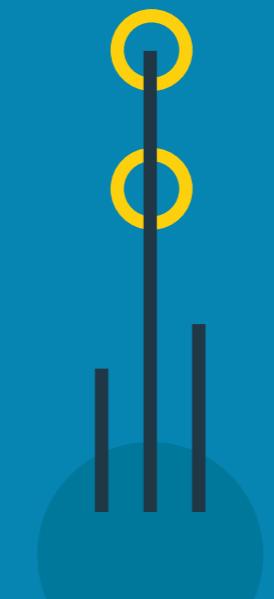
7 Document Modeling

8 Monitoring and Alerting

9 From Dev to Production

Datasets

,05-Mar-2019 - 0



Static Data vs. Time Series Data

- In general, we can categorize most data in our customers' use cases as one of the following:
 - **(relatively) static data:** a large (or small) dataset that may grow or change slowly, like a catalog or inventory of items
 - **time series data:** event data associated with a moment in time that typically grows rapidly, like log files or metrics
- Elasticsearch works great for both types of data
 - and therefore we will use two datasets in the course...



Static Dataset

- Our static dataset is a collection of Elastic blog posts:



16 JANUARY 2018 **ENGINEERING**

Performance Impact of Meltdown on Elasticsearch

By Elastic Engineering

What's the impact of the kernel patches for the Meltdown vulnerability on Elasticsearch?

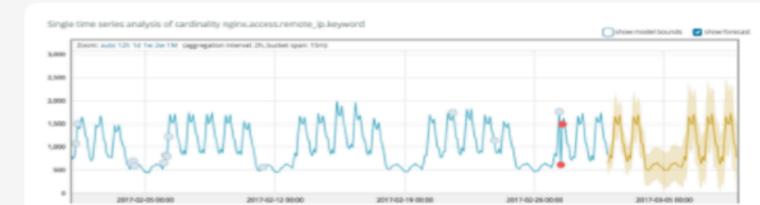


12 JANUARY 2018 **ENGINEERING**

Document-Level Attribute-Based Access Control with X-Pack 6.1

By Mike Barretta

Thanks to a new feature in Elasticsearch 6.1, attribute-based access control is among the X-Pack security features. Learn about what it is and why you need it!



10 JANUARY 2018 **ENGINEERING**

On-demand forecasting with machine learning in Elasticsearch

By Hendrik Muhs • Thomas Grabowski

The newest X-Pack feature in 6.1 is on-demand forecasting. Machine learning can model the data and predict multiple time intervals into the future.



Our Blogs Dataset

- The **blogs** index contains blog posts from elastic.co/blog:

```
{  
  "publish_date": "2017-11-10T07:00:00.000Z",  
  "seo_title": "Apply for an Elastic{ON} Opportunity Grant Today!",  
  "category": "News",  
  "locales": "de-de,fr-fr",  
  "title": "Apply for an Elastic{ON} Opportunity Grant Today!",  
  "content": " For the past few years, our developer relations team has been  
running an informal scholarship program of sorts to help folks from  
underrepresented groups in technology attend Elastic{ON}. ...",  
  "author": "Anna Ossowski",  
  "url": "/blog/apply-for-an-elasticon-opportunity-grant-today"  
}
```

What do we want to build with our data?

- We want users to be able to search our blogs
 - and get relevant and meaningful search results



elastic

logstash

Categories

- (119)
- Engineering (106)
- Releases (73)
- This week in Elasticsearch and Apache Lucene (70)
- The Logstash Lines (41)

Dates (dd/mm/yyyy)

from:

to:

Sort by:

There are 529 results for "logstash" (13 milliseconds)

Logstash 5.0.0 Released [_score: 11.896213]
October 26, 2016 [Releases]
availability of the biggest release of Logstash yet., Previously, Logstash used /opt/logstash directory to install the binaries, whereas Elasticsearch used, Logstash 5.0 is compatible with Elasticsearch 5.x, 2.x, and even 1.x., Extracting fields from unstructured data is a popular Logstash feature., So, from the entire Logstash team, thank you to our users for using and contributing back to Logstash

Welcome Jordan & Logstash [_score: 11.826626]
August 27, 2013 []
is amazing news for so many Elasticsearch and Logstash users., About Logstash Logstash, which just released version 1.2.0, is one of the most popular open source logs, have received requests to offer commercial support for Logstash ., Logstash shares the same vision., Effectively, Logstash is a generic system to process events.

Logstash 1.4.0 beta1 [_score: 11.775666]
February 19, 2014 []
We are pleased to announce the beta release of Logstash 1.4.0!, Contrib plugins package Logstash has grown brilliantly over the past few years with great contributions, Now having 165 plugins, it became hard for us (the Logstash engineering team) to reliably support all, A bonus effect of this decision is that the default Logstash download size shrank by 19MB compared to, Going forward, Logstash release cycles will more closely mirror Elasticsearch's model of releases.

« 1 2 3 4 5 »

Our Logging Dataset

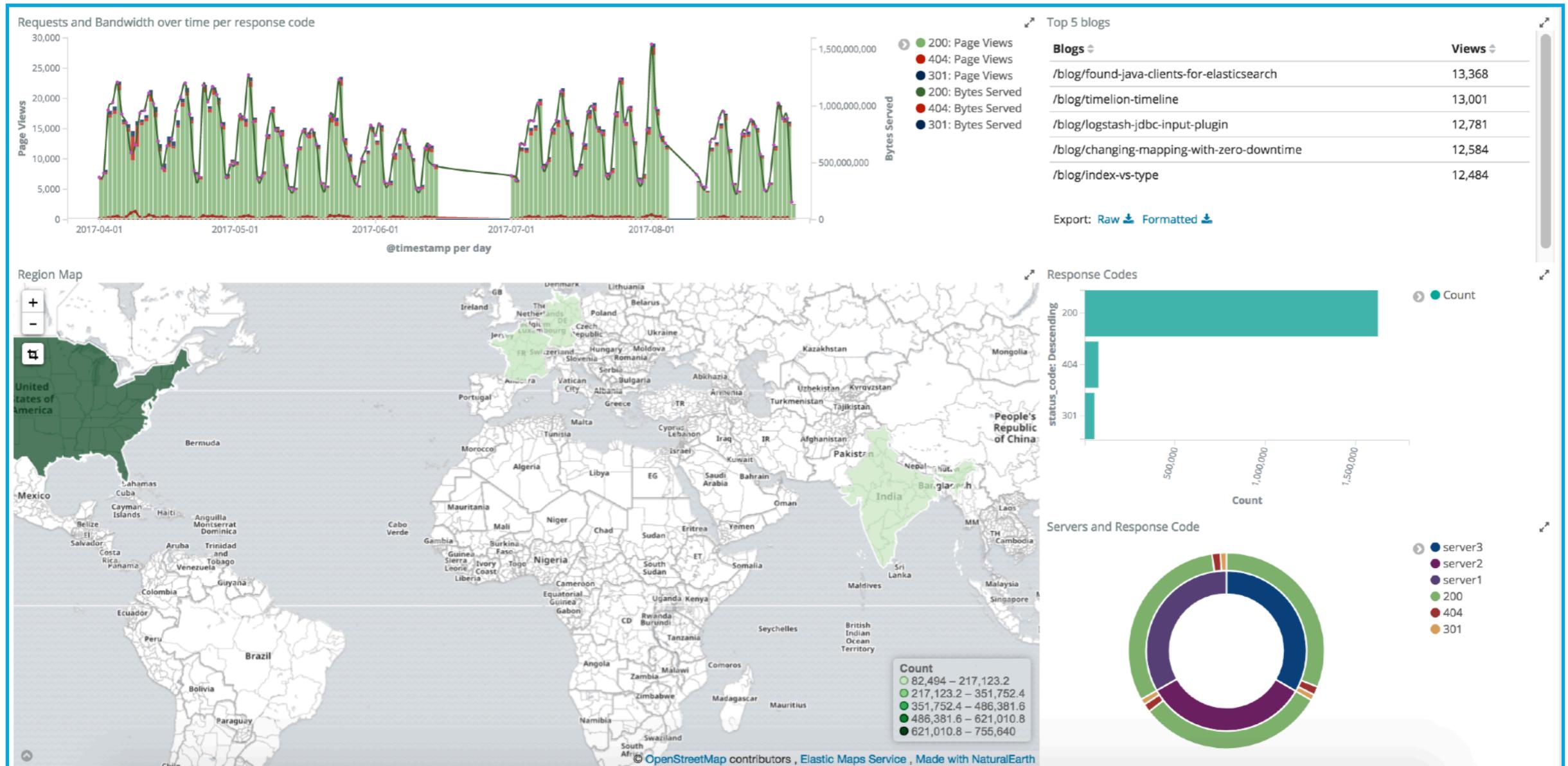
- We also have the web access logs for elastic.co/blog:

```
{  
  "@timestamp": "2017-07-30T03:51:05.551Z",  
  "language": {  
    "url": "/blog/2011/08/05/0.17.4-released.html",  
    "code": "en-us"  
  },  
  "method": "GET",  
  "user_agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X  
10_13_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/  
70.0.3538.77 Safari/537.36",  
  "response_size": 37857,  
  "host": "server1",  
  "http_version": "1.1",  
  "status_code": 404,  
  "runtime_ms": 142,  
  "geoip": {  
    "country_code3": "US",  
    "location": {  
      "lon": -122.1206,  
      "lat": 47.6801  
    },  
    "region_name": "Washington",  
    "city_name": "Redmond",  
    "country_code2": "US",  
    "country_name": "United States",  
    "continent_code": "NA"  
  },  
  "originalUrl": "/blog/2011/08/05/0.17.4-released.html",  
  "level": "info"  
}
```



What do we want from our log data?

- To be able to answer questions about web traffic:



Default Mappings

Blogs Dataset

```
"mappings": {  
    "_doc": {  
        "properties": {  
            "author": {  
                "type": "text",  
                "fields": {  
                    "keyword": {  
                        "type": "keyword",  
                        "ignore_above": 256  
                    }  
                }  
            },  
            "category": {  
                "type": "text",  
                "fields": {  
                    "keyword": {  
                        "type": "keyword",  
                        "ignore_above": 256  
                    }  
                }  
            },  
            "publish_date": {  
                "type": "date"  
            },  
            ...  
        }  
    }  
}
```

Logs Dataset

```
"mappings": {  
    "doc": {  
        "properties": {  
            "geoip": {  
                "properties": {  
                    "city_name": {  
                        "type": "text",  
                        "fields": {  
                            "keyword": {  
                                "type": "keyword",  
                                "ignore_above": 256  
                            }  
                        }  
                    },  
                    "continent_code": {  
                        "type": "text",  
                        "fields": {  
                            "keyword": {  
                                "type": "keyword",  
                                "ignore_above": 256  
                            }  
                        }  
                    },  
                    ...  
                }  
            }  
        }  
    }  
}
```



Lab Environment

~05-Mar-2019~

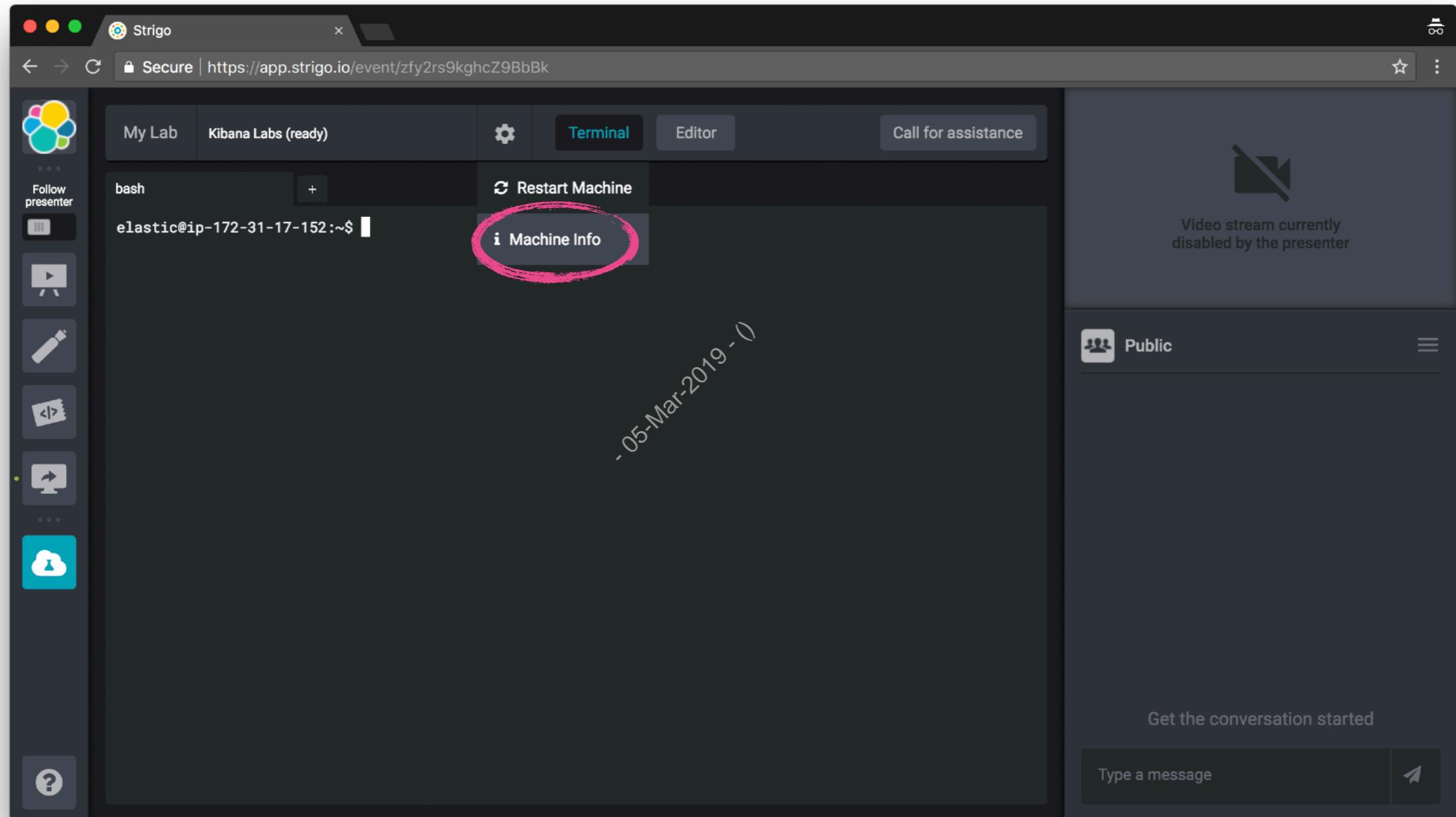
Lab Environment

- Visit Strigo using the link that was shared with you, and log in if you haven't already done so
- Click on "**My Lab**" on the left



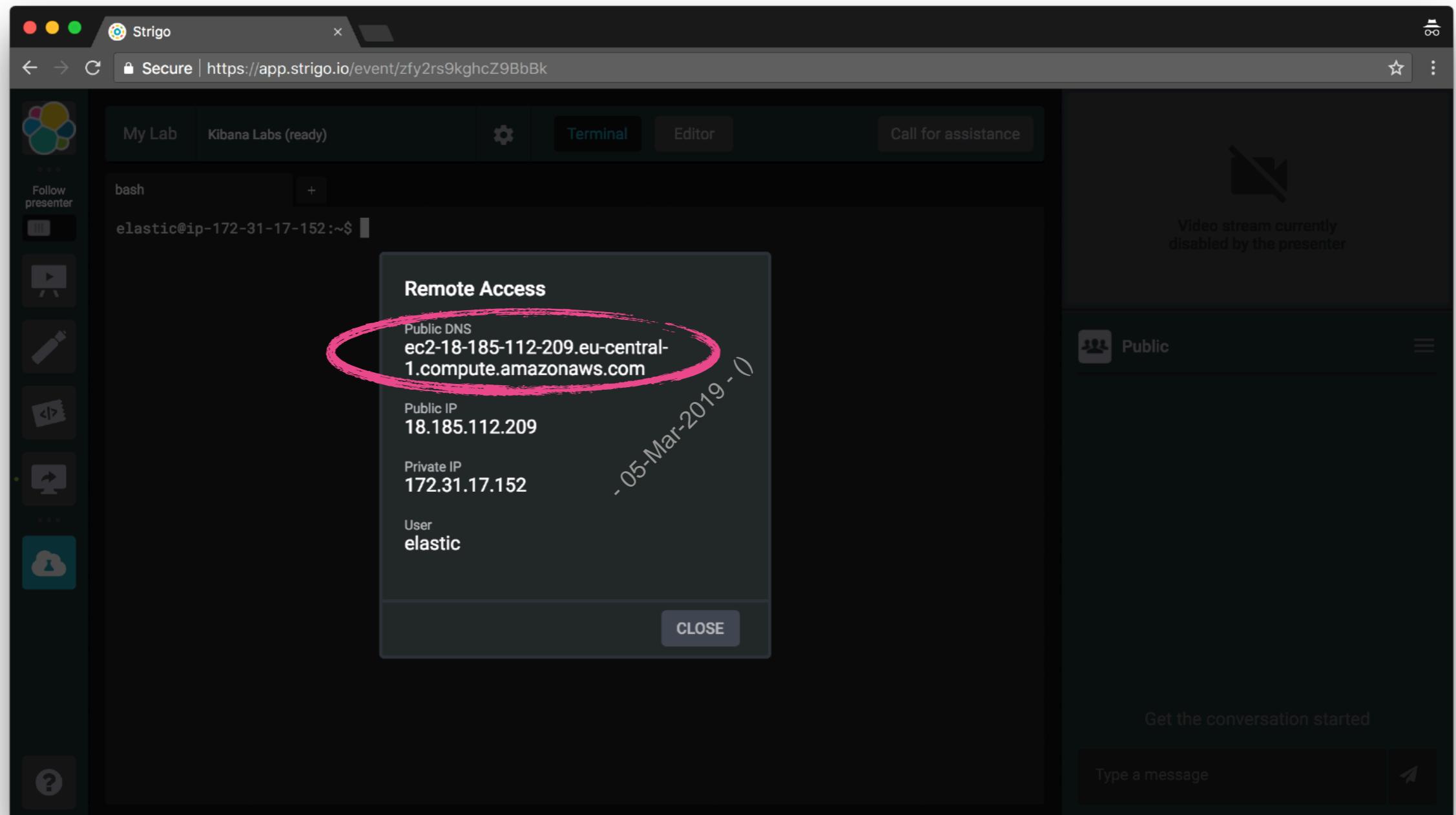
Lab Environment

- Click on the gear icon next to "My Lab" and select "Machine Info"



Lab Environment

- Copy the hostname that is shown under "Public DNS"



Lab Environment

- From here you can access lab instructions and guides
 - You also have them in your .zip file, but it is easier to access and use the lab instructions from here:



elastic

Welcome to Elasticsearch Engineer II

© 2019-08-01

- [Lab Instructions](#)
- [Virtual Classroom User Guide](#)
- [Search Page](#)
- [Kibana](#)

Lab 0

~05-Mar-2019~

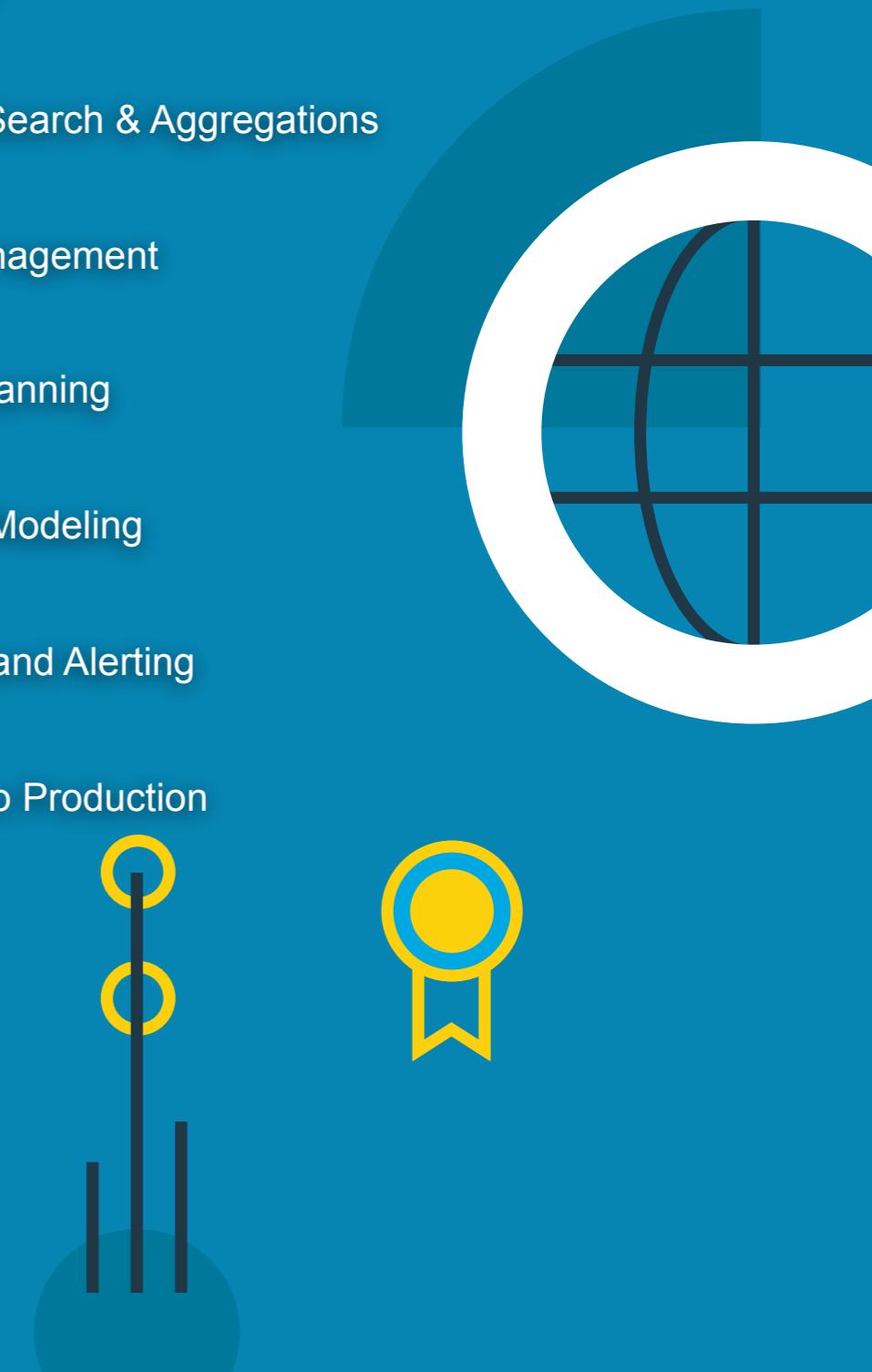
Setup Elasticsearch Cluster

Chapter 1

Elasticsearch Internals

05-Mar-2019 - 0

- 1 Elasticsearch Internals
- 2 Field Modeling
- 3 Fixing Data
- 4 Advanced Search & Aggregations
- 5 Cluster Management
- 6 Capacity Planning
- 7 Document Modeling
- 8 Monitoring and Alerting
- 9 From Dev to Production



Topics covered:

- Lucene Indexing
- Understanding Segments
- Segment Merges
- Elasticsearch Indexing
- Doc Values
- Caching

~05-Mar-2019~0



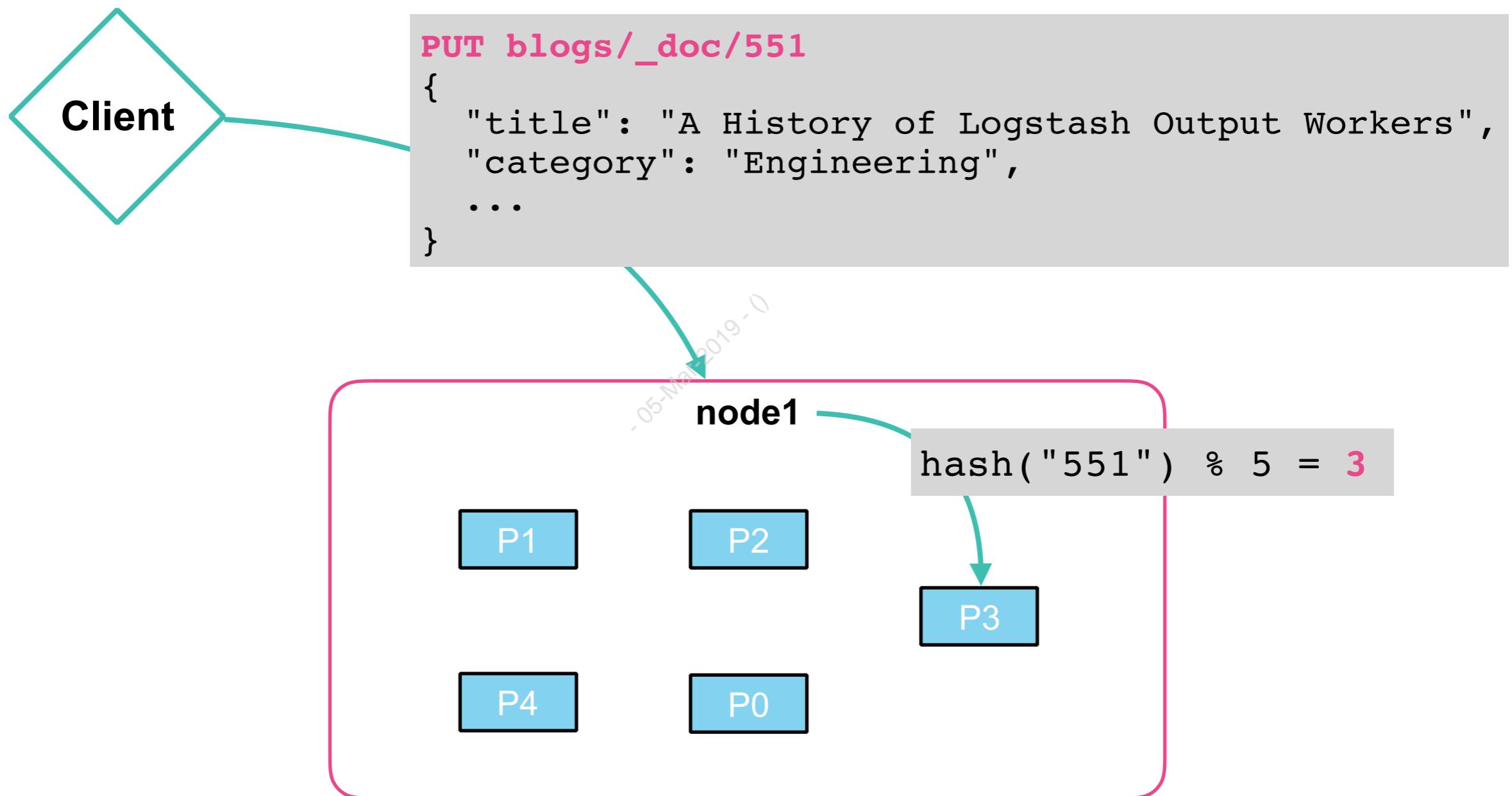
Lucene Indexing

~05-Mar-2019 ~



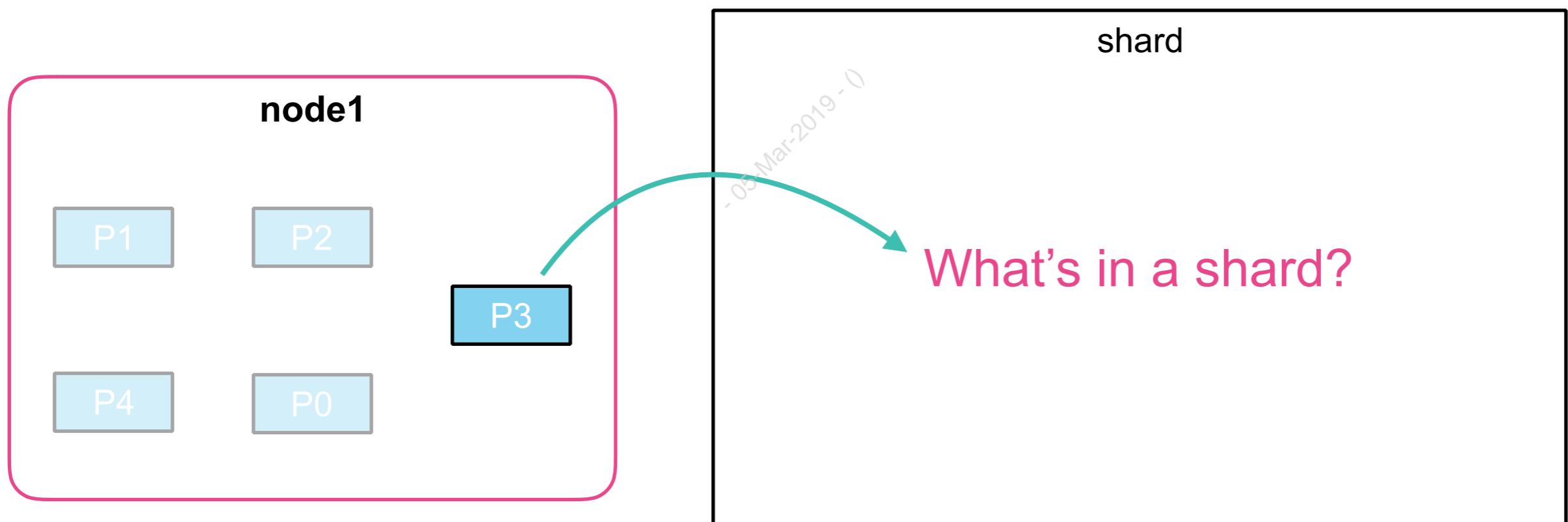
How Data Gets Written

- Node level overview (detailed in Engineer1)
- Next, let's focus on the shard level...

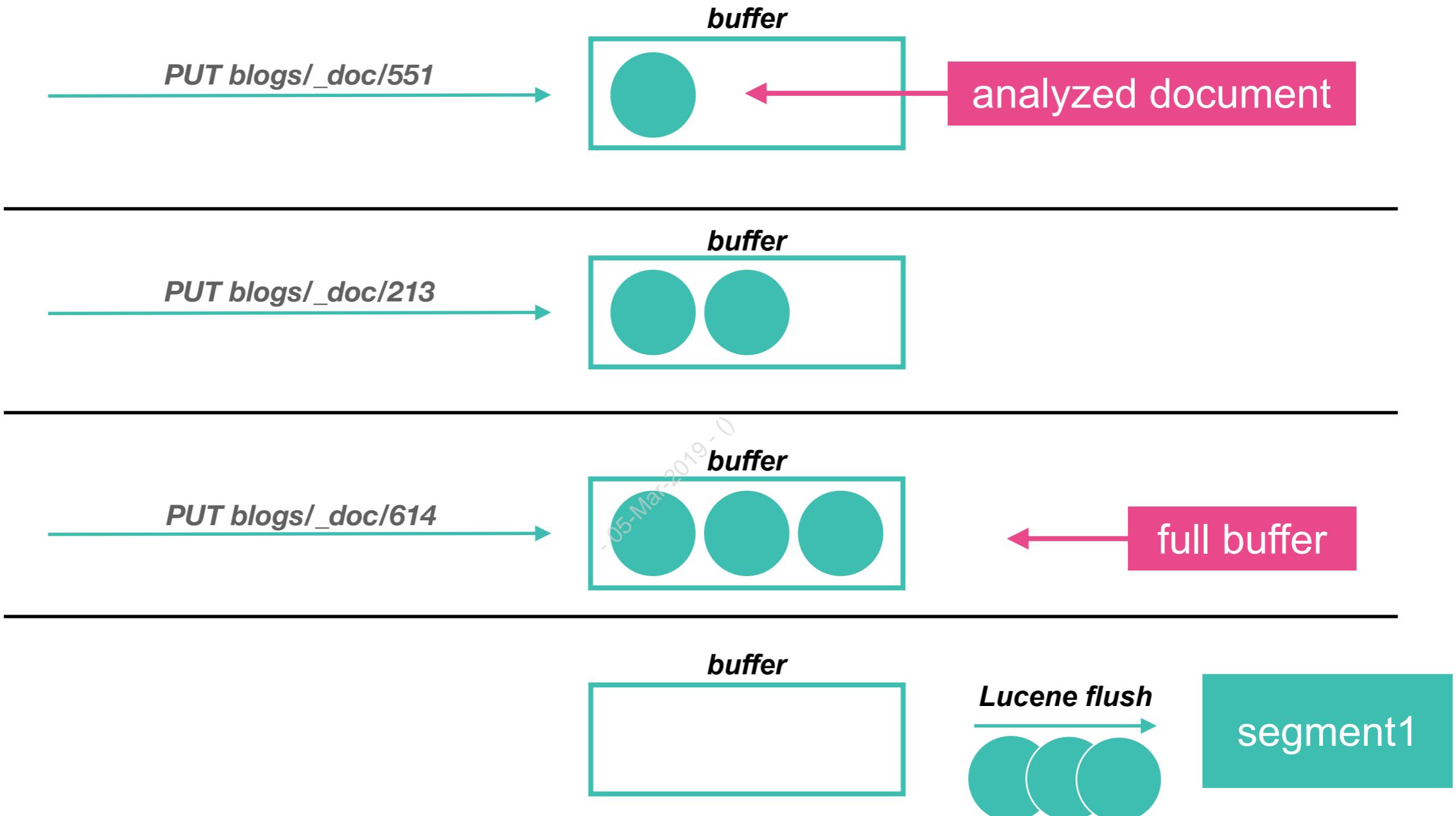


Understanding Shards

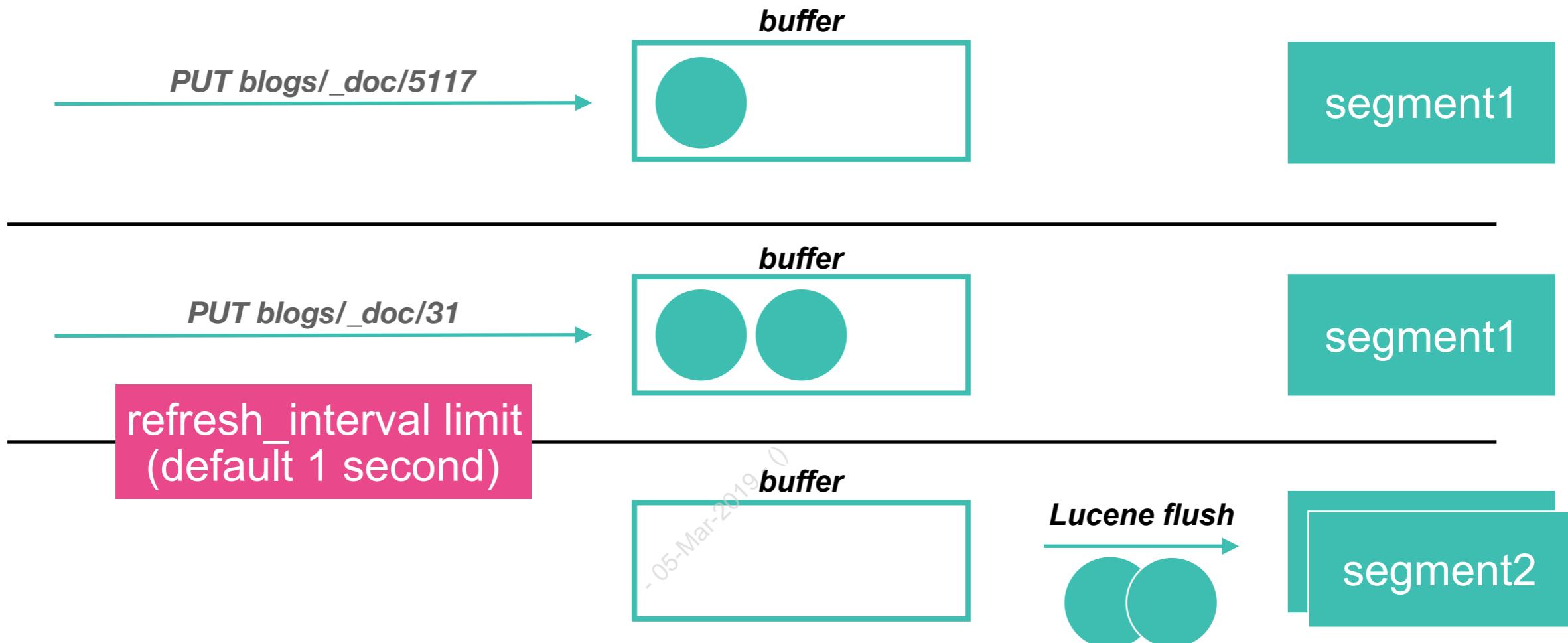
- A **shard** is a single instance of Lucene
 - each shard is a **complete search engine** on its own
 - max # of documents in a shard is **Integer.MAX_VALUE-128**
 - clients do not refer to shards directly (use the index instead)



Lucene Indexing



Lucene Indexing



Indexing Buffer

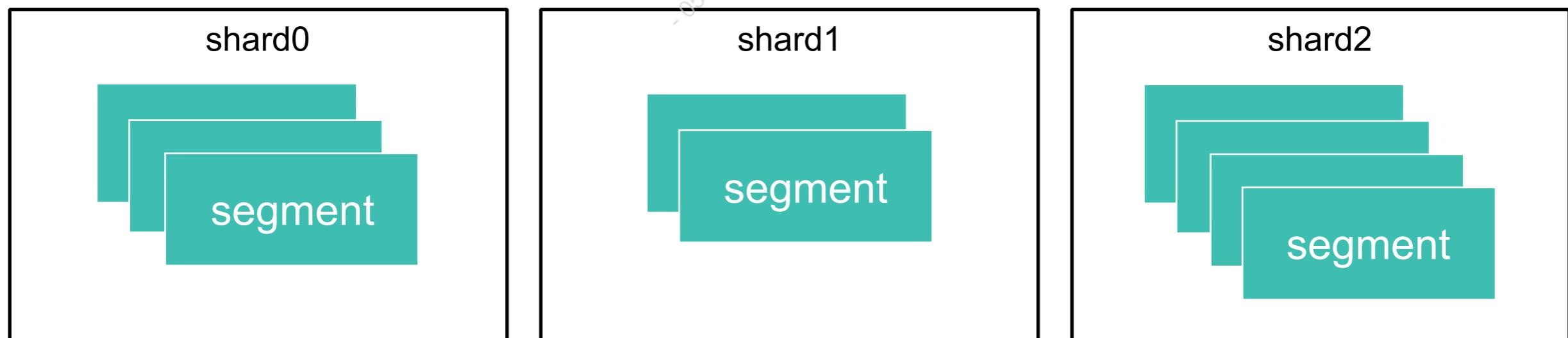
- Documents are analyzed, ...
- then go into a memory buffer
- The indexing buffer **defaults to 10%** of the node heap
- and it **is shared by all shards** allocated to that node
- You can change the buffer size using this static setting, which must be configured on every data node in the cluster:

```
indices.memory.index_buffer_size: 5%
```



Lucene Flush

- A Lucene flush creates new segments from the documents in the memory buffer:
- A Lucene flush happens:
 - when the memory buffer is full
 - Elasticsearch flushes (Lucene commit), which we will see soon
 - when Elasticsearch refreshes...



Elasticsearch Refresh

- New indexed documents are not searchable until a refresh occurs
- By default, every shard is refreshed once every second
 - defined by a dynamic index level setting named **refresh_interval**

```
PUT my_index/_settings
{
  "refresh_interval": "30s"
}
```

Increase the **refresh_interval** in index heavy scenarios to achieve better indexing performance

- The Document APIs that create, update or delete documents have an optional refresh parameter...



The refresh Parameter

- Controls when changes made by **this** request are made visible to searches:
 - false**: the default value, any changes to the document are *not* visible immediately
 - true**: forces a refresh in the affected primary and replica shards so that the changes are visible immediately
 - wait_for**: synchronous request that waits for a refresh to happen

```
PUT my_index/_doc/102/?refresh=wait_for
{
  "firstname" : "James",
  "lastname" : "Brown",
  "address" : "6011 Downtown Lane",
  "city" : "Detroit"
}
```



Understanding Segments

15Mar2019-0

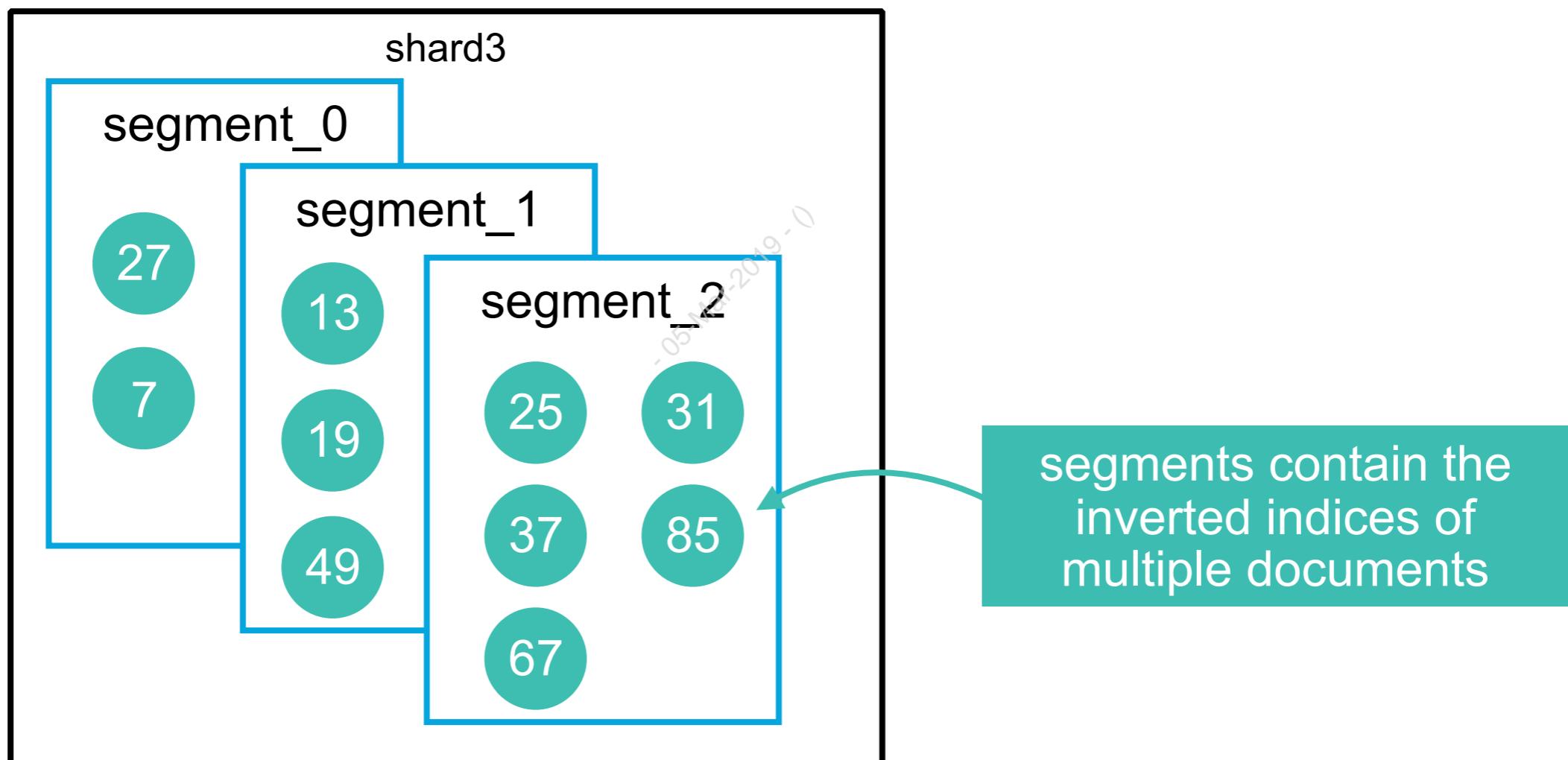
Understanding Segments

- A shard == Lucene instance == ***collection of segments***
 - Think of a segment as an ***immutable “mini-index”***
- Each segment is a fully independent index
 - Each segment is composed by a bunch of files
 - Segment files are written to disk
 - Segment files are ***never updated*** (read only)
- A search request is distributed to the shards and on each shard it is performed sequentially over the segments



Understanding Segments

- A shard == Lucene instance == ***collection of segments***
 - A ***segment*** is a package of many different data structures representing an inverted index for each field



Understanding Segments

- To understand the internals of segments, let's see what the following two documents look like in a segment
 - we will assume their ID's route to the same shard and that they also appear in the same segment

```
PUT my_index/_doc/27
{
  "author": "Uri",
  "category": "Releases",
  "title": "Elastic Cloud Enterprise Beta"
}
```

```
PUT my_index/_doc/14
{
  "author": "Rasmus",
  "category": "Releases",
  "title": "Elastic APM enters beta-1.1"
}
```



Term Dictionary and Frequency Data

segment_0

term dictionary and frequency

author

rasmus	1	14
uri	1	27

title

1	1	14
apm	1	14
beta	2	14, 27
cloud	1	27
elastic	2	14, 27
enterprise	1	27
enters	1	14

category

releases	2	14, 27
----------	---	--------

a dictionary of all terms in all indexed fields, along with the number of docs



Field Names

segment_0

term dictionary and frequency

author

rasmus	1	14
uri	1	27

title

1	1	14
apm	1	14
beta	2	14, 27
cloud	1	27
elastic	2	14, 27
enterprise	1	27
enters	1	14

category

releases	2	14, 27
----------	---	--------

field names

author
category
title

set of field names used in
the index



Term Proximity

segment_0

term dictionary and frequency

author

rasmus	1	14
uri	1	27

title

1	1	14
apm	1	14
beta	2	14, 27
cloud	1	27
elastic	2	14, 27
enterprise	1	27
enters	1	14

category

releases	2	14, 27
----------	---	--------

field names

author
category
title

title term proximity

1	(14: 4, 5)
apm	(14: 1)
beta	(14: 3) (27: 3)
cloud	(27: 1)
elastic	(14: 0) (27: 0)
enterprise	(27: 2)
enters	(14: 2)

the position that the term occurs in each document



Deleted Documents

segment_0

term dictionary and frequency

author

rasmus	1	14
uri	1	27

title

1	1	14
apm	1	14
beta	2	14, 27
cloud	1	27
elastic	2	14, 27
enterprise	1	27
enters	1	14

category

releases	2	14, 27
----------	---	--------

deleted documents

an optional file that indicates which documents are deleted

field names

author
category
title

title term proximity

1	(14: 4, 5)
apm	(14: 1)
beta	(14: 3) (27: 3)
cloud	(27: 1)
elastic	(14: 0) (27: 0)
enterprise	(27: 2)
enters	(14: 2)



Stored Field Values

segment_0

term dictionary and frequency

author

rasmus	1	14
uri	1	27

title

1	1	14
apm	1	14
beta	2	14, 27
cloud	1	27
elastic	2	14, 27
enterprise	1	27
enters	1	14

category

releases	2	14, 27
----------	---	--------

deleted documents

stored field values

_source is stored here

field names

author
category
title

title term proximity

1	(14: 4, 5)
apm	(14: 1)
beta	(14: 3) (27: 3)
cloud	(27: 1)
elastic	(14: 0) (27: 0)
enterprise	(27: 2)
enters	(14: 2)



BKD Trees

segment_0

term dictionary and frequency

author

rasmus	1	14
uri	1	27

title

1	1	14
apm	1	14
beta	2	14, 27
cloud	1	27
elastic	2	14, 27
enterprise	1	27
enters	1	14

category

releases	2	14, 27
----------	---	--------

deleted documents

stored field values

BKD trees

single- and multi-dimensional numerics

field names

author
category
title

title term proximity

1	(14: 4, 5)
apm	(14: 1)
beta	(14: 3) (27: 3)
cloud	(27: 1)
elastic	(14: 0) (27: 0)
enterprise	(27: 2)
enters	(14: 2)



Normalization Factors

segment_0

term dictionary and frequency

author

rasmus	1	14
uri	1	27

title

1	1	14
apm	1	14
beta	2	14, 27
cloud	1	27
elastic	2	14, 27
enterprise	1	27
enters	1	14

category

releases	2	14, 27
----------	---	--------

field names

author
category
title

title term proximity

1	(14: 4, 5)
apm	(14: 1)
beta	(14: 3) (27: 3)
cloud	(27: 1)
elastic	(14: 0) (27: 0)
enterprise	(27: 2)
enters	(14: 2)

deleted documents

stored field values

BKD trees

normalization factors

For each field in each document, a value is stored that is multiplied into the score for hits on that field



Doc Values

segment_0

term dictionary and frequency

author

rasmus	1	14
uri	1	27

title

1	1	14
apm	1	14
beta	2	14, 27
cloud	1	27
elastic	2	14, 27
enterprise	1	27
enters	1	14

category

releases	2	14, 27
----------	---	--------

field names

author
category
title

title term proximity

1	(14: 4, 5)
apm	(14: 1)
beta	(14: 3) (27: 3)
cloud	(27: 1)
elastic	(14: 0) (27: 0)
enterprise	(27: 2)
enters	(14: 2)

deleted documents

stored field values

BKD trees

normalization factors

doc_values

Used for sorting and other operations that require an index to be “uninverted”



Segment Merges

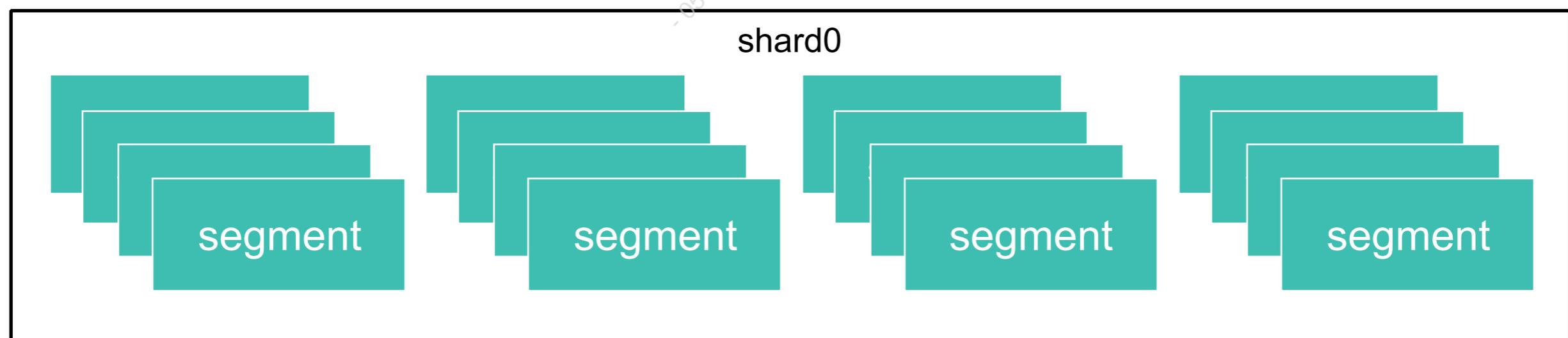
~05-Mar-2019 ~10

Too Many Segments

- One segment per second can be too much
- Queries run sequentially on all segments
- Deletes are "soft" (segments are read only)
- Updates also do "soft" deletes

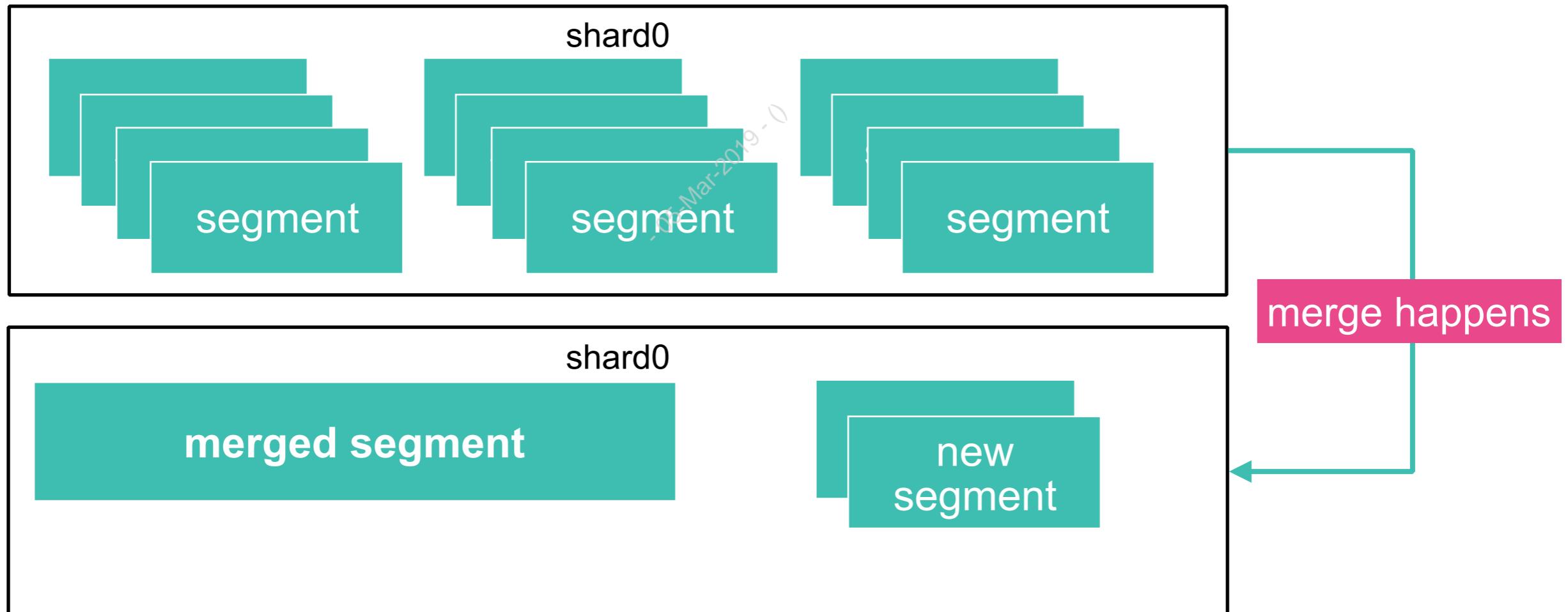
How can a search request be fast?

How do we get rid of deleted documents?



Segments Can Be Merged

- Segments are periodically *merged* into larger segments
 - this is an automatic task on the shard level
- Keeps the index size and number of segments manageable
- Deleted documents also get expunged during a merge



Merging with Indexing Only

- <http://blog.mikemccandless.com/2011/02/visualizing-lucenes-segment-merges.html>



Merging with Updates

- <http://blog.mikemccandless.com/2011/02/visualizing-lucenes-segment-merges.html>



The Force Merge API

- In general, Elasticsearch and Lucene do a good job of deciding when to merge segments
- However, you can force an index to merge its segments:

```
POST blogs/_forcemerge
```

- Keep in mind you should **rarely** worry about invoking **_forcemerge** manually
- If you use it, make sure to **only use _forcemerge** on indices that will never have write operations executed in the future



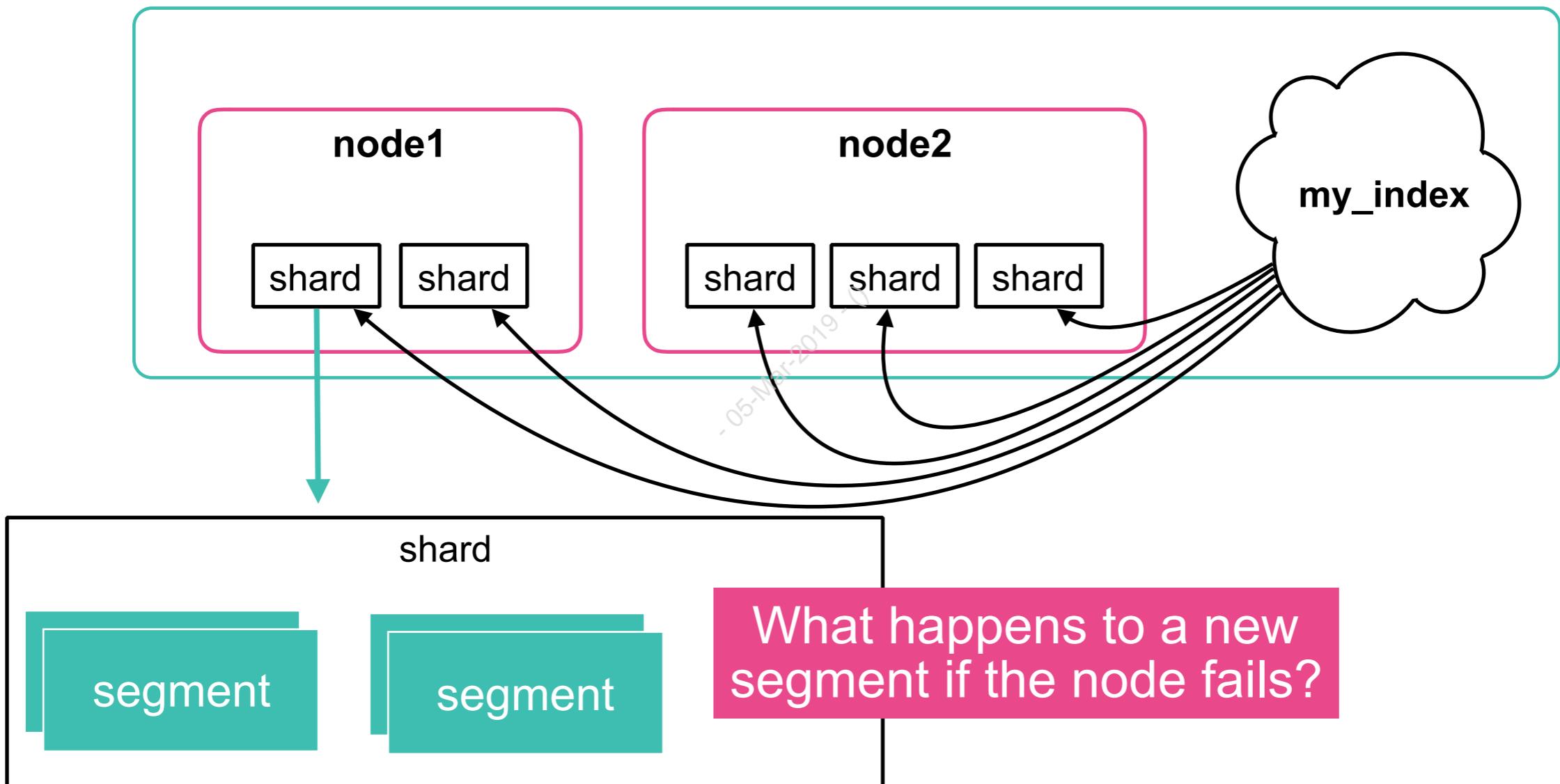
Elasticsearch Indexing

~05-Mar-2019



Index -> Shard -> Segments

- An *index* consists of one or more shards
- A *shard* consists of segments



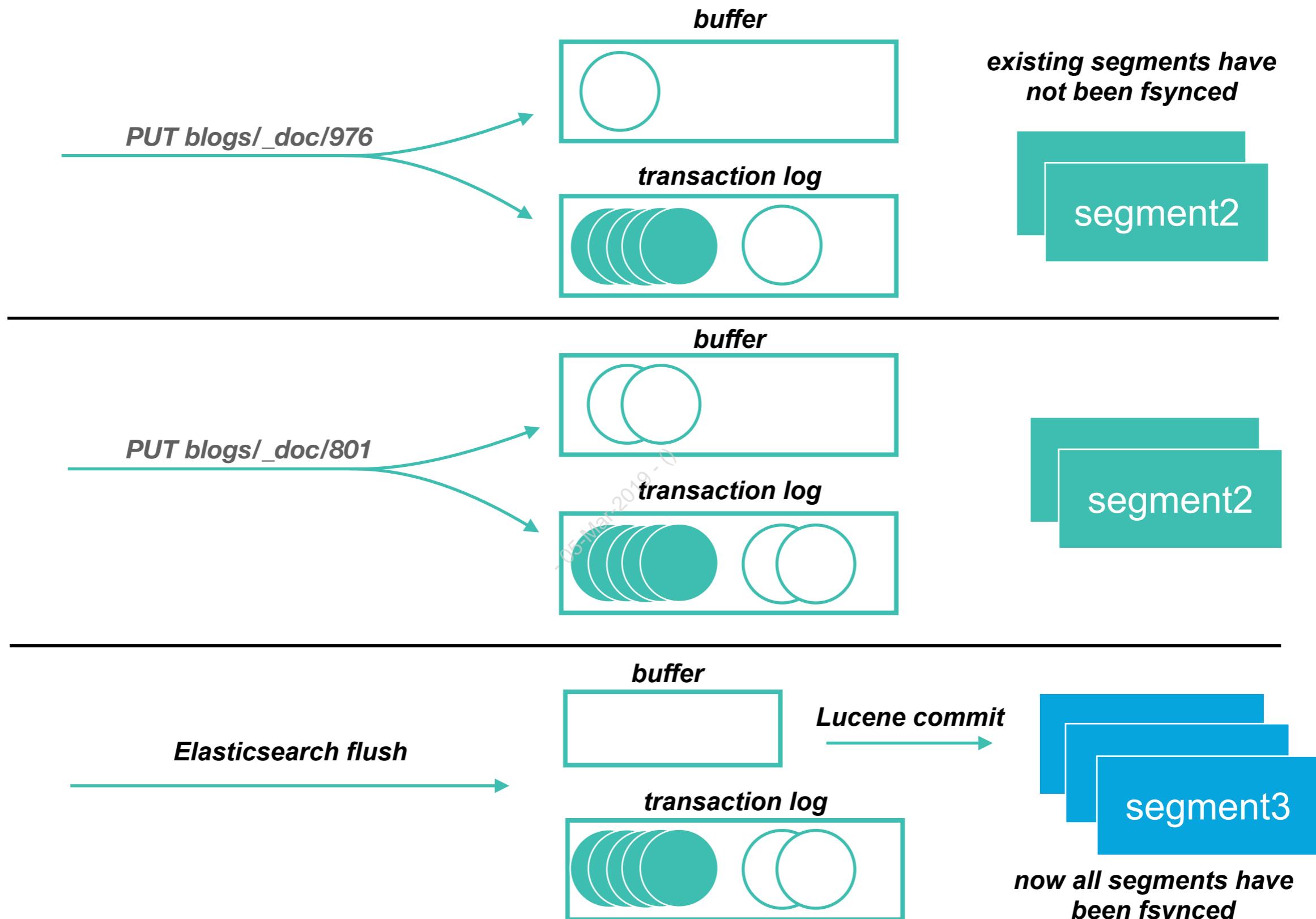
Elasticsearch Indexing

- Segments are fsynced to disk during a *Lucene commit*:
 - a relatively heavy operation
 - should **not** be performed after every index or delete operation
- Until a commit, segment data is susceptible to loss
- To prevent this data loss, Elasticsearch implements a *transaction log* for each shard...

05-Mar-2019



The Transaction Log



Elasticsearch Indexing

- Any write operation is written to the transaction log
- By default, the translog is committed to disk
 - after each write request (index/reindex/delete operations)
 - at the end of a bulk request
- In the event of a crash or hardware failure
 - during a shard recovery, recent transactions can be replayed

05-Mar-2019



The Flush API

- You can force an Elasticsearch flush:

```
POST my_index/_flush
```

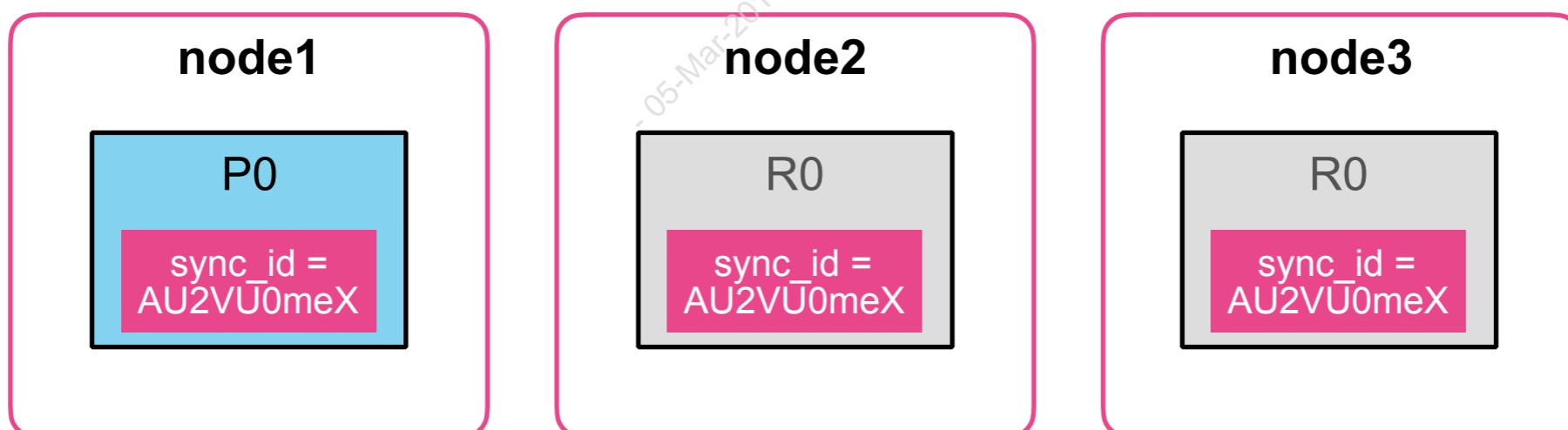
- Do you need to call _flush?
 - Typically, no!
 - Flush happens automatically depending on how many operations get added to the transaction log, how big they are, and when the last flush happened



Synced Flush

- What is a *synced flush*?
 - A synced flush performs a normal flush, then adds a generated unique marker (`sync_id`) to all shards
 - `sync_id` provides a quick way to check if two shards are identical

```
POST my_index/_flush/synced
```



This primary shard and its two replicas are in sync

Doc Values

~05-Mar-2010

Why do I need to care about doc values?

- Let's try something simple, like *sorting* blog posts by author name:

```
GET blogs/_search
{
  "query": {
    "match": {
      "content": "new releases"
    }
  },
  "sort": {
    "author": {
      "order": "asc"
    }
  }
}
```

This simple-looking query
actually fails. Why?



The Error Mentions Fielddata

- Here is the error message from the previous query
 - We are being told to enable ***fielddata***, but also we are being warned that it is expensive and potentially hazardous

```
{  
  "error": {  
    "root_cause": [  
      {  
        "type": "illegal_argument_exception",  
        "reason": "Fielddata is disabled on text fields by  
default. Set fielddata=true on [author] in order to load  
fielddata in memory by uninverting the inverted index. Note  
that this can however use significant memory. Alternatively  
use a keyword field instead."  
      }  
    ],  
    ...  
  }  
}
```

Uninverting an Inverted Index

- We are asking Elasticsearch to sort blogs by the “author” field, which is analyzed text in an inverted index:

The inverted index needs to be *uninverted* somehow if we want to sort by authors' names

author
aaron
alexander
baiera
banon
boness
cam
clint
cohen
...and so on

Not an ideal format
for sorting



Fielddata is probably not the best option

- Enabling **fielddata** allows Elasticsearch to uninvert the inverted field by *loading its values into memory*
 - This is done in the JVM heap “on the fly” (at `_search` time)
 - Old versions of ES had **fielddata** enabled by default, but it was vulnerable to causing out-of-memory issues
- Notice the error message mentions an alternative to enabling **fielddata**:

... fielddata in memory by uninverting the inverted index.
Note that this can however use significant memory.
Alternatively **use a keyword field instead.**"

```
    }  
],  
...
```

Why does “**keyword**” not
have the same issue as
“**text**”?



Doc Values

- An inverted index is great if you are searching for *documents that contain a certain term*
 - but not great when you are searching for *terms that are within a document*
- **Doc values** are a data structure that store the values of a document **on-disk** in a **column-oriented** fashion
 - which makes sorting and aggregations much more efficient
- Doc values are fast and awesome
 - BUT, they **do not exist for analyzed string fields**

So how do I sort by analyzed text?

- ...or perform an aggregation or use it in a script?
 - Well, you should *avoid that scenario* when possible
- Consider indexing the text field as a “**keyword**”, which is unanalyzed text that has doc values

```
"author": {  
  "type": "text",  
  "fields": {  
    "keyword": {  
      "type": "keyword",  
      "ignore_above": 256  
    }  
  }  
}
```

In blogs, we indexed “author” twice: as both “text” and “keyword”

Sort by a keyword Field

- Notice sorting by “**author.keyword**” query works fine and as expected:

```
GET blogs/_search
{
  "query": {
    "match": {
      "content": "new releases"
    }
  },
  "sort": {
    "author.keyword": {
      "order": "asc"
    }
  }
}
```



```
"author": "",  

"author": "A.J. Angus",  

"author": "Aaron Aldrich",  

"author": "Aaron Katz",  

"author": "Aaron Mildenstein",
```



Caching

~05-Mar-2019~



Node Query Cache

- ***One cache per node*** that is shared by all shards
 - Uses the LRU (Least Recent Used) eviction policy
 - It only caches queries inside a filter context
- One static ***node-level setting***:

```
indices.queries.cache.size: "5%"
```

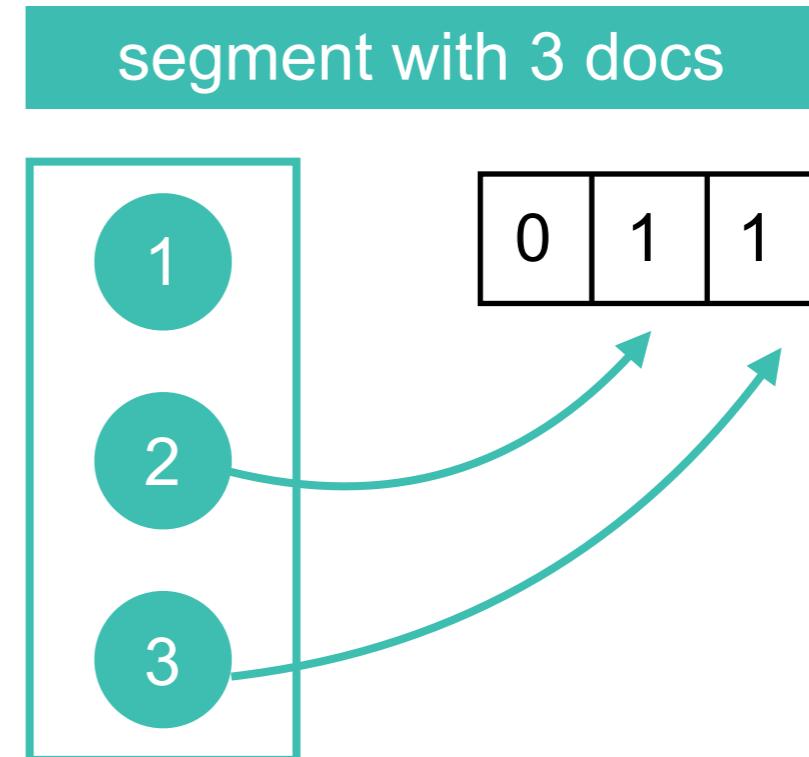
default is **10%** of the heap size. Can also be set with an exact value, like 512mb

- Segment level cache uses ***bit sets***...

Bitsets

- Array of bits, each position represents a document
 - super efficient
- Only built for segments that are "big enough"

```
GET blogs_csv/_search
{
  "query": {
    "bool": {
      "filter": {
        "range": {
          "publish_date": {
            "gte": 2017,
            "lte": 2018
          }
        }
      }
    }
  }
}
```



[better-query-execution-coming-elasticsearch-2-0-frame-of-reference-and-roaring-bitmaps](#)

Shard Request Cache

- One cache per node that is shared by all shards
- LRU (Least Recent Used) eviction policy
- Data modification invalidates the cache
- Good fit for indices that you don't write anymore
- Search requests return results almost instantly
- Static setting that must be configured on every node:

```
indices.request.cache.size: "5%"
```

default is 1% of the heap size. Can also be set with an exact value, like 256mb.

- Shard level request cache
- The cache is enabled on every index by default



Shard Request Cache

- By default, it only caches the results of search requests where size=0
 - hits.total, aggregations, and suggestions
- Use the query-string parameter to force a request caching

```
GET /blogs/_search?request_cache=true
{
  "query": {
    "query_string": {
      "query": "*_source*"
    }
  }
}
```

The whole JSON body is used as the cache key



caches even when size > 0



Chapter Review

~05-Mar-2019~

Summary

- A **shard** is a single instance of Lucene that consists of segments
- A **segment** is a package of many different data structures representing an inverted index for each field
- Any document write operation is written to the **transaction log** after being processed by the internal Lucene index
- During indexing, a segment is created every 1 second (by default)
- Sort and Aggregation of strings should be performed on keyword fields (uses doc_values).
- A query in a **filter** context can be **cached** by Elasticsearch to improve performance



Quiz

1. **True or False:** Increasing the `refresh_interval` value (ex: 30s) is a good practice before indexing a lot of documents.
2. Name three things that a segment stores?
3. **True or False:** You should occasionally invoke `_forcemerge` if you are indexing a lot of documents continuously.
4. **True or False:** After an index operation documents might not be searchable up to 1 second.
5. What happens if you use `?refresh=wait_for` in an index request. Explain one use case that benefits from it?
6. **True or False:** Every write operation is recorded in the translog and the translog is fsynced to disk.
7. When should you run a synced flush?



Lab 1

~05-Mar-2019~0

Elasticsearch Internals

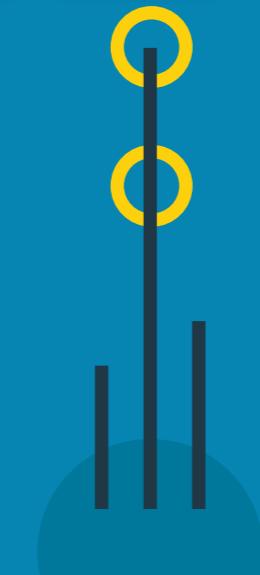


Chapter 2

Field Modeling

05-Mar-2019 - 0

- 1 Elasticsearch Internals
- 2 Field Modeling
- 3 Fixing Data
- 4 Advanced Search & Aggregations
- 5 Cluster Management
- 6 Capacity Planning
- 7 Document Modeling
- 8 Monitoring and Alerting
- 9 From Dev to Production



Topics covered:

- The Need for Modeling
- Modeling Granular Fields
- Modeling Ranges
- Mapping Parameters
- Dynamic Templates
- Controlling Dynamic Fields

~05-Mar-2019~0



The Need for Modeling

~05-Mar-2019~

The Blogs Mapping

```
"mappings": {  
    "_doc": {  
        "properties": {  
            "author": {  
                "type": "text",  
                "fields": {  
                    "keyword": {  
                        "type": "keyword",  
                        "ignore_above": 256  
                    }  
                }  
            },  
            "category": {  
                "type": "text",  
                "fields": {  
                    "keyword": {  
                        "type": "keyword",  
                        "ignore_above": 256  
                    }  
                }  
            },  
            "publish_date": {  
                "type": "date"  
            },  
            ...  
        }  
    }  
}
```

Most of the fields are the default “text” and “keyword”, which does not make sense for some fields

The Logs Mapping

```
"mappings": {  
  "doc": {  
    "properties": {  
      "@timestamp": {  
        "type": "date"  
      },  
      "geoip": {  
        "properties": {  
          "city_name": {  
            "type": "text",  
            "fields": {  
              "keyword": {  
                "type": "keyword",  
                "ignore_above": 256  
              }  
            }  
          },  
          "continent_code": {  
            "type": "text",  
            "fields": {  
              "keyword": {  
                "type": "keyword",  
                "ignore_above": 256  
              }  
            }  
          },  
          ...  
        }  
      }  
    }  
  }  
}
```

The **@timestamp** field is a date, which is great...

...but most of the other fields are **text/keyword**



Some Fixes are Easy

Choose a more appropriate data type:

```
"status_code": 200
```

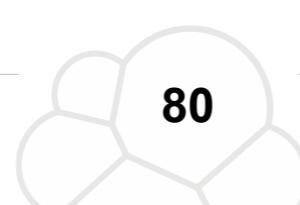
```
"status_code": {  
    "type": "short"  
}
```

25-Mar-2019-0

Clean-up some of the string fields:

```
"language": {  
    "code": "fr-fr"  
}
```

```
"code": {  
    "type": "keyword"  
}
```



Some Fixes Require More Design

- For example, the “**locale**” field in the blogs dataset is a comma-separated list of values
 - Lists are easier to search if they are indexed as an array:

A diagram illustrating the transformation of a field. On the left, a box contains the JSON path `"locales": "de-de,fr-fr"`. A teal curved arrow points from this box to another box on the right. The right box contains two parts: `"locales": { "type": "keyword" }` and `"locales": ["de-de", "fr-fr"]`.

- The “**user-agent**” field in the logging dataset is difficult to search in its current format:

A diagram illustrating the difficulty of searching a field. On the left, a box contains the JSON path `"user_agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/64.0.3282.186 Safari/537.36"`. A teal curved arrow points from this box to a teal rectangular button containing a white question mark.

Modeling Granular Fields

.05Mar2019 - 0

Use Case for Granular Fields

- Suppose we include an Elastic Stack version to each blog post:

```
"version": "6.2.1"
```

- And we want to add a facet so users can filter blogs by a desired version of the Elastic Stack
 - Searching an exact version like **6.2.1** would be easy enough
 - But how would you query the “**version**” field for “**5.4**” or “**6.x**”?
- We could use some complicated regular expressions
 - or we could ***map and index the field better!***



Modeling Granular Fields

- Adding a level of *granularity* can make it easier to answer questions about a field:

```
PUT blogs
{
  "mappings": {
    "_doc": {
      "properties": {
        "version": {
          "properties": {
            "display_name": {
              "type": "keyword"
            },
            "major": {
              "type": "byte"
            },
            "minor": {
              "type": "byte"
            },
            "bugfix": {
              "type": "byte"
            }
          }
        }
      }
    }
  }
}
```

```
PUT blogs/_doc/1
{
  "version": {
    "display_name": "6.2.1",
    "major": 6,
    "minor": 2,
    "bugfix": 1
  }
}
```



Searching Granular Fields

- Now we can search a version number easily at any desired level:

```
GET blogs/_search
{
  "query": {
    "bool": {
      "filter": [
        {
          "match": {
            "version.major": 5
          }
        },
        {
          "match": {
            "version.minor": 4
          }
        }
      ]
    }
  }
}
```

*"I'm searching
for blogs about
version 5.4"*



Modeling Ranges

~05-Mar-2010

Use Case for Ranges

- The blog posts currently have a **publish_date** field
- Suppose we want to add an **end_date** as well:

```
"publish_date": "2017-11-10",  
"end_date": "2018-11-10"
```

- This granular approach above would work fine
 - but there is a datatype (introduced in Elasticsearch 5.2) that is a nice solution for ranges like this...



Range Data Types

- *Range data types* allow you to define a field using a lower and upper bound
- Works with different types:
 - `integer_range`
 - `float_range`
 - `long_range`
 - `double_range`
 - `date_range`
 - `ip_range`

~05-Mar-2019~0



Defining a Range Type

- Let's test a `date_range` field for our blog posts:

```
PUT test_ranges
{
  "mappings": {
    "_doc": {
      "properties": {
        "publish_range": {
          "type": "date_range"
        }
      }
    }
  }
}
```



Indexing a Range Type

- Range data types are defined using a lower and/or upper bound:

```
PUT test_ranges/_doc/1
{
  "publish_range": {
    "gte": "2017-11-10",
    "lt": "2018-11-10"
  }
}
```

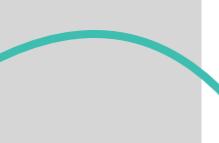
This document defines both an upper and lower bound for “publish_range”



Querying a Range Type

- Use a **range** query to search a range field:

```
GET test_ranges/_search
{
  "query": {
    "range": {
      "publish_range": {
        "gte": "2018-04-01"
      }
    }
  }
}
```

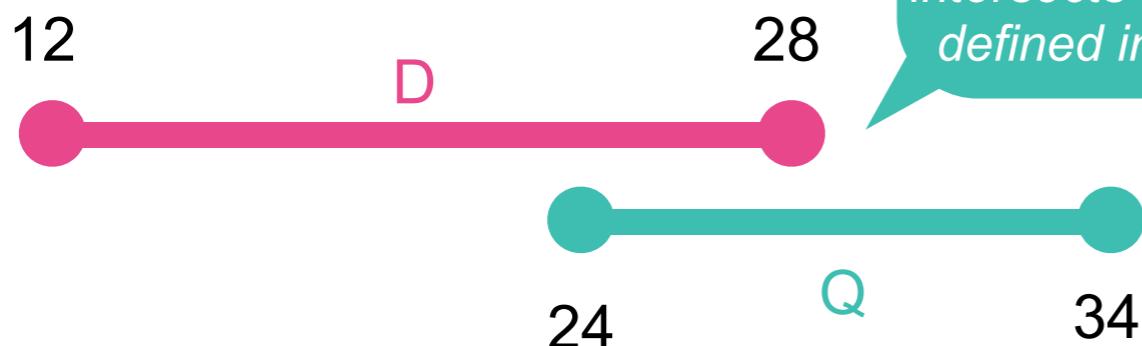


```
"hits": {
  "total": 1,
  "max_score": 1,
  "hits": [
    {
      "_index": "test_ranges",
      "_type": "_doc",
      "_id": "1",
      "_score": 1,
      "_source": {
        "publish_range": {
          "gte": "2017-11-10",
          "lt": "2018-11-10"
        }
      }
    }
  ]
}
```



The relation Parameter

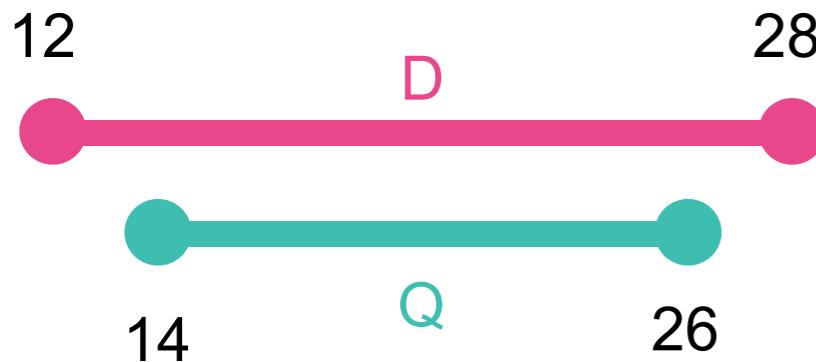
intersects (default)



"Give me all docs where the range defined in the doc intersects with the range defined in query"

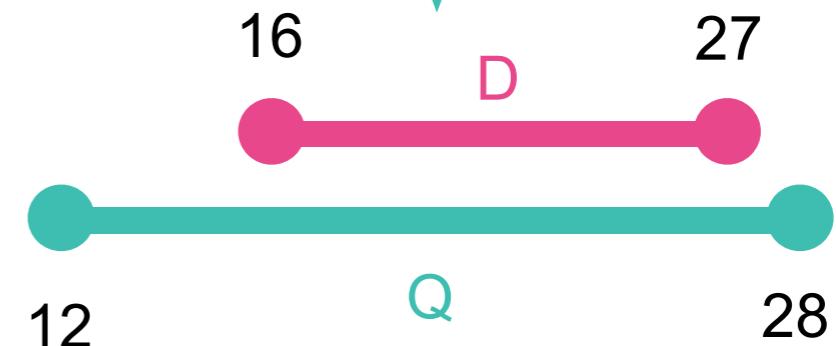
```
GET my_index/_search
{
  "query": {
    "range": {
      "FIELD": {
        "gte": 23,
        "lte": 43,
        "relation": "intersects"
      }
    }
  }
}
```

contains



"Give me all docs where the range defined in the doc contains the range defined in query"

within



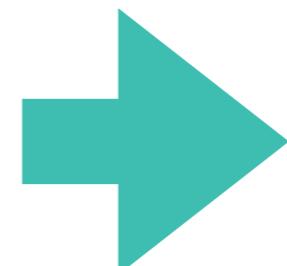
"Give me all docs where the range defined in the doc is within the range defined in query"

The relation Parameter (Example)

```
GET my_range_index/_search
{
  "query": {
    "range": {
      "author_age_range": {
        "gte": 23,
        "lte": 43,
        "relation": "intersects"
      }
    }
  }
}
```

intersects | within | contains

_id	blog_title	author_age
1	“Hello, Kibana”	15 to 24
2	“Where is my log?”	25 to 34
3	“Ingestion problems?”	35 to 44
4	“Aggregate this!”	45 to 54



query_age_range	relation	hits
23 to 43	intersects	1,2,3
25 to 45	within	2,3
35 to 40	contains	3



Mapping Parameters

-05-Mar-2019-

Mapping Parameters

- There are various parameters that can be applied to fields when defining your mappings:

```
"mappings": {  
  "doc": {  
    "properties": {  
      "originalUrl": {  
        "type": "text",  
        "analyzer": "my_url_analyzer"  
      }  
    }  
  }  
...  
}
```

An example of a
mapping parameter

- We will discuss a few mapping parameters now. See the docs for a complete list:
 - <https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping-params.html>

Not Indexing a Field

- Suppose we know that we will *not run any queries* on the **http_version** field of our access logs
 - it is possible to not index a specific field
 - the field is still returned in the **_source** and can be used in aggregations, but the field will not be queryable

```
"mappings": {  
    "doc": {  
        "properties": {  
            "http_version": {  
                "type": "keyword",  
                "index": false  
            }  
        ...  
    }  
}
```

“**http_version**” will
not be indexed



Disabling Doc Values

- Or maybe we will *not run any aggregations* on the **http_version** field of our access logs
 - it is also possible to not have doc_values to a specific field
 - the field is still returned in the **_source** and can be used in queries, but the field cannot be used in aggregations

```
"mappings": {  
    "doc": {  
        "properties": {  
            "http_version": {  
                "type": "keyword",  
                "doc_values": false  
            }  
        ...  
    }  
}
```

“**http_version**” will
not have doc_values



Disabling a Field

- When a field is not indexed, its value is still stored and available for aggregations
- Another option is to ***completely disable a field:***
 - *you cannot query or aggregate this field*
 - but this field is still be returned in the `_source`

```
PUT my_logs/_doc/_mapping
{
  "properties": {
    "url": {
      "enabled": false
    }
  }
}
```

“url” will not be indexed nor stored in doc_values. It will still be stored in `_source`

Disabling an Object

- Setting “enable” to `false` is useful when you want to skip the indexing of an **entire JSON object** in your document
 - suppose you will never perform any queries or aggregations on the fields of the “language” object in the access logs:

```
"mappings": {  
  "doc": {  
    "properties": {  
      "language": {  
        "enabled": false  
      }  
    ...  
  }
```

The “language” object
is disabled

```
{  
  "@timestamp": "2017-05-19T00:47:44.633Z",  
  "user_agent": "Amazon CloudFront",  
  "language": {  
    "code": "en-us",  
    "url": "/blog/category/releases"  
  },  
  "runtime": "454ms"  
}
```



Disabling _all

- `_all` is a special catch-all field which concatenates the values of all of the other fields into ***one big string***
 - the `_all` field is being removed in future versions of Elasticsearch
 - unable to use it for new indices in Elasticsearch 6
- You can disable it to save disk space if your application is not using `_all`

```
PUT blogs
{
  "mappings": {
    "_doc": {
      "_all
```

Only relevant for
Elasticsearch 5.x and
earlier

Use Case for `copy_to`

- The access logs dataset has several fields representing the location of the event:

```
"region_name": "Victoria",
"country_name": "Australia",
"city_name": "Surrey Hills"
```

- Suppose we want to frequently search all three of these fields:
 - we could run a **bool** query with **must** or **should** clauses,
 - or we could copy all three values to a single field during indexing using **copy_to**...



The copy_to Parameter

```
"mappings": {  
  "_doc": {  
    "properties": {  
      "region_name": {  
        "type": "keyword",  
        "copy_to": "locations_combined"  
      },  
      "country_name": {  
        "type": "keyword",  
        "copy_to": "locations_combined"  
      },  
      "city_name": {  
        "type": "keyword",  
        "copy_to": "locations_combined"  
      },  
      "locations_combined": {  
        "type": "text"  
      }  
    }  
  }  
}
```

During indexing, the values will be copied to the **“locations_combined”** field



The copy_to Parameter

- The `locations_combined` field is not a part of `_source`, but it is indexed
 - so you can query on it:

```
GET weblogs/_search
{
  "query": {
    "match": {
      "locations_combined": "victoria australia"
    }
  }
}
```

I am searching for events in “Victoria Australia”

```
"hits": [
  {
    "_index": "weblogs",
    "_type": "_doc",
    "_id": "1",
    "_score": 0.5753642,
    "_source": {
      "region_name": "Victoria",
      "country_name": "Australia",
      "city_name": "Surrey Hills"
    }
  }
]
```



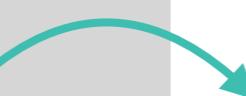
Null Values

- When a field is set to **null**, it is treated as though that field has no value:

```
PUT ratings/_doc/1
{
  "rating": null
}
```

```
PUT ratings/_doc/2
{
  "rating": 5.0
}
```

```
GET ratings/_search?size=0
{
  "aggs": {
    "average_rating": {
      "avg": {
        "field": "rating"
      }
    }
  }
}
```



```
"aggregations": {
  "average_rating": {
    "value": 5
  }
}
```



Specifying a Default Value for nulls

- Use the `null_value` parameter to assign a value to a field if it is `null`
 - The `_source` is not altered, but the value of `null_value` is indexed

```
PUT ratings
{
  "mappings": {
    "_doc": {
      "properties": {
        "rating": {
          "type": "float",
          "null_value": 1.0
        }
      }
    }
  }
}
```

If “`rating`” is `null`, then
1.0 will be indexed for
that field



Specifying a Default Value for nulls

- Notice the average changes now:

```
PUT ratings/_doc/1
{
  "rating": null
}
```

```
PUT ratings/_doc/2
{
  "rating": 5.0
}
```

```
GET ratings/_search?size=0
{
  "aggs": {
    "average_rating": {
      "avg": {
        "field": "rating"
      }
    }
  }
}
```

```
"aggregations": {
  "average_rating": {
    "value": 3
  }
}
```



Coercing Data

- By default, Elasticsearch attempts to **coerce** data to match the data type of the field
 - For example, suppose the “rating” field is a “long”:

```
PUT ratings/_doc/1
{
  "rating": 4
}
```

All three PUT
commands work fine

```
PUT ratings/_doc/2
{
  "rating": "3"
}
```

```
PUT ratings/_doc/3
{
  "rating": 4.5
}
```

A “sum” aggregation on
“rating” returns “11”

Disabling Coercion

- You can ***disable coercion*** if you do not want Elasticsearch to try and clean up your dirty fields:

```
"mappings": {  
    "_doc": {  
        "properties": {  
            "rating": {  
                "type": "long",  
                "coerce": false  
            }  
        }  
    }  
}
```

Set the “coerce” parameter to **false**

```
PUT ratings/_doc/1  
{  
    "rating": 4  
}  
PUT ratings/_doc/2  
{  
    "rating": "3"  
}  
PUT ratings/_doc/3  
{  
    "rating": 4.5  
}
```

Works fine

Fails

Fails

The `_meta` Field

- Using the “`_meta`” field, you can put any custom metadata you want in your mapping
 - Any “`_meta`” data is associated with the type (not your individual documents)

```
PUT blogs/_mapping/_doc
{
  "_meta": {
    "blog_mapping_version": "2.1"
  }
}
```

Store any application-specific
JSON here...



Dynamic Templates

· 05-Mar-2019 ·

Use Case for Dynamic Templates

- Suppose you have documents with a *large number of fields*,
- or documents with *dynamic field names* not known at the time of your mapping definition
 - and nested key/value pairs are not a good solution for your use case
- Using *dynamic templates*, you can define a field's mapping based on:
 - the field's **datatype**,
 - the **name** of the field, or
 - the **path** to the field

05-Mar-2019



Defining a Dynamic Template

- Suppose you want any unmapped string fields to be mapped as type “**keyword**” by default:

```
PUT test2
{
  "mappings": {
    "_doc": {
      "dynamic_templatesmatch_mapping_type": "string",
            "mapping": {
              "type": "keyword"
            }
          }
        }
      ]
    }
  }
}
```

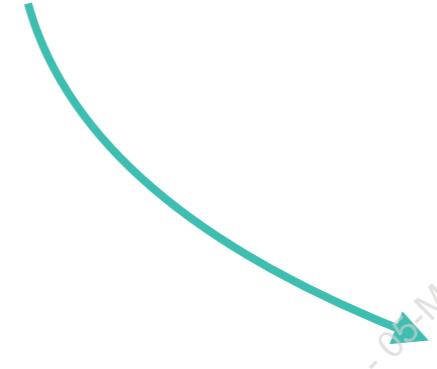
If the field is a string, map it as “**keyword**”



Test Our Template

```
POST test2/_doc
{
  "blog_reaction": ":thumbsup:"
}

GET test2/_mapping
```



```
"properties": {
  "blog_reaction": {
    "type": "keyword"
  }
}
```



Matching Field Names

- Use the “match” parameter to match a field name to a mapping:

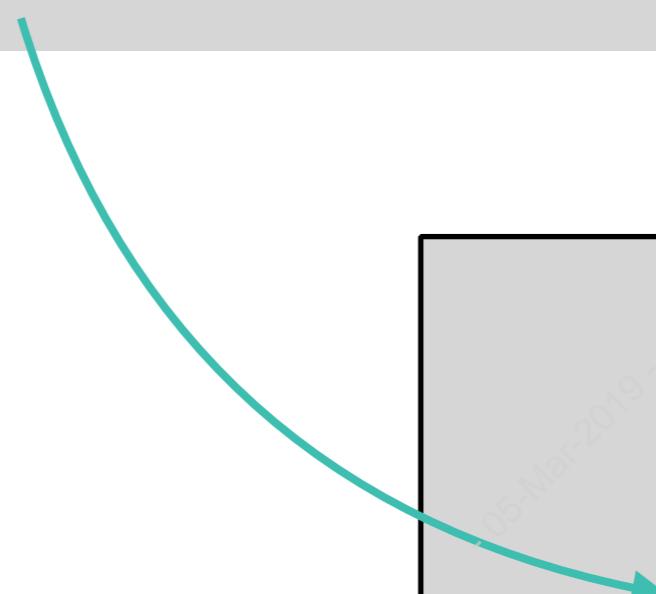
```
PUT test2/_doc/_mapping
{
  "dynamic_templates": [
    {
      "my_float_fields": {
        "match": "f_*",
        "mapping": {
          "type": "float"
        }
      }
    }
  ]
}
```

If an unmapped field name starts with “f_”, then it will be mapped as a float



Matching Field Names

```
POST test2/_doc/  
{  
  "f_avg_response_time": "34.8"  
}
```



```
"properties": {  
  "blog_reaction": {  
    "type": "keyword"  
  },  
  "f_avg_response_time": {  
    "type": "float"  
  }  
}
```



Controlling Dynamic Fields

.05Mar2019 - 0

Strict Mappings

- By default, if a document is indexed with an unexpected field, the mapping is *dynamically* modified
- In production, you will likely define your mappings prior to any indexing
 - and you probably *do not want your mappings to change*

```
POST blogs/_doc/  
{  
  "some_new_field": "This is quite unexpected"  
}
```

*I will simply add
this field to your
mapping.*

Controlling the dynamic feature...

- You can control the effect of new fields added to a mapping using the “dynamic” property (three options):

	<i>doc indexed?</i>	<i>fields indexed?</i>	<i>mapping updated?</i>
“true”	✓	✓	✓
“false”	✓	✗	✗
“strict”	✗		

~05-Mar-2019 ~0

```
PUT blogs/_doc/_mapping
{
  "dynamic": "strict"
```

Do not index a document
that contains fields not
already defined in the
mapping



If dynamic is set to “strict”...

- ...then indexing a document with undefined fields generates an error:

```
POST blogs/_doc/  
{  
  "some_other_field": "This wont't work"  
}
```

```
{  
  "error": {  
    "root_cause": [  
      {  
        "type": "strict_dynamic_mapping_exception",  
        "reason": "mapping set to strict, dynamic  
introduction of [some_other_field] within [_doc] is not  
allowed"  
      },  
      {  
        "type": "strict_dynamic_mapping_exception",  
        "reason": "mapping set to strict, dynamic introduction  
of [some_other_field] within [_doc] is not allowed"  
      },  
      "status": 400  
    ]  
  }  
}
```



Adding Fields to a Mapping

- If “dynamic” is “strict”, you can still add a field by modifying the mapping directly
 - you just can’t add a field to the mapping dynamically

```
PUT blogs/_doc/_mapping
{
  "properties": {
    "some_other_field": {
      "type": "text"
    }
  }
}
```

Add a new field named
“**some_other_field**” to the
mapping

05-Mar-2019 - 0

The **POST** command on the
previous slide will work now



Chapter Review

~05-Mar-2019~

Summary

- Adding a ***level of granularity*** can make it easier to answer questions about a field
- ***Range data types*** allow you to define a field using a lower and upper bound
- Setting “**index**” to **false** ***disables indexing***, but its value is still stored and available for aggregations
- Setting “**enabled**” to **false** completely ***disables a field*** so that it is not indexed or available for searches or aggs
- Use the “**null_value**” parameter to assign a value to a field if it is **null**
- Using ***dynamic templates***, you can define a field’s mapping based on its name or datatype
- You can control the effect of new fields added to a mapping using the “**dynamic**” property



Quiz

1. How might the following field be modeled more effectively for searching and aggregating dev environments?

```
"dev_environment": "Emacs; Notepad++; PyCharm"
```

2. How might the following field be modeled more effectively?

```
"interview_likelihood": "90%"
```

3. How would you map a field that you never need to use for searches or aggregations?

4. How would you configure an index so that it rejects documents that contain fields not defined in its mapping?

5. What is the default value of the “**relation**” parameter in a **range** query?

Lab 2

Field Modeling

~05-Mar-2019~0



Chapter 3

Fixing Data

05-Mar-2019 - 0

- 1 Elasticsearch Internals
- 2 Field Modeling
- 3 Fixing Data
- 4 Advanced Search & Aggregations
- 5 Cluster Management
- 6 Capacity Planning
- 7 Document Modeling
- 8 Monitoring and Alerting
- 9 From Dev to Production



Topics covered:

- Tools for Fixing Data
- Fixing Mappings
- Reindexing Tips
- Picking Up Mapping Changes
- Fixing Fields

~05-Mar-2019~0



Fixing Data

- In the previous chapter we discussed
 - different problems that a dataset can have
 - best practices for field modeling
- Now we will discuss techniques to fix data issues, like:
 - fields with incorrect types
 - comma separated string should be an array
 - changing an existing mapping
- First, let's look into tools that can help you with the task...



Tools for Fixing Data

-05-Mar-2019-

Overview of Painless

- **Painless** is a scripting language designed specifically for use with Elasticsearch
 - It's fast, secure, and has a Groovy-like syntax
- Supports all of Java's data types and a subset of the Java API:
 - <https://www.elastic.co/guide/en/elasticsearch/reference/current/painless-api-reference.html>
- Languages like Groovy, Javascript, and Python are no longer available for scripting as of Elasticsearch 6.0



Use Cases for Painless

- ***Accessing document fields*** for various reasons:
 - update/delete a field in a document
 - perform computations on fields before returning them
 - customize the score of a document
 - working with aggregations
- ***Ingest Node***
 - execute a script within an ingest pipeline
- ***Reindex API***
 - manipulate data as it is getting reindexed
- Lots of use cases, as long as you use it wisely!



Simple Use Case for a Script

- We will start with a simple use case of updating a field
- ***Be warned!***
 - Updating a field results in the ***entire document getting deleted and reindexed***
 - We will discuss a better solution later in the course if you need to update a field frequently
- Let's add a field for the number of views of a blog post:

```
PUT my_index/_doc/1
{
  "blog_id": "h81CKmIBCLh5xF6i7Y2f",
  "num_of_views": 3
}
```

Let's write a script to
increment “**num_of_views**”



An Inline Script

- There are two ways to run a Painless script: ***inline*** or ***stored***
 - The following example demonstrates an inline script that updates the “num_of_views” field:

```
POST my_index/_doc/1/_update
{
  "script": {
    "source": "ctx._source.num_of_views += params.new_views",
    "params": {
      "new_views": 2
    }
  }
}
```

“source” is
the code

Handy for short scripts - just
write the code inline

“params” is for
optional parameters



A Stored Script

- Scripts can be **stored** in the cluster state
 - and invoked later using the script's ID:

```
POST _scripts/add_new_views ← id of the script
{
  "script": {
    "lang": "painless",
    "source": "ctx._source.num_of_views += params.new_views"
  }
}
```

```
POST my_index/_doc/1/_update
{
  "script": {
    "id": "add_new_views",
    "params": {
      "new_views": 2
    }
  }
}
```

the **params** are passed
to the script



Accessing Fields

- The syntax in Painless for accessing a field's value depends on the context:

Context	Syntax for accessing fields
Ingest node: access fields using <code>ctx</code>	<code>ctx.field_name</code>
Updates: use the <code>_source</code> field	<code>ctx._source.field_name</code>
Search and aggs: if doc values are enabled, <code>doc</code> is very efficient to access a field	<code>doc['field_name'].value</code> if you expect an array, use <code>doc['field_name']</code>

<https://www.elastic.co/guide/en/elasticsearch/painless/current/painless-contexts.html>



Script Caching

- The first time Elasticsearch sees a new script, it compiles it and stores the compiled version in a **cache**
 - both inline and stored scripts are stored in the cache
- A new script can evict a cached script
 - the default size of the cache is 100 scripts
 - configurable using **script.cache.max_size**
 - or set a timeout using **script.cache.expire**



Compilation can be Expensive

- If you compile too many unique scripts within a small amount of time, Elasticsearch will reject the new dynamic scripts
 - and throw a ***circuit_breaking_exception*** error
- By default, up to 75 compilations per 5-minute window (“**75/5m**”) can be compiled
 - configured by **script.max_compilations_rate**
 - may be able to avoid this issue by **using parameters...**



Parameters in Scripts

- Scripts have to match exactly to take advantage of the cache
 - Be careful with literals in your scripts:

```
"script": {  
  "source": "ctx._source.num_of_views += 2"  
}
```

```
"script": {  
  "source": "ctx._source.num_of_views += 3"  
}
```

Two different scripts, so
two compilations needed

- Use *parameters* instead...

```
"script": {  
  "source": "ctx._source.num_of_views += params.new_views"  
}
```



Reindexing APIs

- We will also use the ***Reindex*** and ***Update By Query*** APIs in this chapter to fix data
 - The **_reindex** endpoint is for reindexing from one index to another
 - The **_update_by_query** endpoint is for reindexing into the same index
- And for demonstration purposes, we will use the **_delete_by_query** endpoint as well
 - deletes all documents that are hits for a given query



Fixing Mappings

~05-Mar-2019 10:00

Fixing Mappings

- Our **blogs** fields are not mapped very well
 - In particular, the default “**text**”/“**keyword**” mapping of our string fields needs some cleaning up:

```
"category": {  
    "type": "text",  
    "fields": {  
        "keyword": {  
            "type": "keyword",  
            "ignore_above": 256  
        }  
    }  
},  
"content": {  
    "type": "text",  
    "fields": {  
        "keyword": {  
            "type": "keyword",  
            "ignore_above": 256  
        }  
    }  
},  
...  
05-Mar-2019
```

“category” only has 10 distinct values, making it a good candidate for “**keyword**” only

“content” is a large amount of text, so “**keyword**” seems unnecessary

Fixing Mappings

- You can not change the data type of a field in a mapping
 - We will have to define a new index for our blogs with the desired data types:

```
PUT blogs_fixed
{
  "mappings": {
    "doc": {
      "properties": {
        "author": {
          "type": "text",
          "fields": {
            "keyword": {
              "type": "keyword",
              "ignore_above": 256
            }
          }
        },
        "category": {
          "type": "keyword"
        },
        "content": {
          "type": "text"
        },
        "locales": {
          "type": "keyword"
        },
        ...
      }
    }
  }
}
```

We will index “author” as both “text” and “keyword”

...but clean up the other string fields to fit our dataset and use case better



Applying the New Mapping

- Now we need to copy the old blogs into our new index
 - which can be easily accomplished using the *Reindex API*
- The `_reindex` command indexes the documents from the “source” index into the “dest” index
 - done in batches, with a default batch size of 1,000

```
POST _reindex
{
  "source": {
    "index": "my_source_index",
    "query": { ... },
    ...
  },
  "dest": {
    "index": "my_destination_index"
  }
}
```

Optional “query” to specify which documents to reindex



Let's Reindex the Blogs

- All documents from “blogs” will be reindexed into “**blogs_fixed**”:

```
POST _reindex
{
  "source": {
    "index": "blogs"
  },
  "dest": {
    "index": "blogs_fixed"
  }
}
```

Test the New Mapping

- We should be able to aggregate on “category”, since it is a “keyword” now

```
GET blogs_fixed/_search
{
  "size": 0,
  "aggs": {
    "top_categories": {
      "terms": {
        "field": "category",
        "size": 5
      }
    }
  }
}
```



```
"aggregations": {
  "top_categories": {
    "doc_count_error_upper_bound": 0,
    "sum_other_doc_count": 305,
    "buckets": [
      {
        "key": "Engineering",
        "doc_count": 440
      },
      {
        "key": "",
        "doc_count": 333
      },
      {
        "key": "Releases",
        "doc_count": 238
      },
    ]
  }
}
```



Reindexing Tips

~05-Mar-2019 10

Dealing with Versions

- Suppose something went wrong while we were reindexing the **blogs** index
 - we do not want to have to start over!
- By default, `_reindex` blindly overwrites existing documents in the destination
 - this is the behavior of setting the “`version_type`” parameter to “`internal`” (the default value)
- Let’s discuss a few tips for dealing with document versions and reindexing...

Behavior of “internal”

- When “`version_type`” equals “internal”, version numbers are ignored
 - existing documents in the destination index are overwritten

<i>source index</i>	<i>dest index</i>	“internal” behavior
<code>_id: 456</code> <code>_version: 4</code>	<code>_id: 456</code> <code>_version: 2</code>	overwritten
<code>_id: 123</code> <code>_version: 1</code>	<code>_id: 123</code> <code>_version: 3</code>	overwritten
<code>_id: abc</code> <code>_version: 2</code>	<code>_id: abc</code> <code>_version: 2</code>	overwritten
<code>_id: 789</code> <code>_version: 1</code>	(no doc with id “789” exists)	created



Behavior of “external”

- When “`version_type`” equals “`external`”, the version number of the document in the source index is preserved
 - a “newer” document overwrites an “older” document

<i>source index</i>	<i>dest index</i>	“ <i>external</i> ” behavior
<code>_id: 456</code> <code>_version: 4</code>	<code>_id: 456</code> <code>_version: 2</code>	overwritten
<code>_id: 123</code> <code>_version: 1</code>	<code>_id: 123</code> <code>_version: 3</code>	exception
<code>_id: abc</code> <code>_version: 2</code>	<code>_id: abc</code> <code>_version: 2</code>	exception
<code>_id: 789</code> <code>_version: 1</code>	(no doc with id “789” exists)	created



“external” Version Type

```
POST _reindex
{
  "source": {
    "index": "blogs"
  },
  "dest": {
    "index": "blogs_fixed",
    "version_type": "external"
  }
}
```

Let's run the same reindex again, but change “version_type” to “external”

We get 1,000 failures because the documents already exist in the destination

"failures": [

```
{
  "index": "blogs_fixed",
  "type": "doc",
  "id": "Cc1CKmIBCLh5xF6i7Y2b",
  "cause": {
    "type": "version_conflict_engine_exception",
    "reason": "[doc][Cc1CKmIBCLh5xF6i7Y2b]: version conflict, current version [1] is higher or equal to the one provided [1]",
    "index_uuid": "HUkf-mfCQfOghnqSIi5kBQ",
    "shard": "0",
    "index": "blogs_fixed"
  },
  "status": 409
},
```



Version Conflicts

- Notice in the previous example that only 1 batch of 1,000 documents was reindexed because a failure occurred
 - set “**conflicts**” equal to “**proceed**” to ignore conflicts
 - keeps a running count of them instead

```
POST _reindex
{
  "source": {
    "index": "blogs"
  },
  "dest": {
    "index": "blogs_fixed",
    "version_type": "external"
  },
  "conflicts": "proceed"
}
```



```
{
  "took": 65,
  "timed_out": false,
  "total": 1594,
  "updated": 0,
  "created": 0,
  "deleted": 0,
  "batchesversion_conflicts
```



Updating a Document

- To better demonstrate “**external**”, let’s change some of the documents in the “**blogs**” index
 - we will use an **_update_by_query** to modify every blog whose “**category**” is a blank string
- This will increase the “**_version**” number of 333 blogs:

```
POST blogs/_update_by_query
{
  "query": {
    "match": {
      "category.keyword": ""
    }
  },
  "script": {
    "source": "ctx._source.category = \"None\""
  }
}
```

blank string

333 documents are updated



Reindex Again with “external”

- This time just the changed “blogs” are reindexed into “blogs_fixed”

```
POST _reindex
{
  "source": {
    "index": "blogs"
  },
  "dest": {
    "index": "blogs_fixed",
    "version_type": "external"
  },
  "conflicts": "proceed"
}
```

05-Mar-2019, 08:00:00

```
{
  "took": 130,
  "timed_out": false,
  "total": 1594,
  "updated": 333,
  "created": 0,
  "deleted": 0,
  "batches": 2,
  "version_conflicts": 1261,
  "noops": 0,
  "retries": {
    "bulk": 0,
    "search": 0
  },
  ...
}
```

333 blogs were
“updated”



Using “op_type”

- We can set “op_type” to “create” if we are not worried about version numbers
 - only missing documents get reindexed

<i>source index</i>	<i>dest index</i>	“external” behavior
<code>_id: 456</code> <code>_version: 4</code>	<code>_id: 456</code> <code>_version: 2</code>	exception
<code>_id: 123</code> <code>_version: 1</code>	<code>_id: 123</code> <code>_version: 3</code>	exception
<code>_id: abc</code> <code>_version: 2</code>	<code>_id: abc</code> <code>_version: 2</code>	exception
<code>_id: 789</code> <code>_version: 1</code>	(no doc with id “789” exists)	created



Delete By Query API

- Let's do another demo, by first deleting some documents
 - The `_delete_by_query` endpoint will delete all the documents from an index that hit the provided query

```
POST blogs_fixed/_delete_by_query
{
  "query": {
    "range": {
      "publish_date": {
        "lte": "2016"
      }
    }
  }
}
```

Deletes 812 blogs from
2016 and earlier



Setting “op_type” to “create”

```
POST _reindex
{
  "source": {
    "index": "blogs"
  },
  "dest": {
    "index": "blogs_fixed",
    "op_type": "create"
  },
  "conflicts": "proceed"
}
```

~05-Mar-2019~0

Notice “op_type” throws
an exception if the
document already exists
in the target

```
{
  "took": 257,
  "timed_out": false,
  "total": 1594,
  "updated": 0,
  "created": 812,
  "deleted": 0,
  "batches": 2,
  "version_conflicts": 782,
  "noops": 0,
  "retries": {
    "bulk": 0,
    "search": 0
  },
}
```

812 documents were
reindexed



Picking Up Mapping Changes

.05-Nov-2019 - 0

Adding a Multi-Field to a Mapping

- Suppose we want to add a multi-field to the mapping of the **blogs_fixed** index
 - Specifically, suppose we want to analyze the “**content**” field in a new way:

```
PUT blogs_fixed/_mapping/_doc
{
  "properties": {
    "content": {
      "type": "text",
      "fields": {
        "english": {
          "type": "text",
          "analyzer": "english"
        }
      }
    }
  }
}
```

Add a new multi-field named “**english**” that uses the **english** analyzer



Picking Up a New Multi-Field

- The mapping has changed, but existing documents already in the index do not have this new multi-field:

```
GET blogs_fixed/_search
{
  "query": {
    "match": {
      "content.english": "performance tips"
    }
  }
}
```

No hits, because no documents have this field yet



Update By Query

- Run an `_update_by_query` to have existing documents pick up the new “`content.english`” field:

```
POST blogs_fixed/_update_by_query
```

```
GET blogs_fixed/_search
{
  "query": {
    "match": {
      "content.english": "performance tips"
    }
  }
}
```

426 hits



Adding Your Own Reindex Batch Field

- What happens if something fails during the reindexing?
 - We probably do not want to start over from the beginning!
- You can add a field to each document just for a reindex job
 - a simple numeric field that is set to 1 if a document has been reindexed successfully:

```
PUT blogs_fixed/_doc/_mapping
{
  "properties": {
    "reindexBatch": {
      "type": "short"
    }
  }
}
```

Add a field to our mapping just to track the reindexing

Updating Our Reindex Batch Field

```
POST blogs_fixed/_update_by_query
{
  "query": {
    "bool": {
      "must_not": [
        {
          "range": {
            "reindexBatch": {
              "gte": 1
            }
          }
        }
      ]
    },
    "script": {
      "source": """
if(ctx._source.containsKey("content")) {
  ctx._source.content_length = ctx._source.content.length();
} else {
  ctx._source.content_length = 0;
}
ctx._source.reindexBatch=1;
"""
    }
  }
}
```

Find only the documents that need updating

Update the batch number

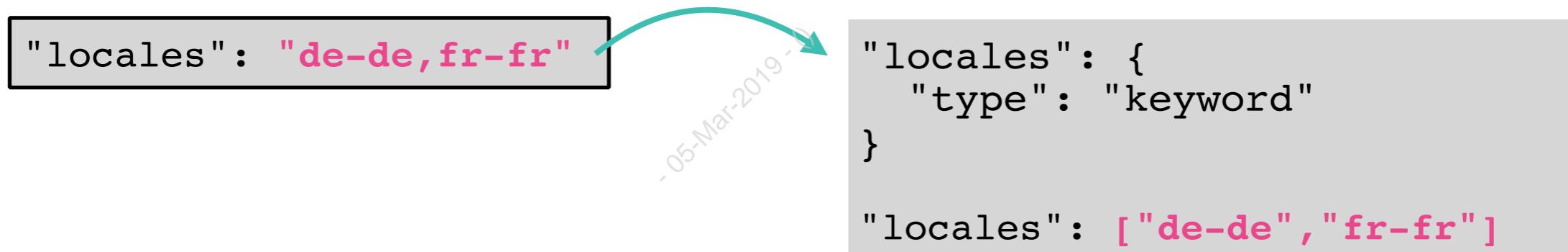


Fixing Fields

~05-Mar-2019~0

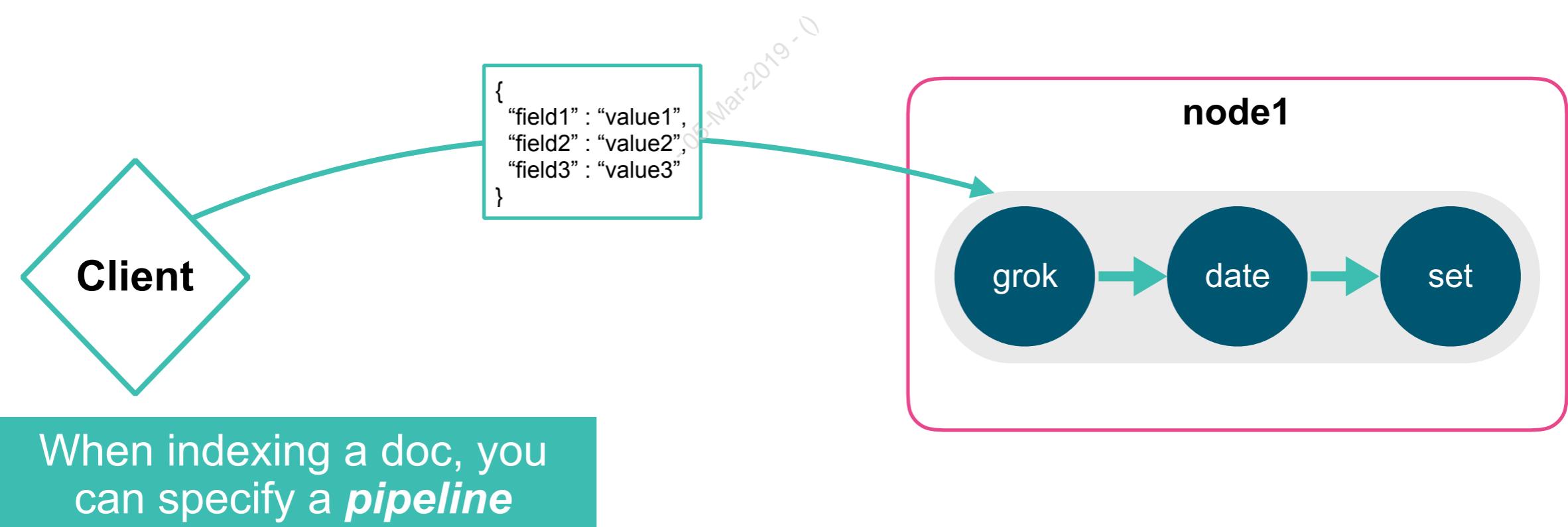
Splitting a Field

- Let's fix the “**locales**” field by splitting it into an array of “**keyword**” values
 - We could figure this out with a script, but there is an easier way...



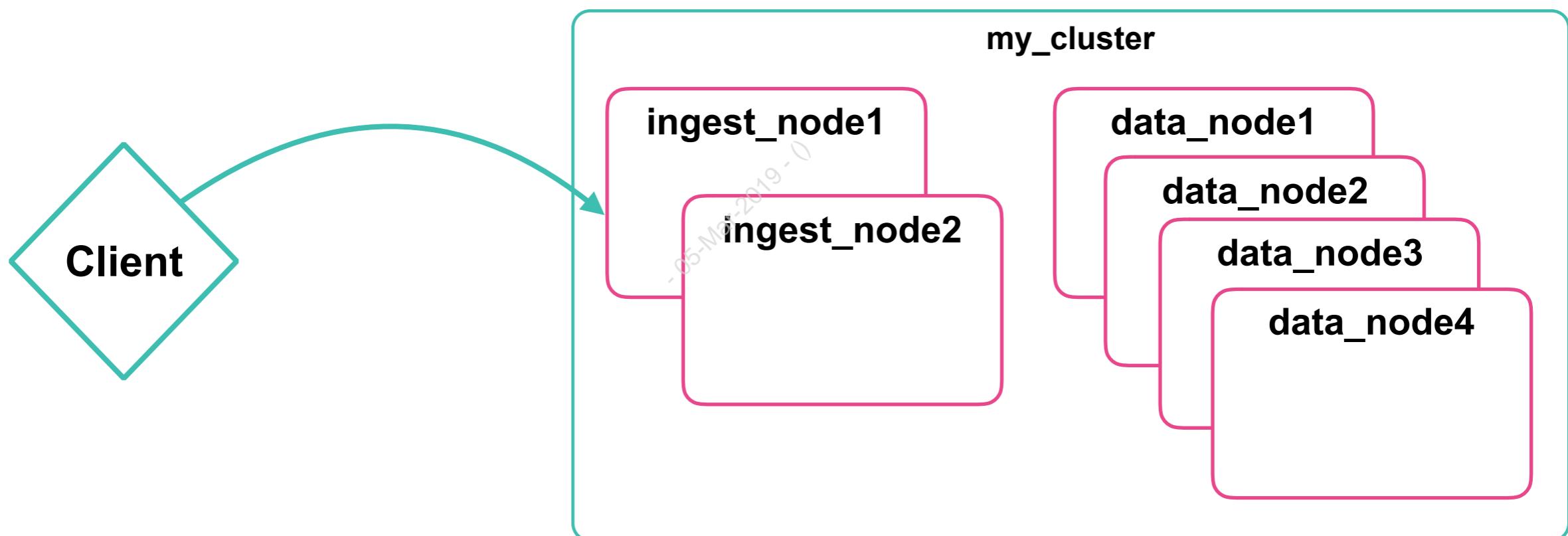
Ingest Node

- **Ingest nodes** provide the ability to pre-process a document right before it gets indexed
 - an ingest node intercepts an index or bulk API request,
 - applies transformations,
 - passes the documents back to the index or bulk API



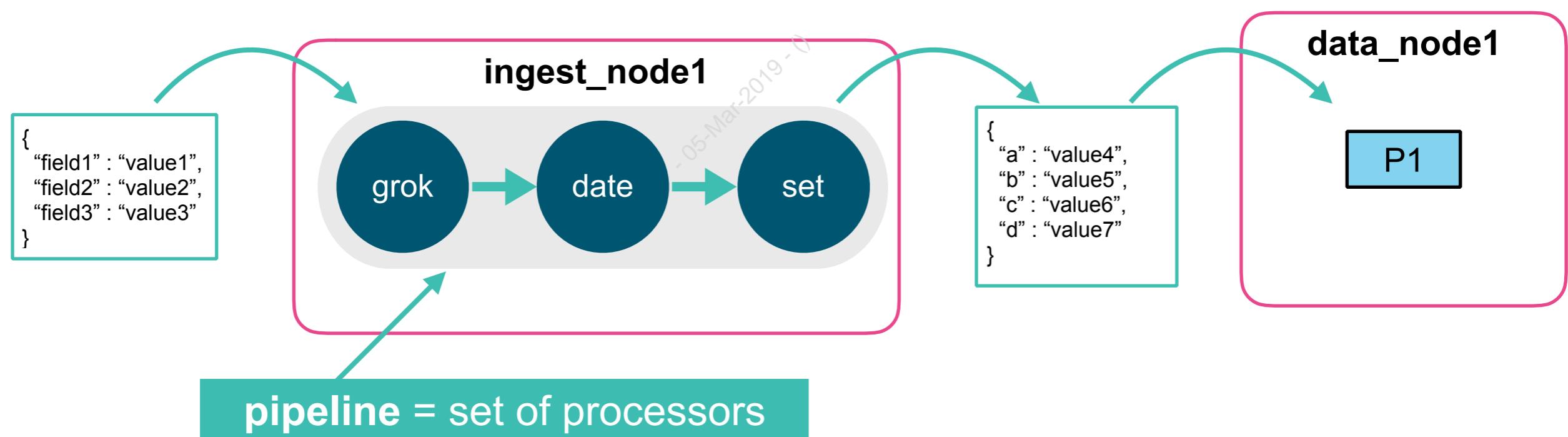
Dedicated Ingest Nodes

- All nodes are ingest nodes by default
 - set `node.ingest` to `false` to disable
 - larger clusters will likely have dedicated ingest nodes (as discussed later in the course)



What is a Pipeline?

- A *pipeline* is a set of processors
 - a processor is similar to a filter in Logstash
 - has read and write access to documents that pass through the pipeline



Defining Pipelines

- Pipelines are defined using a **PUT** with the *Ingest API*
 - stored in the cluster state

```
unique id for your pipeline  
PUT _ingest/pipeline/my-pipeline-id  
{  
  "description" : "DESCRIPTION",  
  "processors" : [  
    {  
      ...  
    }  
  ],  
  "on_failure" : [  
    {  
      ...  
    }  
  ]  
}
```

array of processors

optional array of processors
if an error occurs



Example of a Pipeline

- The following pipeline adds a field to a document using the “set” processor

```
PUT _ingest/pipeline/my_pipeline
{
  "processors": [
    {
      "set": {
        "field": "number_of_views",
        "value": 0
      }
    }
  ]
}
```

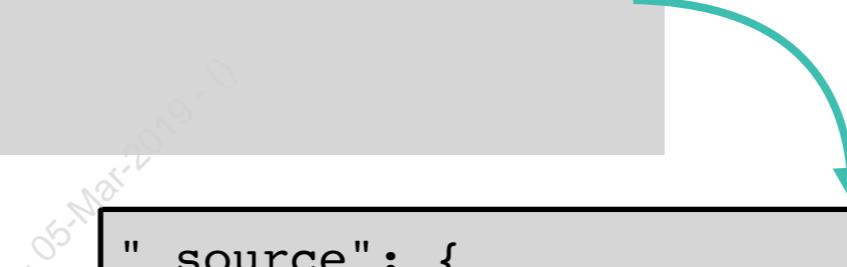
Adds “number_of_views” if it does not exist, or sets it to 0 if it already does



Testing a Pipeline

- Use the `_simulate` endpoint to test a pipeline

```
POST _ingest/pipeline/my_pipeline/_simulate
{
  "docs": [
    {
      "_source": {
        "author": "Shay Banon",
        "blog_title": "You know, for Search!"
      }
    }
  ]
}
```



```
"_source": {
  "number_of_views": 0,
  "blog_title": "You know, for Search!",
  "author": "Shay Banon"
}
```



The split Processor

- The built-in ***split processor*** is perfect for splitting the “locales” field into an array:

```
PUT _ingest/pipeline/blogs_pipeline
{
  "processors": [
    {
      "split": {
        "field": "locales",
        "separator": ","
      }
    }
  ]
}
```

Can be a regular expression as well



Adding a Useful Field

- Suppose we want to analyze the web traffic to our blogs based on the length of the blog
 - using the length of the “**content**” field
- We do not have this value in our **blogs** index
 - We could compute it at request time, but that is expensive
 - A better solution would be to compute the length of the blog once, at index time

~05-Mar-2019~05



The script Processor

- The **script processor** allows scripts to be executed within ingest pipelines:

```
PUT _ingest/pipeline/blogs_pipeline
{
  "processors": [
    {
      "split": {
        "field": "locales",
        "separator": ","
      }
    },
    {
      "script": {
        "source": """
if(ctx.containsKey("content")) {
  ctx.content_length = ctx.content.length();
} else {
  ctx.content_length = 0;
}
"""
      }
    }
  ]
}
```

“ctx” is a map, and we check to see if it contains a “content” field

Assigning “ctx.content_length” adds the field to the doc if it is not already defined



The script Processor

```
POST _ingest/pipeline/blogs_pipeline/_simulate
{
  "docs": [
    {
      "_source": {
        "locales": "de-de,fr-fr,ja-jp,ko-kr",
        "content": "This is a test."
      }
    },
    {
      "_source": {
        "locales": "en-en"
      }
    }
  ]
}
```



```
"locales": [
  "de-de",
  "fr-fr",
  "ja-jp",
  "ko-kr"
],
"content": "This is a test.",
"content_length": 15
...
"locales": [
  "en-en"
],
"content_length": 0
...
```



Fix the Blogs Index

- Let's run our new pipeline on all of existing indexed blogs:

```
POST blogs_fixed/_update_by_query?pipeline=blogs_pipeline
```

```
GET blogs_fixed/_search
```

05-Mar-2019-0

```
{  
  "locales": [  
    "de-de",  
    "fr-fr",  
    "ja-jp",  
    "ko-kr",  
    "zh-chs"  
  ],  
  "author": "Steve Dodson",  
  "category": "News",  
  "title": "Introducing Machine Learning for the Elastic Stack",  
  "publish_date": "2017-05-04T06:00:00.000Z",  
  "seo_title": "",  
  "content": "...",  
  "url": "/blog/introducing-machine-learning-for-the-elastic-stack",  
  "content_length": 5861  
}
```



Chapter Review

~05-Mar-2019~

Summary

- **Painless** is a scripting language designed specifically for use with Elasticsearch
- Painless scripts can be defined *inline* or *stored* in the cluster
- The first time Elasticsearch sees a new script, it compiles it and stores the compiled version in a **cache**
- You can copy documents from one index to another using the **Reindex API**
- The **Update By Query API** allows you to reindex a collection of documents into the same index
- **Ingest nodes** provide the ability to pre-process a document right before it gets indexed
- A **pipeline** is a set of processors that are executed by an ingest node when a document is indexed



Quiz

1. **True or False:** The Painless scripting language was developed just for Elasticsearch.
2. **True or False:** Adding a field to a mapping automatically adds that field to all the documents already indexed
3. In a `_reindex` request, what is the effect of setting “`version_type`” to “`external`”?
4. How many documents are updated in the following request?

```
POST messages/_update_by_query
```

5. What is the effect of the following pipeline?

```
"processors" : [  
  {  
    "script" : {  
      "source" : "ctx._index=ctx.clientip.country_iso_code.toLowerCase()"  
    }  
  }  
]
```



Lab 3

Fixing Data

~05-Mar-2019~0



Chapter 4

Advanced Search & Aggregations

-05-Mar-2019-

- 1 Elasticsearch Internals
- 2 Field Modeling
- 3 Fixing Data
- 4 Advanced Search & Aggregations
- 5 Cluster Management
- 6 Capacity Planning
- 7 Document Modeling
- 8 Monitoring and Alerting
- 9 From Dev to Production



Topics covered:

- Searching for Patterns
- Dealing with null Values
- Scripted Searches
- Search Templates
- Aggregations
- Pipeline Aggregations

~05-Mar-2019~0



Searching for Patterns

..05-Mar-2019..Q

Wildcard Query

- The **wildcard** query is a term-based search that contains a wildcard:
 - * = anything
 - ? = any single character
- **wildcard** can be expensive, especially if you have leading wildcards

```
GET blogs/_search
{
  "query": {
    "wildcard": {
      "title.keyword": {
        "value": "* 5.*"
      }
    }
  }
}
```

I'm looking for all blog posts about 5.x releases



Regexp Query

- The **regexp** query is a term-based search that offers even more power to match patterns
 - Like **wildcard** queries, **regexp** queries can be very expensive
 - Syntax is based on the Lucene regular expression engine

<https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-regexp-query.html#regexp-syntax>

```
GET blogs/_search
{
  "query": {
    "regexp": {
      "title.keyword": ".*5\\.[0-2]\\.[0-9].*"
    }
  }
}
```

I'm looking for all blog posts about 5.0, 5.1 or 5.2 releases

The pattern provided must match the entire string (always anchored)



Dealing with null Values

05-Mar-2019 - 0

Dealing with null Values

- Your documents will likely have missing or null fields
 - If you want to know if a field exists *and* is not null, use the **exists** query
- For example, the “**locales**” field in **blogs** is non-null if the blog is written in multiple languages

*“I am looking for **blogs** written in multiple languages.”*

```
GET blogs/_search
{
  "query": {
    "exists": {
      "field": "locales"
    }
  }
}
```

Finding Missing Fields

- The opposite of **exists** is finding fields that are missing
 - You can wrap **exists** in a **must_not** clause to find documents that are missing a field

```
GET logs_server*/_search
{
  "query": {
    "bool": {
      "must_not": {
        "exists": {
          "field": "geoip.region_name"
        }
      }
    }
  }
}
```

"I am looking for log events missing the region name."



Using exists on Objects

- The **exists** query works on objects as well
 - For example, **geoip** is an object in our logs indices

```
GET logs_server*/_search
{
  "query": {
    "bool": {
      "must_not": {
        "exists": {
          "field": "geoip"
        }
      }
    }
  }
}
```

*“Which log events
are missing the **geoip**
details?”*



Scripted Queries

· 05-Mar-2019 · 0

The Script Query

- Suppose we want **blogs** that have more than one “locale”

```
GET blogs_fixed/_search
{
  "query": {
    "bool": {
      "filter": {
        "script": {
          "script": {
            "source": "doc['locales'].size() > 1"
          }
        }
      }
    }
  }
}
```

Script queries must
compute a Boolean value

```
"hits": {
  "total": 54,
  "max_score": 0,
  "hits": [
    {
      "_source": {
        "locales": [
          "de-de",
          "fr-fr",
          "ja-jp",
          "ko-kr",
          "zh-chs"
        ],
        "author": "Steve Dodson",
        "category": "News",
        "title": "Introducing Machine
Learning for the Elastic Stack",
      }
    }
  ]
}
```



The Execute API

- It is possible to test a script before executing it in a query using the execute API
- The API supports multiple contexts depending on how the script is being executed

```
POST /_scripts/painless/_execute
{
  "script": {
    "source": "doc['locales'].size() > 1"
  },
  "context": "filter",
  "context_setup": {
    "index": "blogs_fixed",
    "document": {
      "locales": ["fr-fr", "de-de"]
    }
  }
}
```

Script to test

Context in which the script is applied

Specify the document on which the script will be applied

```
{
  "result": true
}
```



Scripted Fields

~05-Mar-2019 ~0

Script Fields

- You can **add fields to a query response** that are generated in a script
 - Use **script_fields** to return a script evaluation for each hit:

```
GET blogs_fixed/_search
{
  "script_fields": {
    "day_of_week": {
      "script": {
        "source": """
def d = new Date(doc['publish_date'].value.millis);
return d.toString().substring(0,3);
"""
      }
    }
  }
}
```

The name of the field being added to the response

In Painless, use **def** to declare variables

The value returned from the script



Script Fields

- The response from the previous query includes the day of the week
 - but the `_source` did not get returned?

```
"hits": [  
  {  
    "_index": "blogs_fixed",  
    "_type": "doc",  
    "_id": "Cc1CKmIBCLh5xF6i7Y2b",  
    "_score": 1,  
    "fields": {  
      "day_of_week": [  
        "Fri"  
      ]  
    }  
  },  
  {  
    "_index": "blogs_fixed",  
    "_type": "doc",  
    "_id": "g81CKmIBCLh5xF6i7Y-v",  
    "_score": 1,  
    "fields": {  
      "day_of_week": [  
        "Tue"  
      ]  
    }  
  },  
  {  
    "_index": "blogs_fixed",  
    "_type": "doc",  
    "_id": "g81CKmIBCLh5xF6i7Y-w",  
    "_score": 1,  
    "fields": {  
      "day_of_week": [  
        "Wed"  
      ]  
    }  
  }]
```



Adding `_source` to Script Fields

- If you add a `script_fields` section, the query no longer returns the `_source` by default
 - but it is easy to add `_source` to the response

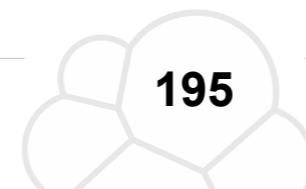
```
GET blogs_fixed/_search
{
  "_source": [], ←
  "script_fields": {
    "day_of_week": {
      "script": {
        "source": """
def d = new Date(doc['publish_date'].value.millis);
return d.toString().substring(0,3);
"""
      }
    }
  }
}
```

Now you will get the `_source` plus the define “`script_fields`”



Performance Concerns

- It is important to *understand the cost* of using a script in some use cases
 - For example, the script in a **script_fields** clause needs to be executed on each hit
- It may be much more efficient to calculate these needed values once, at index time
 - perhaps using an ingest pipeline
 - or modeling your data in a way where scripting can be avoided



Search Templates

„05-Mar-2019“

Use Case for Search Templates

- Our users write some very lengthy and complex queries
 - In many cases, the same query is executed over and over, but with a few different values
- **Search templates** allow you to define a query with **parameters** that can be defined at execution time
- Benefits include:
 - avoid repeating code in multiple places
 - minimize mistakes
 - easier to test and execute your queries
 - share queries between applications
 - allow users to only execute a few predefined queries



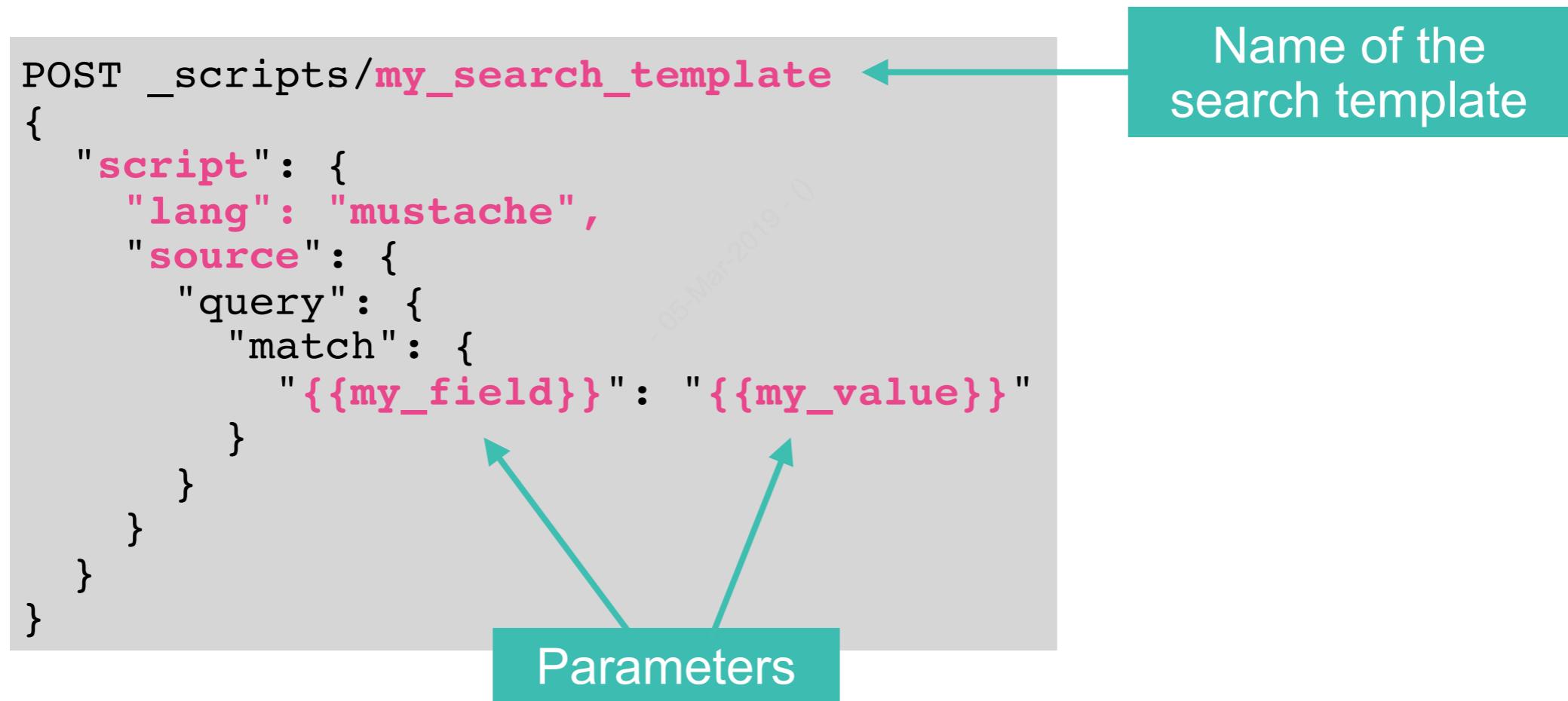
Defining a Template

- Templates are stored in the cluster state using the `_scripts` endpoint
 - the language for search templates is “mustache”

```
POST _scripts/my_search_template
{
  "script": {
    "lang": "mustache",
    "source": {
      "query": {
        "match": {
          "{{my_field}}": "{{my_value}}"
        }
      }
    }
  }
}
```

← Name of the search template

↑ Parameters



Using a Stored Template

- Use the `_search/template` endpoint to execute a stored template
 - passing in the necessary parameter values

*"I am looking for
blogs that have **shard** in
the **title**."*

```
GET blogs/_search/template
{
  "id": "my_search_template",
  "params": {
    "my_field": "title",
    "my_value": "shard"
  }
}
```



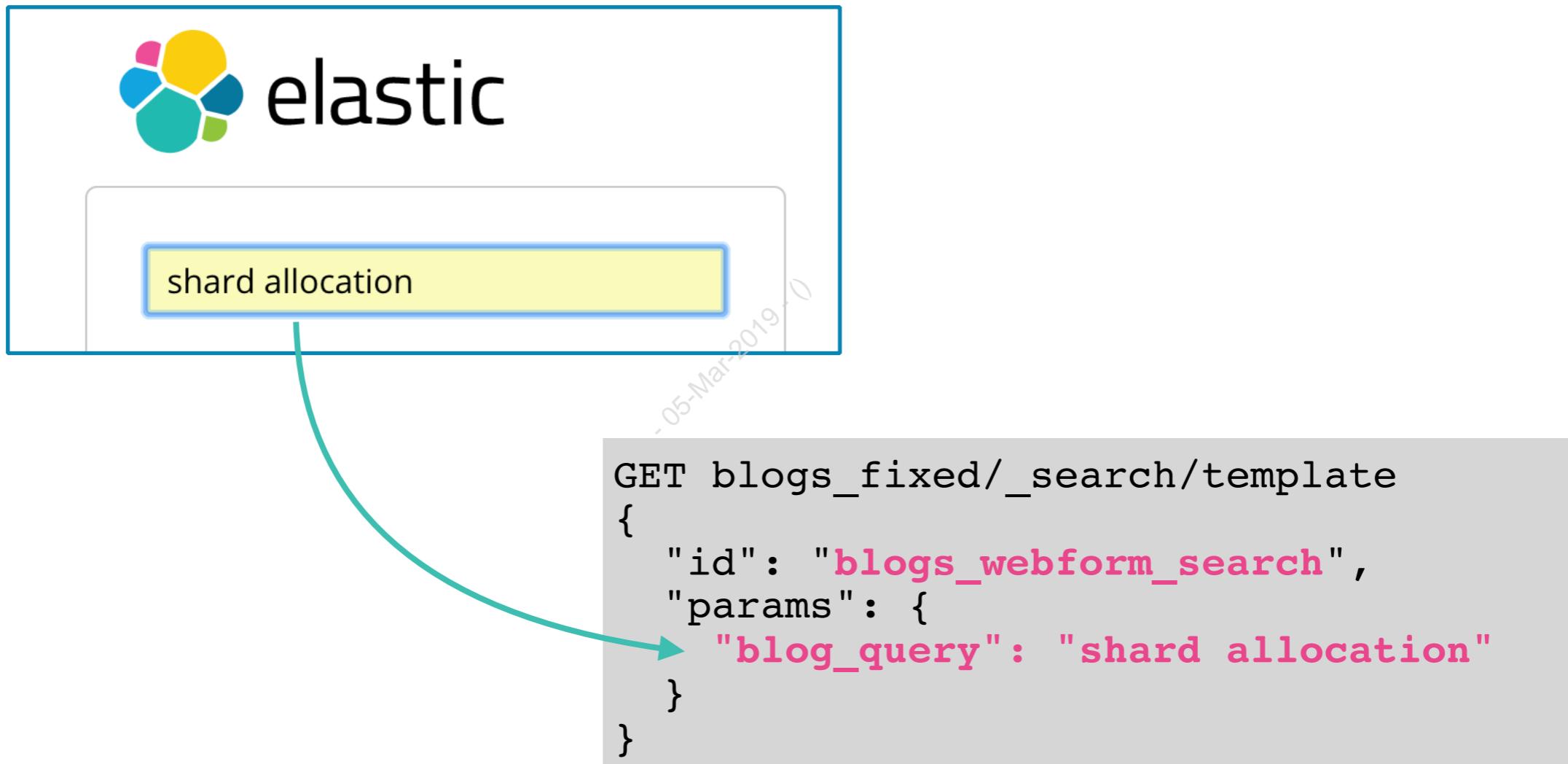
Template for the Blogs Web App

```
POST _scripts/blogs_webform_search
{
  "script": {
    "lang": "mustache",
    "source": {
      "query": {
        "bool": {
          "must": {
            "multi_match": {
              "query": "{{blog_query}}",
              "fields": ["title","title.*","content","content.*"],
              "operator": "and",
              "type": "most_fields"
            }
          },
          "should": {
            "multi_match": {
              "query": "{{blog_query}}",
              "fields": ["title","title.*","content","content.*"],
              "type": "phrase"
            }
          }
        }
      }
    }
  }
}
```



Using the Blogs Templates

- When a user searches for something on our website, we can simply execute the search template with the input:



Conditionals

- Mustache does not have ***if/else*** logic
 - But you can define a ***section*** that gets skipped if the parameter is false or not defined

```
{{#param1}}
  "This section is skipped if param1 is null or false"
{{/param1}}
```

~05-Mar-2019~ Ø



Example with a Conditional

```
POST _scripts/blogs_with_date_search
{
  "script": {
    "lang": "mustache",
    "source": """
      {
        "query": {
          "bool": {
            "must": [
              {"match": {"content": "{{search_term}}"}},
              "{{#search_date}}
                ,
                "filter": {
                  "range": {
                    "publish_date": {"gte": "{{search_date}}"}
                  }
                }
              {{/search_date}}
            ]
          }
        }
      }
    """
  }
}
```

Add a filter clause if the
“`search_date`” parameter
is defined



Test the Conditional Script

```
GET blogs_fixed/_search/template
{
  "id": "blogs_with_date_search",
  "params": {
    "search_term": "shay banon"
  }
}
```

47 hits

```
GET blogs_fixed/_search/template
{
  "id": "blogs_with_date_search",
  "params": {
    "search_term": "shay banon",
    "search_date": "2017-07-01"
  }
}
```

5 hits



Aggregations

~05-Mar-2019~Q

The Percentiles Aggregation

- The ***percentiles*** metrics aggregation calculates percentiles over a numeric field
 - percentiles show the point at which a certain percentage of observed values occur

```
GET logs_server*/_search
{
  "size": 0,
  "aggs": {
    "runtime_percentiles": {
      "percentiles": {
        "field": "runtime_ms"
      }
    }
  }
}
```

05-Mar-2019

```
"runtime_percentiles": {
  "values": {
    "1.0": 0,
    "5.0": 88.00109639047503,
    "25.0": 95.00000000000001,
    "50.0": 103.37306961911929,
    "75.0": 159.88916204500126,
    "95.0": 685.1015874756147,
    "99.0": 4198.930939937213
  }
}
```

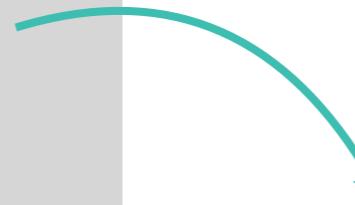
95% of logs have a runtime less than 685.1 ms



Percentiles Example

- You can specify the percentiles to be computed
 - For example, the following agg computes the quintiles (20% intervals) for the **runtime** of the log events

```
GET logs_server*/_search
{
  "size" : 0,
  "aggs": {
    "runtime_quintiles": {
      "percentiles": {
        "field": "runtime_ms",
        "percents": [
          20,
          40,
          60,
          80,
          100
        ]
      }
    }
  }
}
```



```
"runtime_quintiles": {
  "values": {
    "20.0": 93.83826098733662,
    "40.0": 99,
    "60.0": 111.35144881124744,
    "80.0": 236.679414908546,
    "100.0": 59756
  }
}
```

80% of the responses have of runtime less than 236 ms



Percentiles Rank

- Instead of providing percents and getting back values, you can pass in a value and get back its percentile
 - percentiles_rank** is essentially the opposite logic of **percentile**

```
GET logs_server*/_search
{
  "size": 0,
  "aggs": {
    "runtime_ranks": {
      "percentile_ranks": {
        "field": "runtime_ms",
        "values": [100, 500]
      }
    }
  }
}
```

You can pass in multiple values

```
"runtime_ranks": {
  "values": {
    "100.0": 43.74582092577736,
    "500.0": 94.35230182775403
  }
}
```

43.7% of responses occurred in less than 100 ms



Motivation for top_hits

- Suppose we do a search for blogs about Logstash filters, and we want to bucket them by author:

```
GET blogs/_search
{
  "size": 0,
  "query": {
    "match": {
      "content": "logstash filters"
    }
  },
  "aggs": {
    "blogs_by_author": {
      "terms": {
        "field": "author.keyword"
      }
    }
  }
}
```

Suyog wrote 69 blogs about Logstash filters, but which blogs are the most relevant?

```
"buckets": [
  {
    "key": "Suyog Rao",
    "doc_count": 69
  },
  {
    "key": "Alexander Reelsen",
    "doc_count": 67
  },
  {
    "key": "Megan Wieling",
    "doc_count": 31
  },
  {
    "key": "Leslie Hawthorn",
    "doc_count": 29
  },
]
```



Example of top_hits

```
GET blogs/_search
{
  "size": 0,
  "query": {
    "match": {
      "content": "logstash filters"
    }
  },
  "aggs": {
    "blogs_by_author": {
      "terms": {
        "field": "author.keyword"
      },
      "aggs": {
        "logstash_top_hits": {
          "top_hits": {
            "size": 5
          }
        }
      }
    }
  }
}
```

Returns the top 5 blogs from each author (based on the `_score` from the “`match`” query)



The output of top_hits:

- Notice the top 5 hits from each bucket are returned in the “aggregations” clause of the response

```
"buckets": [
  {
    "key": "Suyog Rao",
    "doc_count": 69,
    "logstash_top_hits": {
      "hits": {
        "total": 69,
        "max_score": 6.6510196,
        "hits": [
          {
            "_index": "blogs",
            "_type": "doc",
            "_id": "TM1CKmIBCLh5xF6i7Y2b",
            "_score": 6.6510196,
            "_source": {
              "publish_date": "2016-06-27T06:00:00.000Z",
              "seo_title": "",
              "category": "The Logstash Lines",
              "locales": "",
              "title": "Logstash Lines: More Monitoring
Info, Beats Input Improvements",
            }
          }
        ]
      }
    }
  }
]
```



The missing Aggregation

```
GET logs_server*/_search
{
  "size": 0,
  "aggs": {
    "missing_latitude": {
      "missing": {
        "field": "geoip.location.lat"
      }
    },
    "missing_longitude": {
      "missing": {
        "field": "geoip.location.lon"
      }
    }
  }
}
```

*“How many log events are **missing** the IP location?”*

```
"aggregations": {
  "missing_latitude": {
    "doc_count": 6397
  },
  "missing_longitude": {
    "doc_count": 6397
  }
}
```



Scripted Aggregations

- Typically, the values used in an aggregation are from a **field** within the document:

A simple **terms** agg on a field in the document

```
GET blogs/_search
{
  "size": 0,
  "aggs": {
    "author_terms": {
      "terms": {
        "field": "author.keyword"
      }
    }
  }
}
```

Scripted Aggregations

- It is also possible to define a *script* which will generate the values used in an aggregation
 - The script generates a value per document

```
GET blogs/_search
{
  "size": 0,
  "aggs": {
    "blogs_by_day_of_week": {
      "terms": {
        "script": {
          "source": "doc['publish_date'].value.dayOfWeek"
        }
      }
    }
  }
}
```

A **terms** agg that buckets by the day of the week

```
"buckets": [
  {
    "key": "2",
    "doc_count": 415
  },
  {
    "key": "1",
    "doc_count": 385
  },
]
```

1 = Monday,
2 = Tuesday,
etc.



Scripted Aggregations

- Another common use case for scripts in aggs is to combine multiple fields for a **terms** agg
 - We can view the days of the week that authors publish blogs:

```
GET blogs/_search
{
  "size": 0,
  "aggs": {
    "blogs_by_author_and_day_of_week": {
      "terms": {
        "script": {
          "source": "doc['author.keyword'].value + '_'+doc['publish_date'].value.dayOfWeek"
        }
      }
    }
  }
}
```

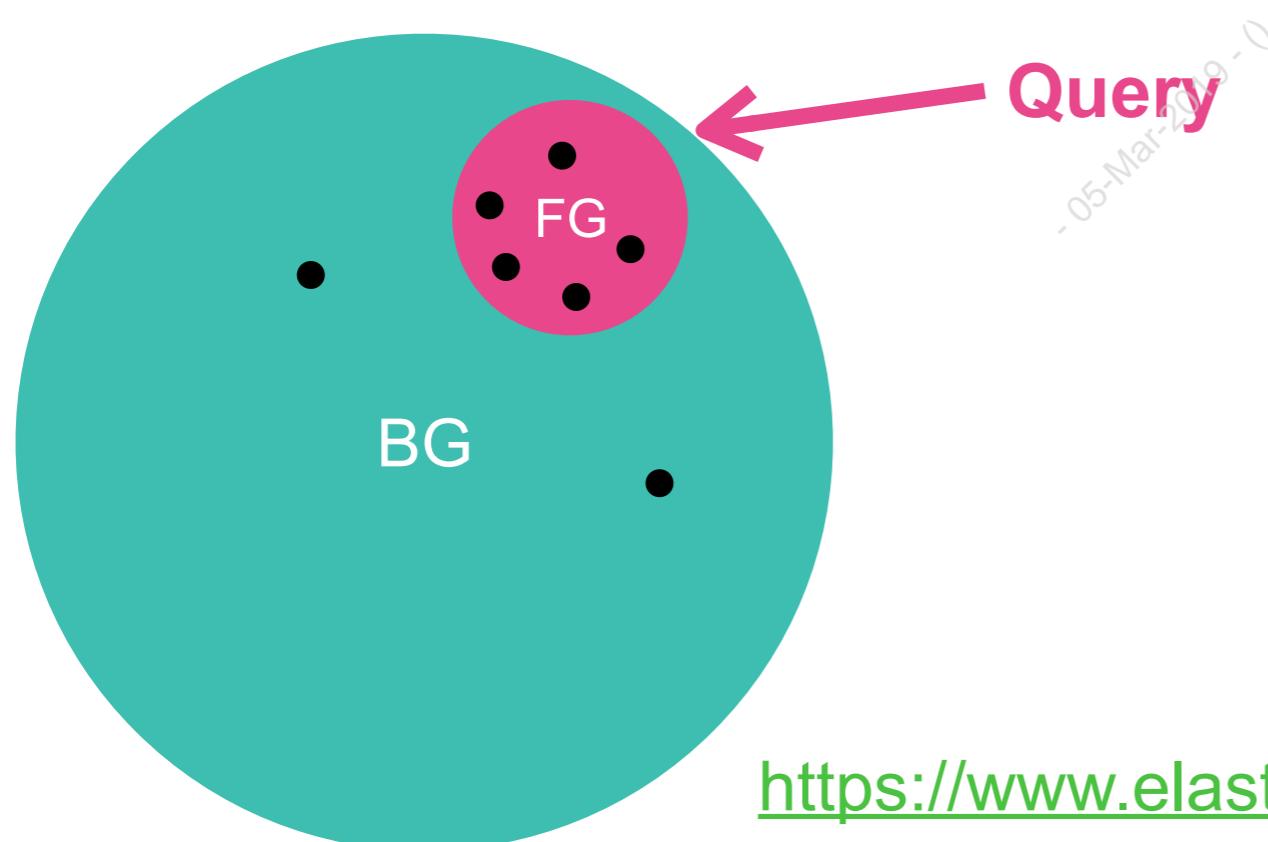


```
"buckets": [
  {
    "key": "Alexander Reelsen_3",
    "doc_count": 67
  },
  {
    "key": "Livia Froelicher_1",
    "doc_count": 52
  },
  {
    "key": "Clinton Gormley_1",
    "doc_count": 42
  },
  {
    "key": "Clinton Gormley_2",
    "doc_count": 38
  },
]
```



The Significant Terms Aggregation

- Terms Aggregation + Noise Filter
 - Discards **commonly common** terms that **terms agg** would return
- Low frequency **terms** in the background data pop out as high frequency **terms** in the foreground data
 - Finds **uncommonly common** terms in your dataset



Some Use Cases:

- Recommendation
- Fraud Detection
- Defect Detection

And more...

<https://www.elastic.co/blog/significant-terms-aggregation>



Let's Start with a Terms Agg

```
GET blogs/_search
{
  "size": 0,
  "aggs": {
    "author_buckets": {
      "terms": {
        "field": "author.keyword",
        "size": 10
      },
      "aggs": {
        "content_terms": {
          "terms": {
            "field": "content",
            "size": 10
          }
        }
      }
    }
  }
}
```

*Let's look for a relationship between **authors** and **content**.*



The Results of the Terms Agg

- These would be considered “***commonly common***” terms that our authors use in their blogs:

```
{  
  "key": "Monica Sarbu",  
  "doc_count": 89,  
  "content_terms": {  
    "doc_count_error_upper_bound": 66,  
    "sum_other_doc_count": 15096,  
    "buckets": [  
      {  
        "key": "and",  
        "doc_count": 89  
      },  
      {  
        "key": "the",  
        "doc_count": 89  
      },  
      {  
        "key": "to",  
        "doc_count": 88  
      },  
      {  
        "key": "in",  
        "doc_count": 86  
      },  
      {  
        "key": "is",  
        "doc_count": 86  
      },  
      {  
        "key": "this",  
        "doc_count": 86  
      }  
    ]  
  }  
}
```

Monica likes to blog about “**and**”,
“**the**”, “**to**”, “**in**” and so on.



Using significant_terms

- Now let's try the same agg with **significant_terms**:

```
GET blogs/_search
{
  "size": 0,
  "aggs": {
    "author_buckets": {
      "terms": {
        "field": "author.keyword",
        "size": 10
      },
      "aggs": {
        "content_significant_terms": {
          "significant_terms": {
            "field": "content",
            "size": 10
          }
        }
      }
    }
  }
}
```

*Let's look for the
“uncommonly common”
relationship between
author and **content**.*



The Output of significant_terms

```
"key": "Monica Sarbu",
"doc_count": 89,
"content_significant_terms": {
  "buckets": [
    {
      "key": "metricbeat",
      "doc_count": 66,
      "score": 8.295430260419582,
      "bg_count": 97
    },
    {
      "key": "filebeat",
      "doc_count": 66,
      "score": 5.849324105618168,
      "bg_count": 133
    },
    {
      "key": "beat",
      "doc_count": 56,
      "score": 5.3810714135420605,
      "bg_count": 105
    },
    {
      "key": "beats",
      "doc_count": 84,
      "score": 4.668550553314773,
      "bg_count": 253
    },
    ...
  ]
}
```

It appears Monica is an expert on Beats!



Pipeline Aggregations

~05-Mar-2018

Use Case for Pipeline Aggregations

- The following aggregations calculate the monthly response size of the logs dataset:
 - but suppose we want the **cumulative** sum as well...

```
GET logs_server*/_search
{
  "size": 0,
  "aggs": {
    "logs_by_month": {
      "date_histogram": {
        "field": "@timestamp",
        "interval": "month"
      },
      "aggs": {
        "monthly_sum_response": {
          "sum": {
            "field": "response_size"
          }
        }
      }
    }
  }
}
```

This gives us the sum from each month, but suppose we want the cumulative sum



Pipeline Aggregations

- A *pipeline aggregation* performs computations on the results of other aggregations
 - The input of a pipeline aggregation is typically the output of another aggregation
 - Use the **buckets_path** parameter to reference the values in the other aggregations
- Pipeline aggregations are executed on the coordinating node, *after* the results of the input agg are collected



Example of a Pipeline Aggregation

```
GET logs_server*/_search
{
  "size": 0,
  "aggs": {
    "logs_by_month": {
      "date_histogram": {
        "field": "@timestamp",
        "interval": "month"
      },
      "aggs": {
        "monthly_sum_response": {
          "sum": {
            "field": "response_size"
          }
        },
        "cumulative_sum_response": {
          "cumulative_sum": {
            "buckets_path": "monthly_sum_response"
          }
        }
      }
    }
  }
}
```

The input of “**cumulative_sum**” is the result of “**monthly_sum_response**”

“**cumulative_sum**” is a pipeline agg



Example of a Pipeline Aggregation

- The output from the previous search looks like:

```
"buckets": [
  {
    "key_as_string": "2017-03-01T00:00:00.000Z",
    "key": 1488326400000,
    "doc_count": 255,
    "monthly_sum_response": {
      "value": 15860968
    },
    "cumulative_sum_response": {
      "value": 15860968
    }
  },
  {
    "key_as_string": "2017-04-01T00:00:00.000Z",
    "key": 1491004800000,
    "doc_count": 467961,
    "monthly_sum_response": {
      "value": 25446117219
    },
    "cumulative_sum_response": {
      "value": 25461978187
    }
  },
  ...
]
```

monthly sum

cumulative sum

monthly sum

cumulative sum



Specifying buckets_path

- When referring to an aggregation at the same level, **buckets_path** can simply be the name of the agg:

```
GET logs_server*/_search
{
  "size": 0,
  "aggs": {
    "logs_by_month": {
      "date_histogram": {
        "field": "@timestamp",
        "interval": "month"
      },
      "aggs": {
        "monthly_sum_response": {
          "sum": {
            "field": "response_size"
          }
        },
        "monthly_max_response": {
          "max": {
            "field": "response_size"
          }
        },
        "cumulative_sum_response": {
          "cumulative_sum": {
            "buckets_path": "monthly_sum_response"
          }
        }
      }
    }
  }
}
```

In this example, the input agg and cumulative agg are at the same level



Aggregation Separator ‘>’

- If a pipeline aggregation is at a *parent level*, use the > symbol to reference the desired sub-level aggregation being used for the input:

```
GET logs_server*/_search
{
  "size": 0,
  "aggs": {
    "logs_by_month": {
      "date_histogram": {
        "field": "@timestamp",
        "interval": "month"
      },
      "aggs": {
        "monthly_sum_response": {
          "sum": {
            "field": "response_size"
          }
        }
      }
    },
    "max_monthly_sum": {
      "max_bucket": {
        "buckets_path": "logs_by_month>monthly_sum_response"
      }
    }
  }
}
```

“**max_monthly_sum**” is a single output over all buckets and the max value of “**monthly_sum_response**”

Use a ‘>’ symbol as a separator between aggregations



Examples of Pipeline Aggs

- There are about a dozen pipeline aggs available, including:
 - **avg_bucket**: calculates the average of a specified metric
 - **sum_bucket**: calculates the sum of a specified metric
 - **min_bucket** and **max_bucket**: for finding the min or max
 - **moving_avg**: calculates a moving average over a specified window
 - **bucket_script**: executes a script
- View the documentation for a complete list of pipeline aggregations:
 - <https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations-pipeline.html>



Chapter Review

~05-Mar-2019~

Summary

- Use **script_fields** to *add fields to a query response* that are generated in a script
- **Search templates** allow you to define a query with parameters that can be defined at execution time
- The **percentiles** aggregation calculates percentiles over a numeric field
- The **top_hits aggregation** keeps track of the most relevant documents being aggregated over
- A **pipeline aggregation** performs computations on the results of other aggregations



Quiz

1. **True or False:** Wildcards are powerful and should be used often.
2. How would you check how many documents do *not* have a specific field?
3. **True or False:** Scripts can be used to query, to aggregate and to return new calculated values.
4. Why would you use search templates?
5. In pipeline aggregations, when do you use the '>' symbol?
6. **True or False:** The input of a pipeline aggregation is the output of another aggregation.
7. In our **logs_server*** indices, how could you verify that 95% of web requests are executed in less than 100ms?



Lab 4

05-Mar-2019 10

Advanced Search & Aggregations

Chapter 5

Cluster Management

05-Mar-2019 - 0

- 1 Elasticsearch Internals
- 2 Field Modeling
- 3 Fixing Data
- 4 Advanced Search & Aggregations
- 5 Cluster Management
- 6 Capacity Planning
- 7 Document Modeling
- 8 Monitoring and Alerting
- 9 From Dev to Production



Topics covered:

- Elasticsearch Architecture Recap
- Dedicated Nodes
- Hot/Warm Architecture
- Shard Filtering
- Shard Filtering for Hardware
- Shard Allocation Awareness
- Forced Awareness

.05-Mar-2019-0



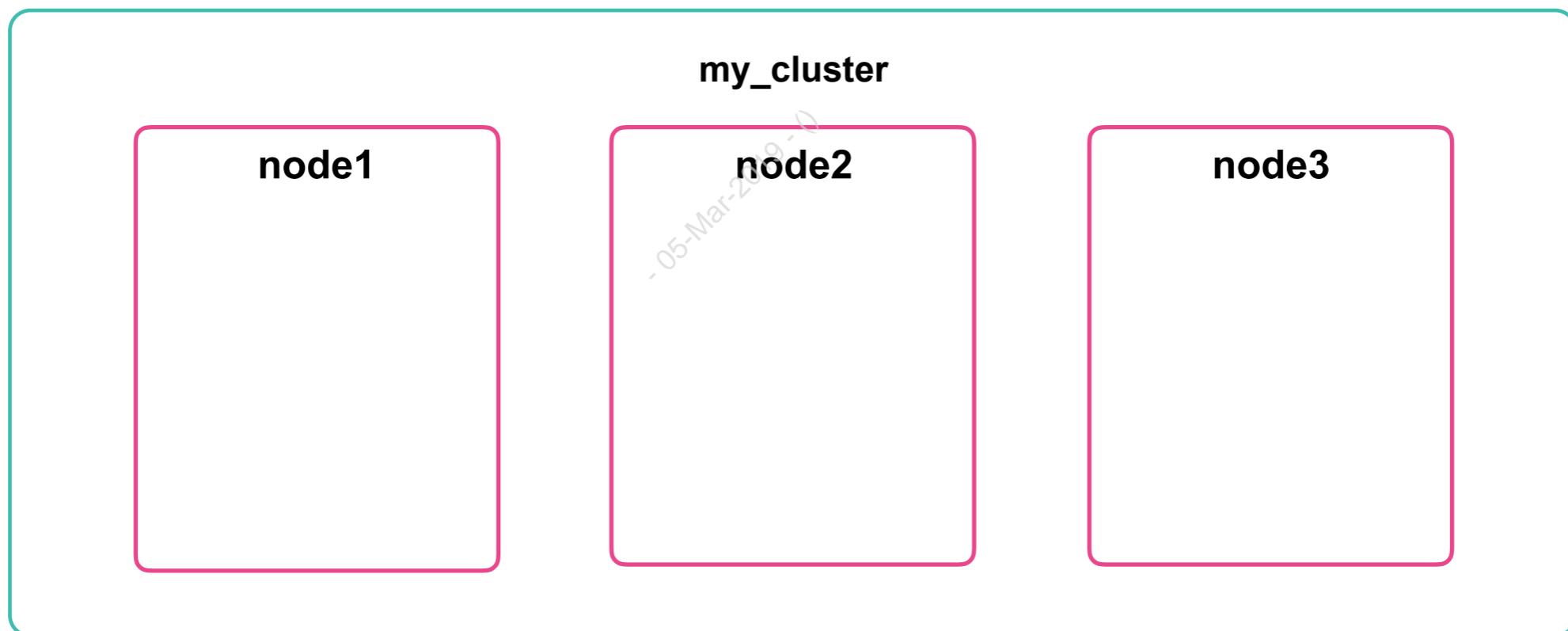
Elasticsearch Architecture Recap

-05-Mar-2019-0



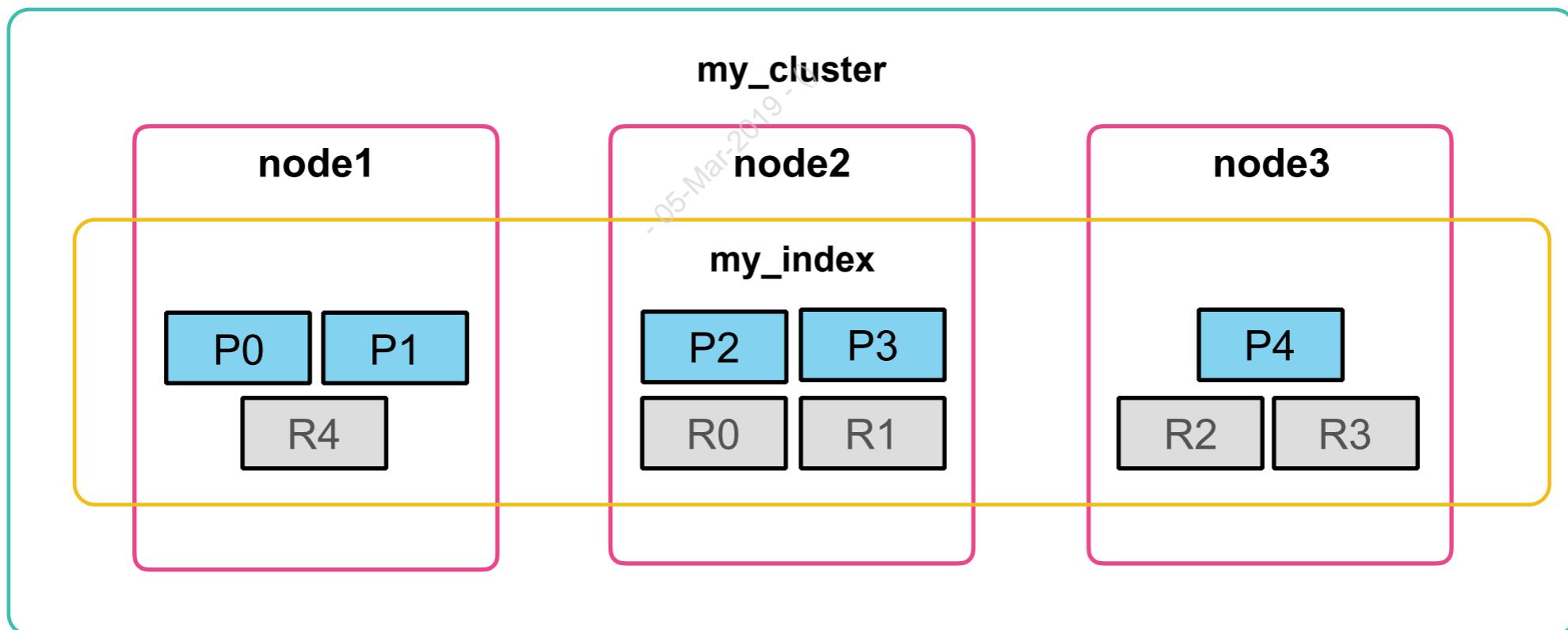
An Elasticsearch Cluster

- The largest unit of scale in Elasticsearch is a **cluster**
- A cluster is made up out of one or more **nodes**
- Each node is a running Elasticsearch process and typically there is a 1x1 relationship between a **server** and a node



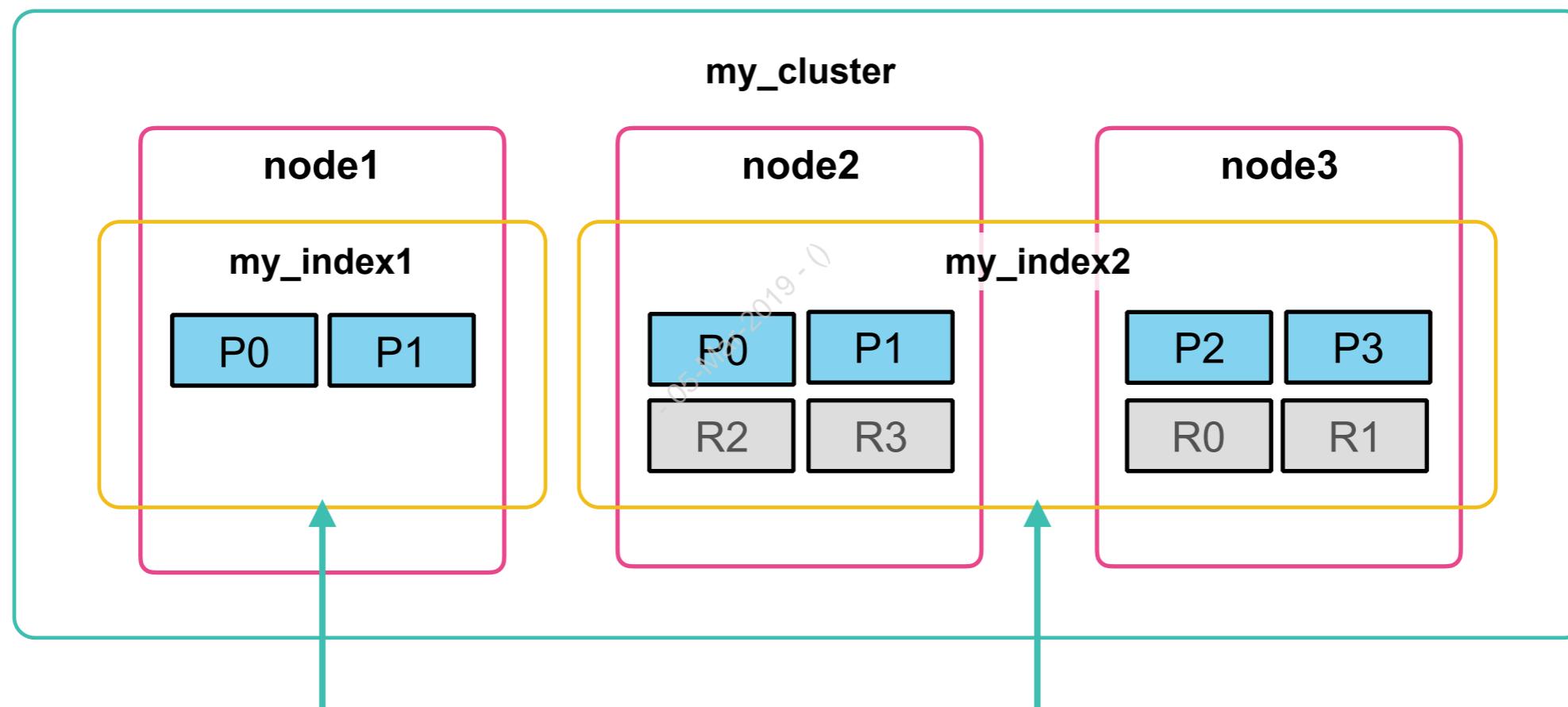
Indexes and Shards

- In your cluster, you create one or more *indices*
- Each index is sharded and its *shards* are distributed over the nodes
- Elasticsearch automatically distributes the shards evenly across the nodes



More Control

- Maybe you want more control over
 - your servers specification
 - or where the shards of your indices are allocated to



The shards of **my_index1** live on **node1**

The shards of **my_index2** live on **node2** and **node3**



Dedicated Nodes

~05-Mar-2019

Node Roles

- There are several roles a node can have:
 - Master eligible
 - Data
 - Ingest
 - Machine Learning
 - Coordinating
- Nodes can be ***dedicated*** nodes that only take on a single role...

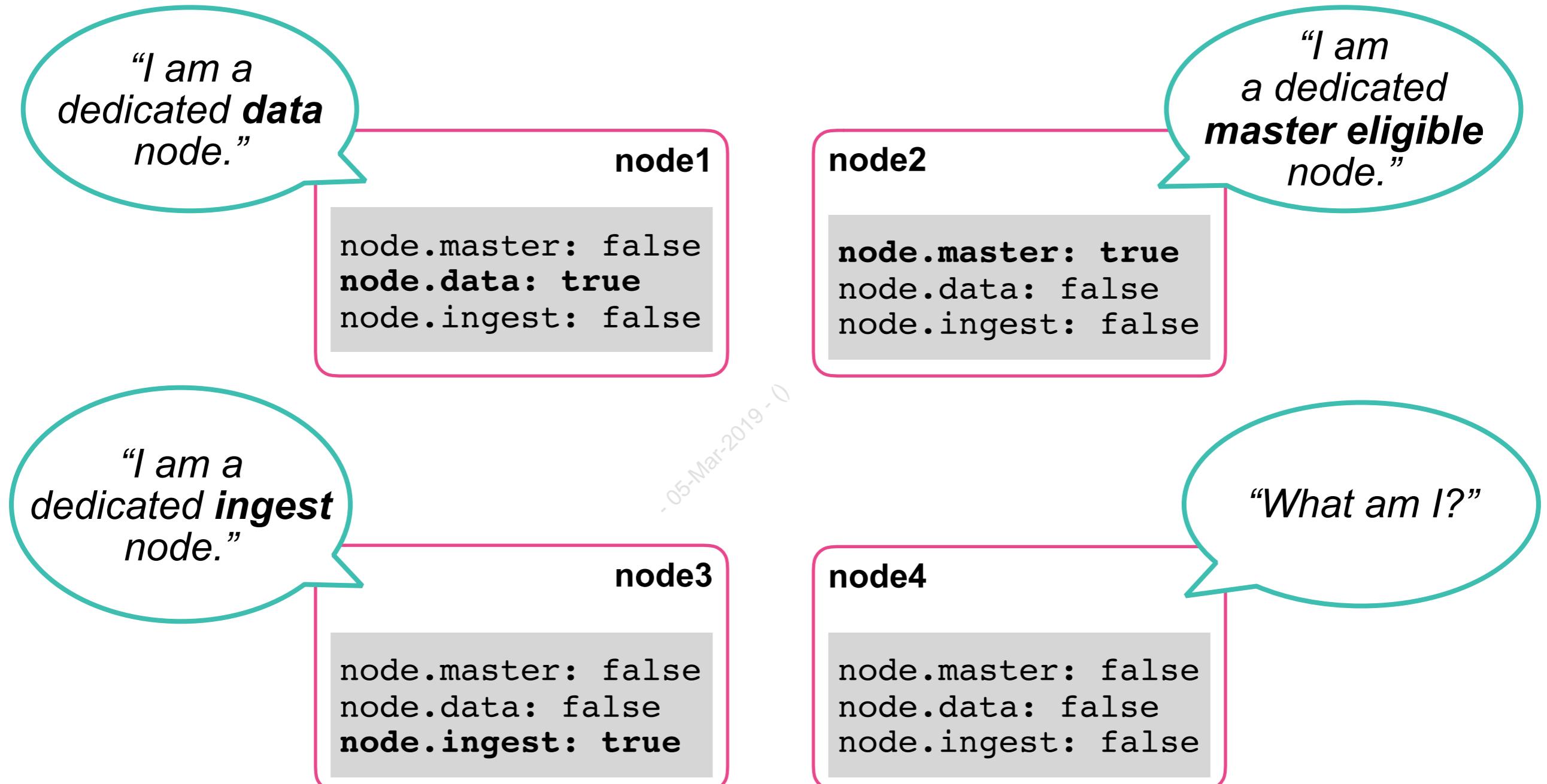
Configuring Node Roles

- By default, a node is a master-eligible, data, and ingest node:

Node type	Configuration parameter	Default value
master eligible	node.master	true
data	node.data	true
ingest	node.ingest	true

Dedicated Nodes

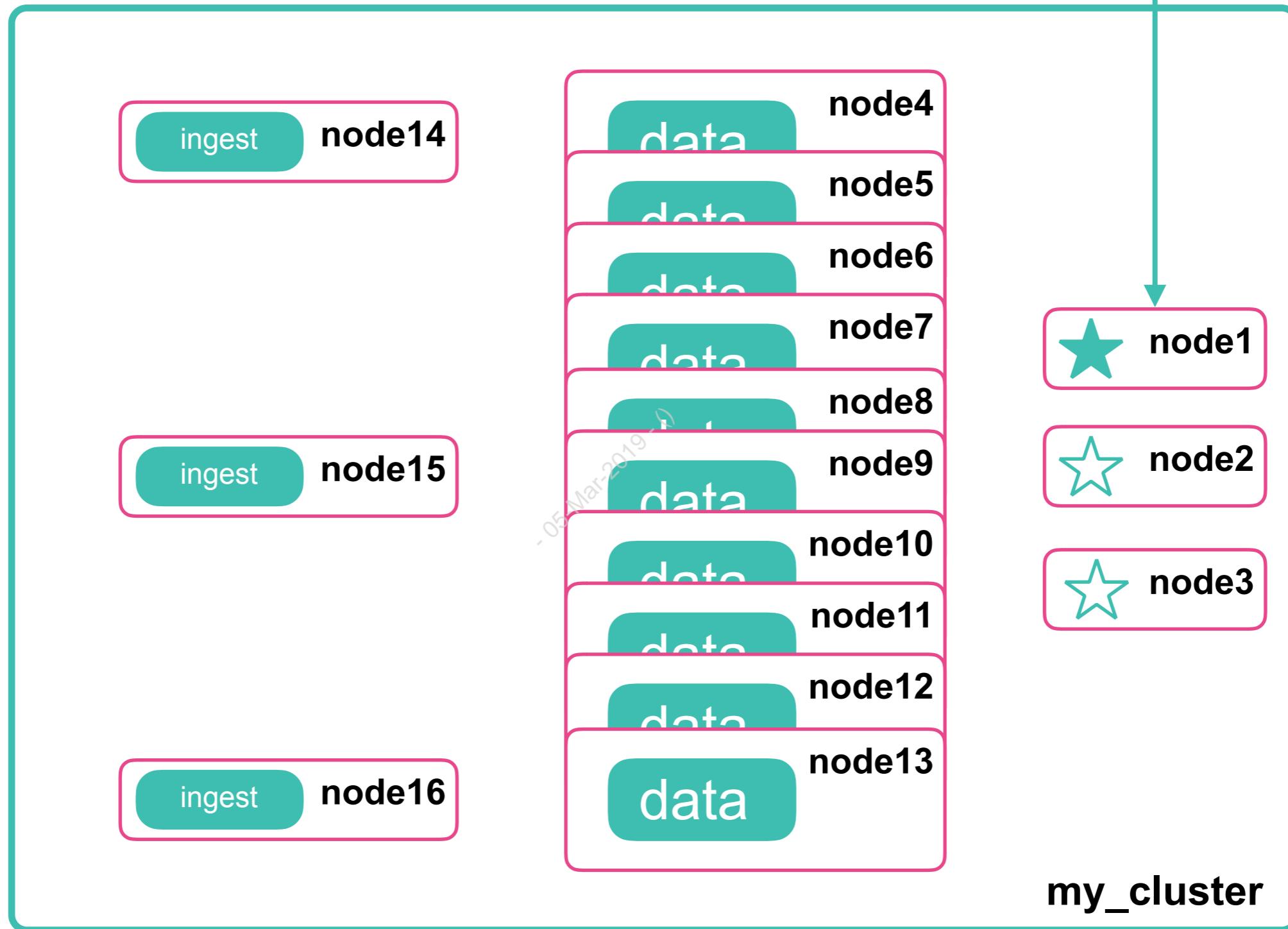
- You can configure a node to have a single role.



Dedicated Nodes Example

- Add data nodes to scale the cluster

minimum_master_nodes: 2



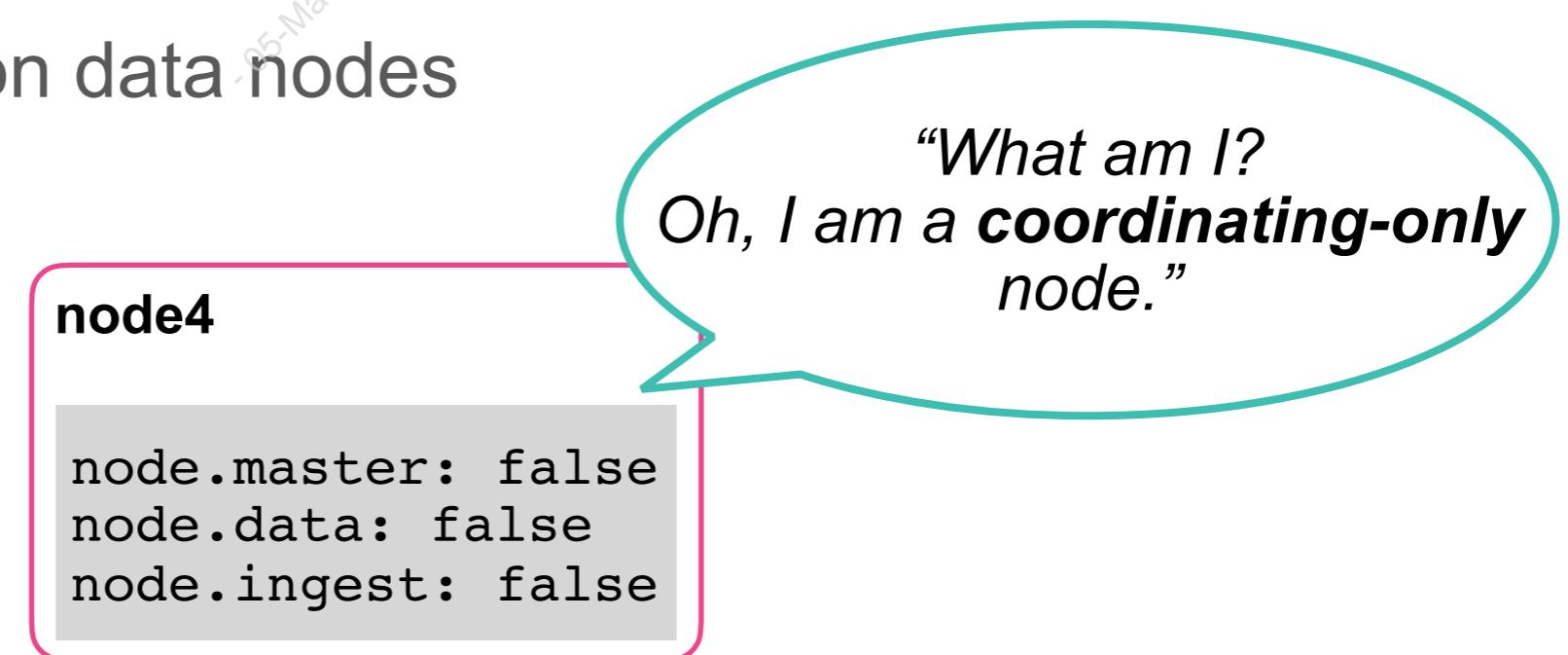
Why Use Dedicated Nodes?

- Machines can be selected for specific purposes
- ***Dedicated master eligible nodes***
 - can focus on the cluster state
 - machines with low CPU, RAM, and disk resources
- ***Dedicated data nodes***
 - can focus on data storage and processing client requests
 - machines with high CPU, RAM, and disk resources
- ***Dedicated ingest nodes***
 - can focus on data processing
 - machines with low disk, medium RAM, and high CPU resources

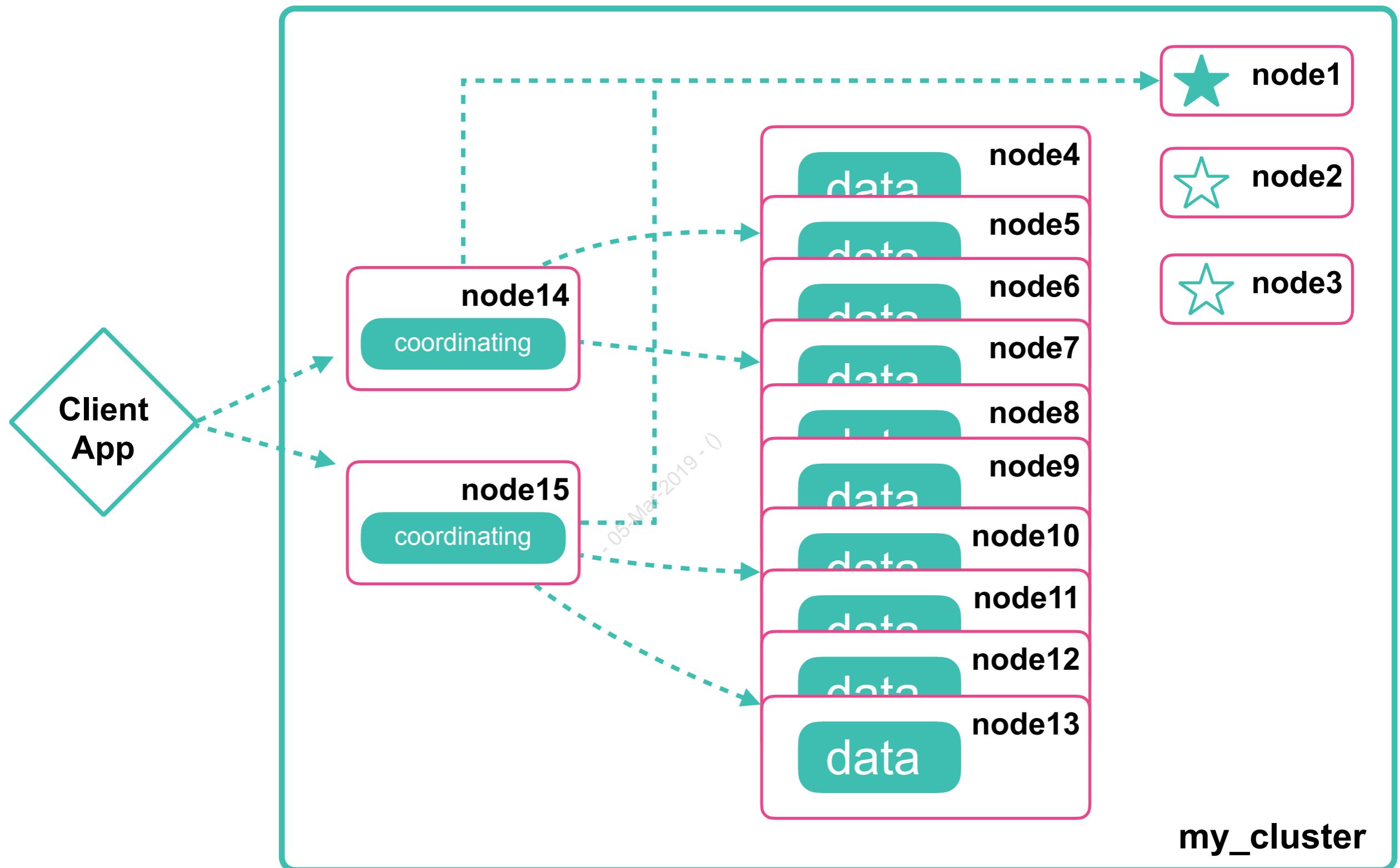


Coordinating Only Node

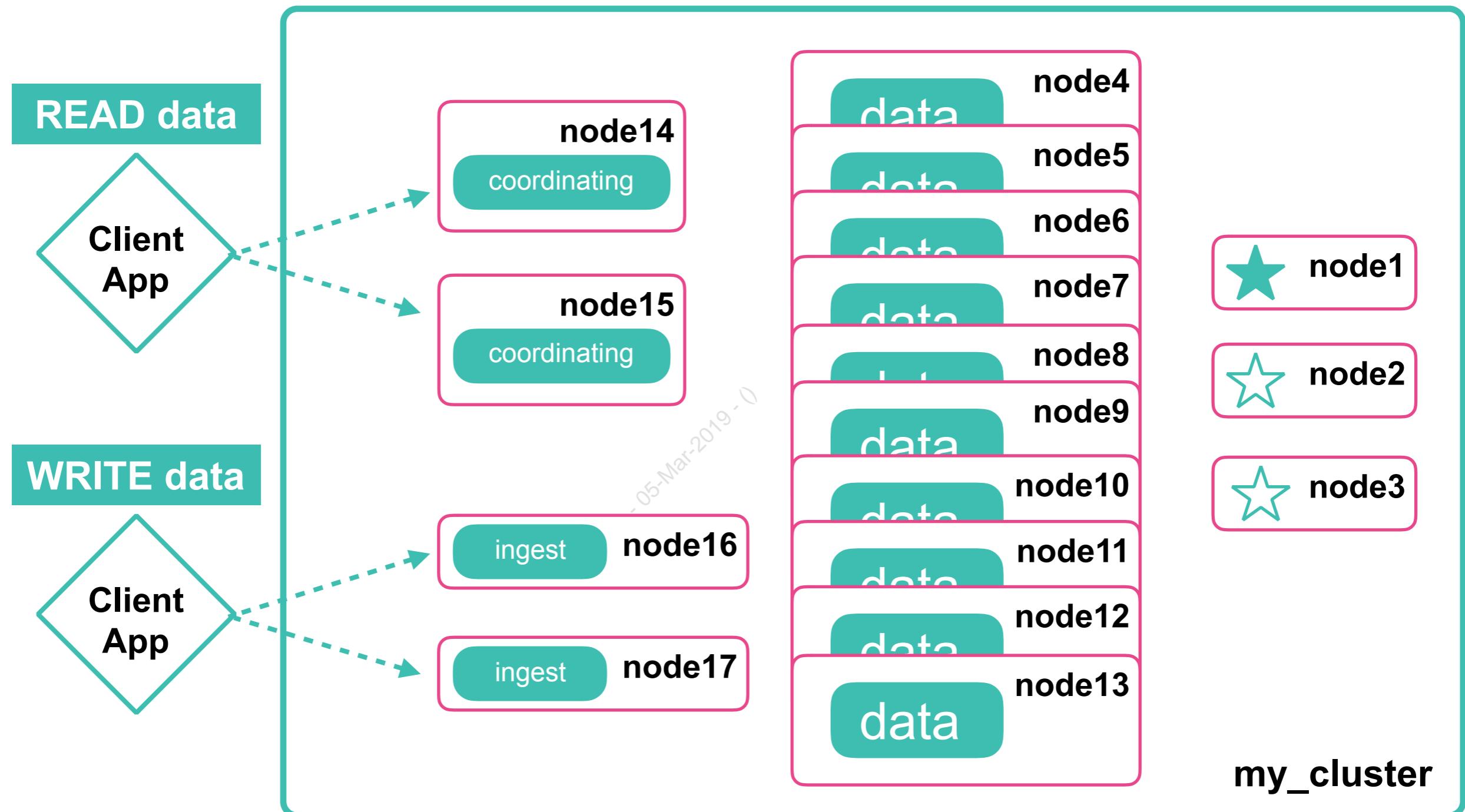
- A ***coordinating only node*** is specifically configured to *not* be a master, data or ingest node
 - machines with low disk, medium/high RAM, and medium/high CPU resources (medium/high depends on use cases)
- useful in specific situations in large clusters
 - behave like smart load balancers
 - perform the gather/reduce phase of search requests
 - lightens the load on data nodes



Coordinating Only Node



Dedicated Nodes Architecture



Hot/Warm Architecture

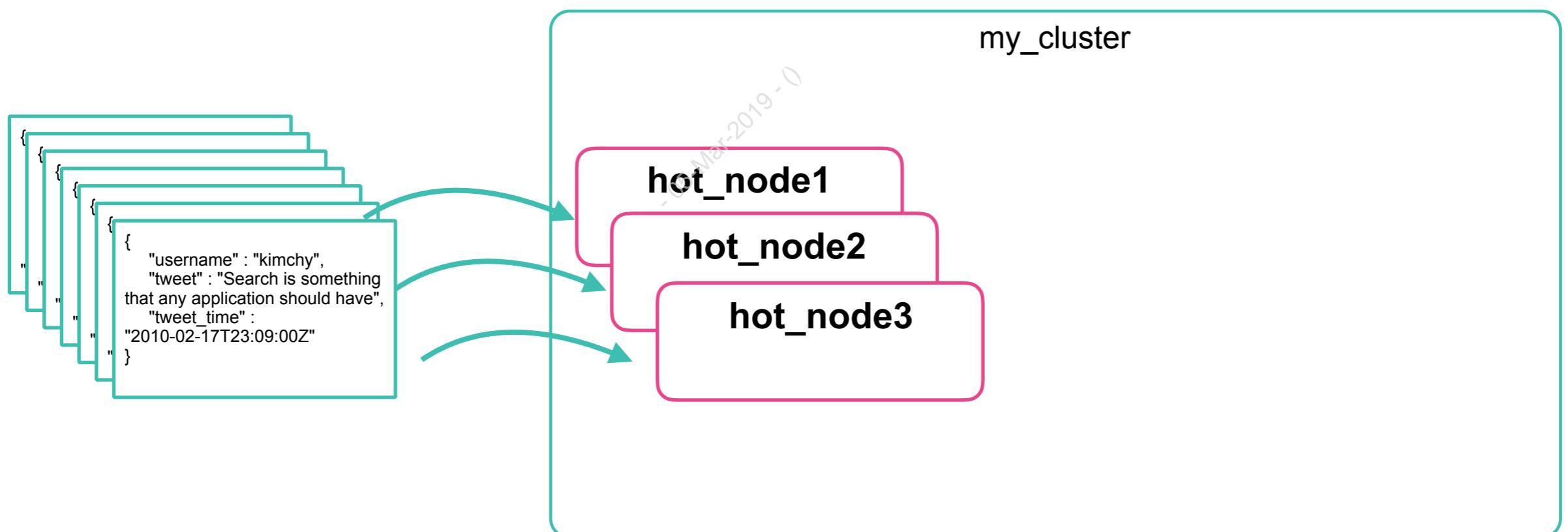
Hot/Warm Architecture

- You can configure the **dedicated data nodes** in your cluster to use a *hot/warm architecture*
 - useful for scenarios where you want to control which nodes perform indexing vs. query handling
- Fine-grained control over **data allocation**
- Dedicated data nodes can be used as:
 - **Hot nodes**
 - for supporting the indices with new documents being written to
 - **Warm nodes**
 - for handling read-only indices that are not as likely to be queried frequently



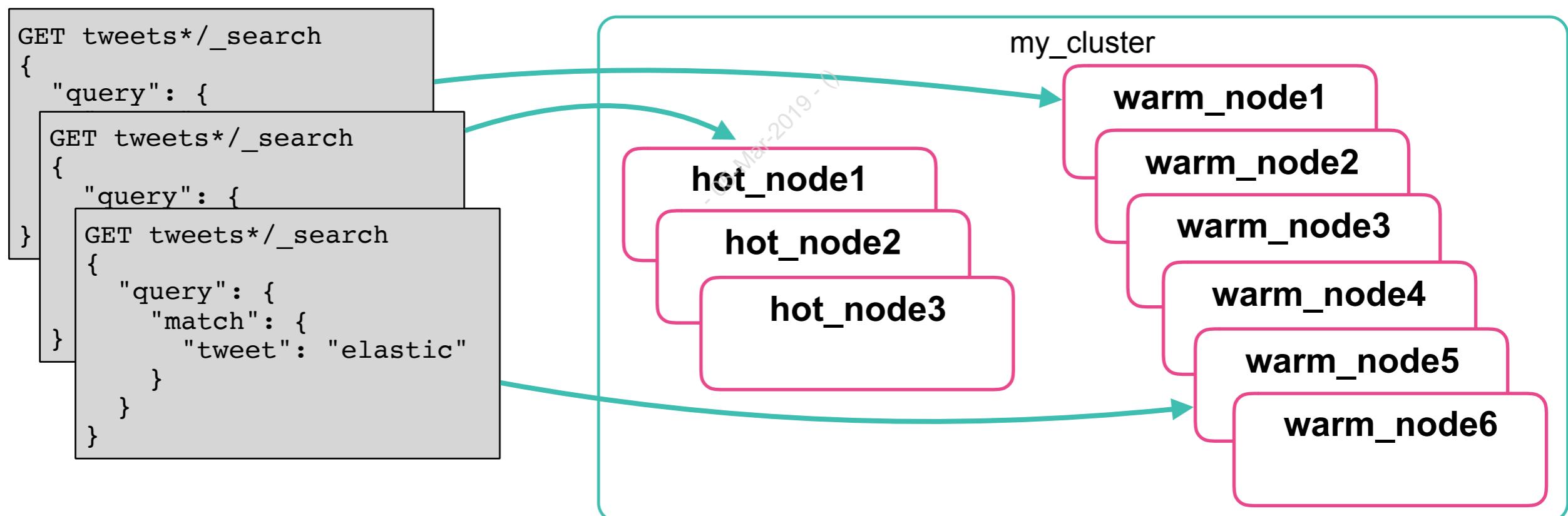
Hot Nodes

- Use *hot nodes* for the indexing
 - indexing is a CPU and IO intensive operation, so hot nodes should be powerful servers
 - faster storage than the warm nodes



Warm Nodes

- Use **warm nodes** for older, read-only indices
 - tend to utilize large attached disks (usually spinning disks)
 - larger amounts of data may require additional nodes to meet performance requirements



Shard Filtering

- How is a hot/warm architecture deployed?
 - you configure ***shard filtering***, which we discuss next...

~05-Mar-2019~0



Shard Filtering

· 05-Mar-2019 · 0

Shard Filtering

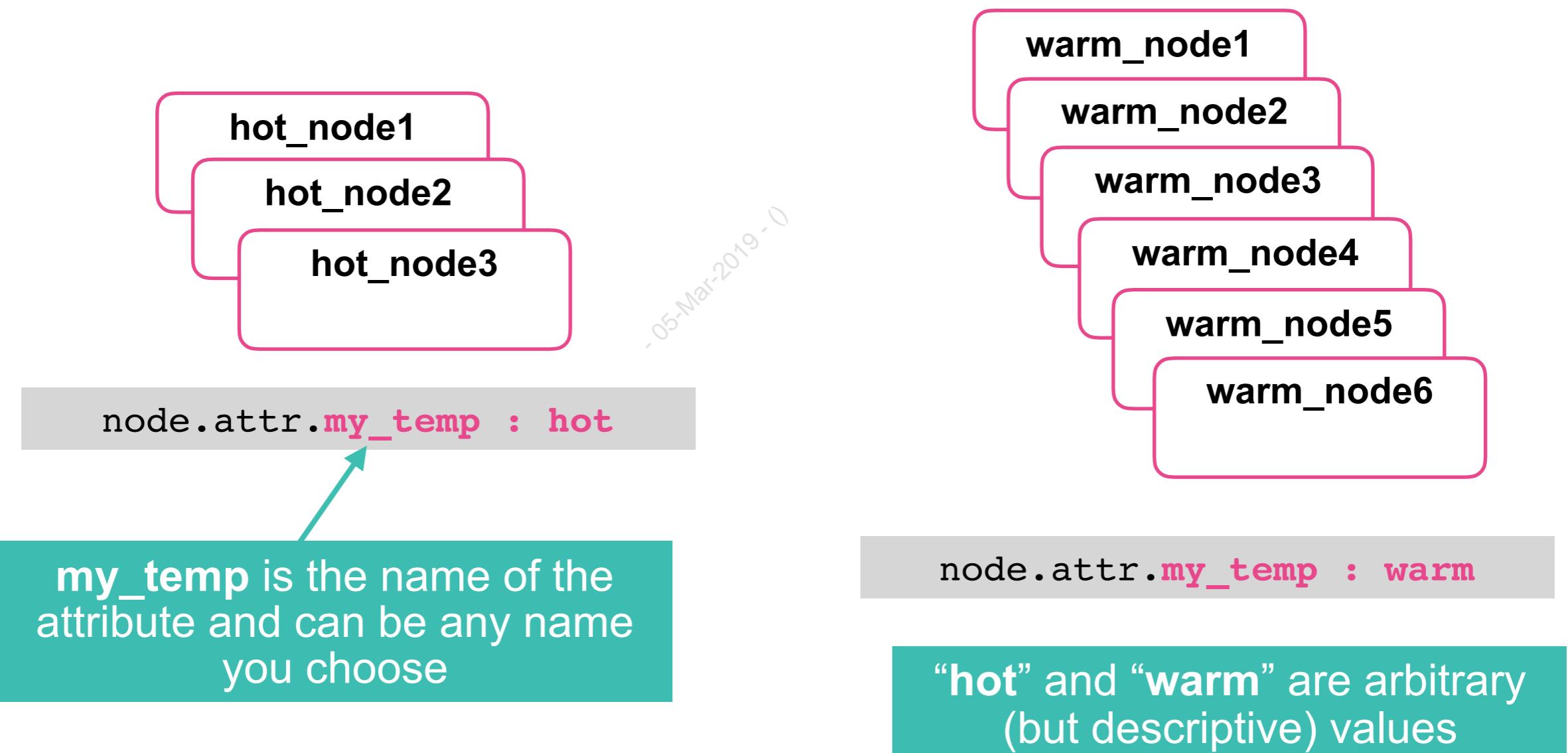
- **Shard filtering** refers to the ability to control which nodes the shards for an index are allocated:
 - use **node.attr** to tag your nodes
 - use **index.routing.allocation** to assign indexes to nodes
- Three types of rules for assigning indexes to nodes:

dynamic setting	assign the index to a node whose {attr} has:
<code>index.routing.allocation.include.{attr}</code>	at least one of the values
<code>index.routing.allocation.exclude.{attr}</code>	none of the values
<code>index.routing.allocation.require.{attr}</code>	all of the values



1. Tag the Nodes

- Step 1: *tag* your nodes using the `node.attr` property:
 - a node attribute can be any name and any value
 - either in `elasticsearch.yml` or the `-E` command line option



2. Configure the Hot Data

- Step 2: configure your indexes to be allocated to the tagged nodes
 - suppose we want the logs from March, 2017, to be allocated to a “hot” node:

```
PUT logs-2017-03
{
  "settings": {
    "index.routing.allocation.require.my_temp" : "hot"
  }
}
```

The shards of
logs-2017-03 will only be
on “hot” nodes

- use index templates to automatically create all new indexes on hot nodes



3. Move Older Shards to Warm

- Let's move the log index from the previous month to "warm" nodes
 - index.routing.allocation** is a dynamic setting, so we can change it using the API:

```
PUT logs-2017-02/_settings
{
  "index.routing.allocation.require.my_temp" : "warm"
}
```

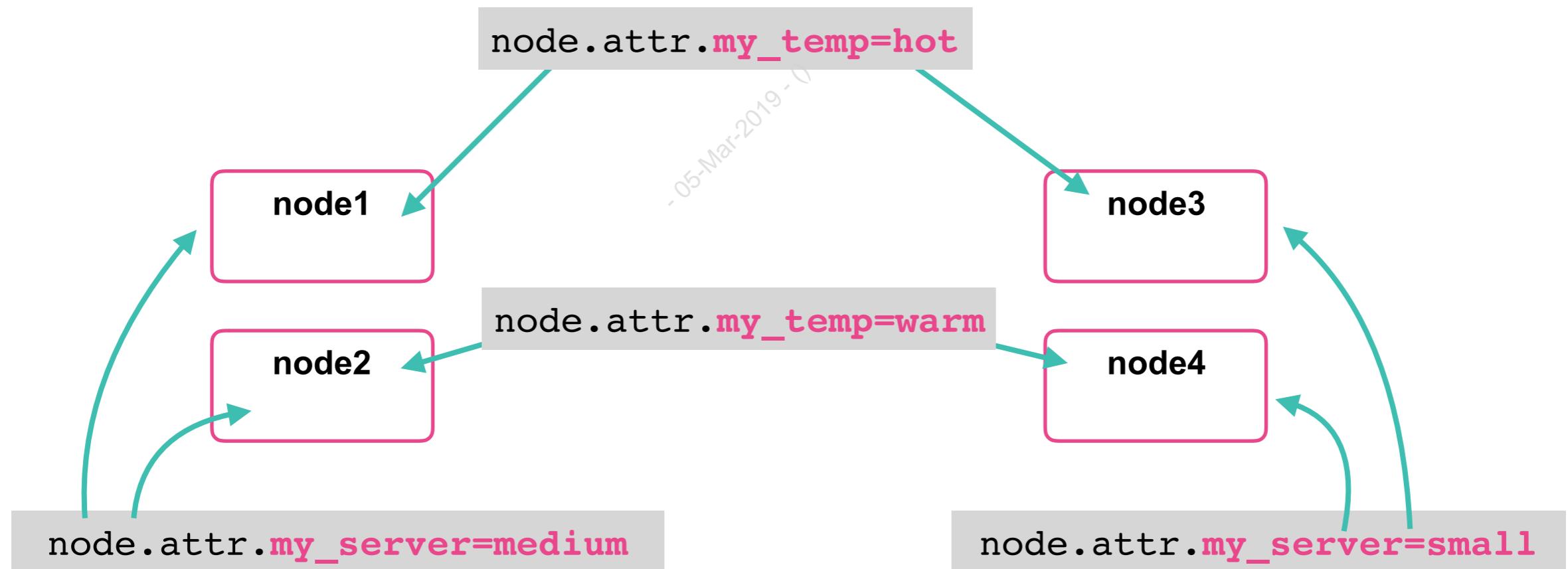
Move the shards from February, 2017
to the warm nodes

Shard Filtering for Hardware

05 Mar 2019 - 0

Shard Filtering for Hardware

- Suppose you are implementing a hot/warm architecture
 - and your nodes are tagged accordingly with “`my_temp`”
- But you also have different sizes of hardware:
 - so you tag those using “`my_server`” as “`small`”, “`medium`”, “`large`”



Configure Your Indices

```
PUT my_index1
{
  "settings": {
    "number_of_shards": 2,
    "number_of_replicas": 1,
    "index.routing.allocation.include.my_server" : "medium",
    "index.routing.allocation.require.my_temp" : "hot"
  }
}
```

*Put **my_index1** on a **medium** server that is also “**hot**”*

```
PUT my_index2
{
  "settings": {
    "number_of_shards": 1,
    "number_of_replicas": 1,
    "index.routing.allocation.include.my_server" : "medium,small",
    "index.routing.allocation.exclude.my_temp" : "hot"
  }
}
```

*Put **my_index2** on any server that is not “**hot**”*

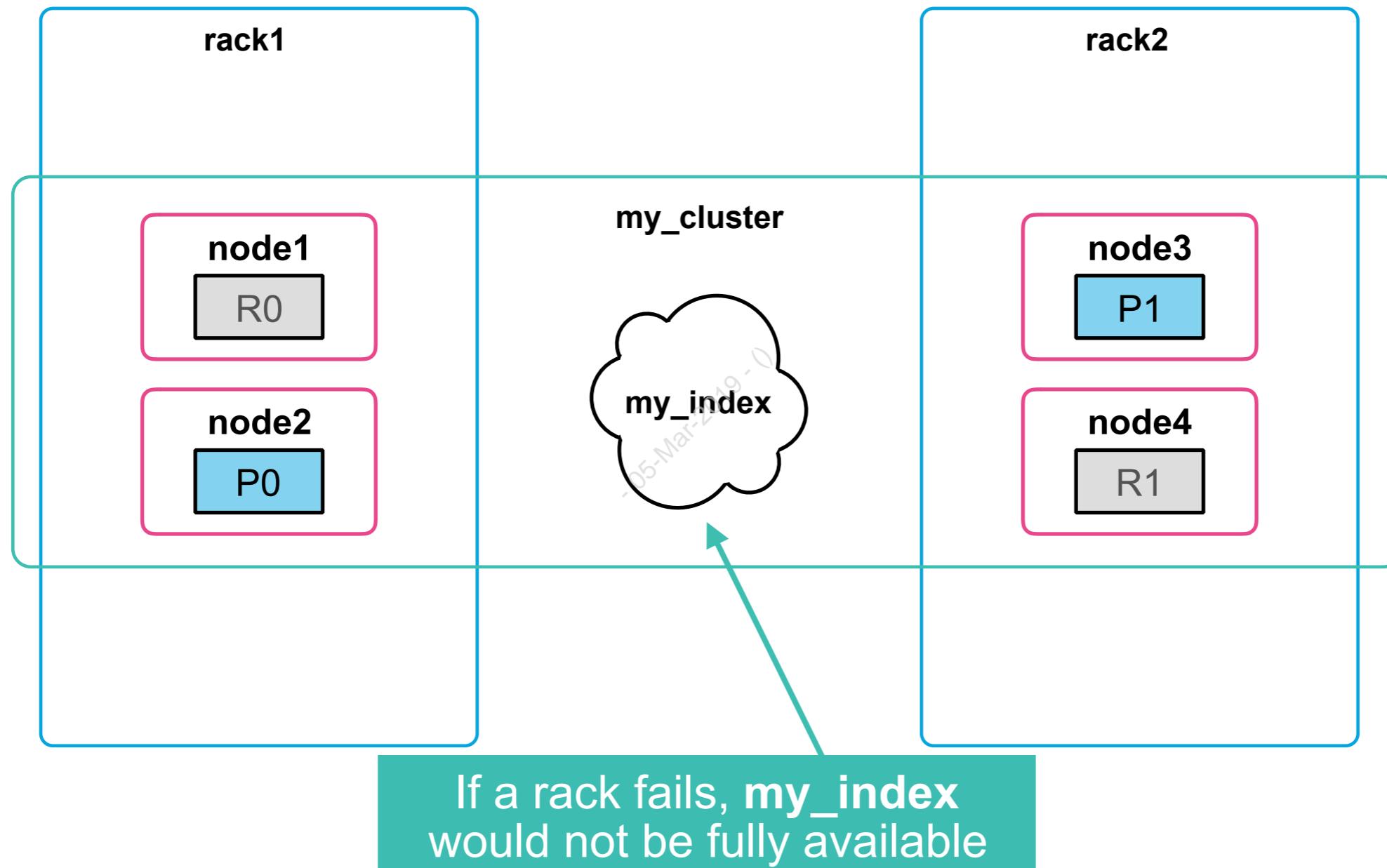


Shard Allocation Awareness

-05 Mar 2019 - 0

Awareness Example

- Suppose your hardware is spread across two racks (or zones, or VMs, or any grouping you have):



Shard Allocation Awareness

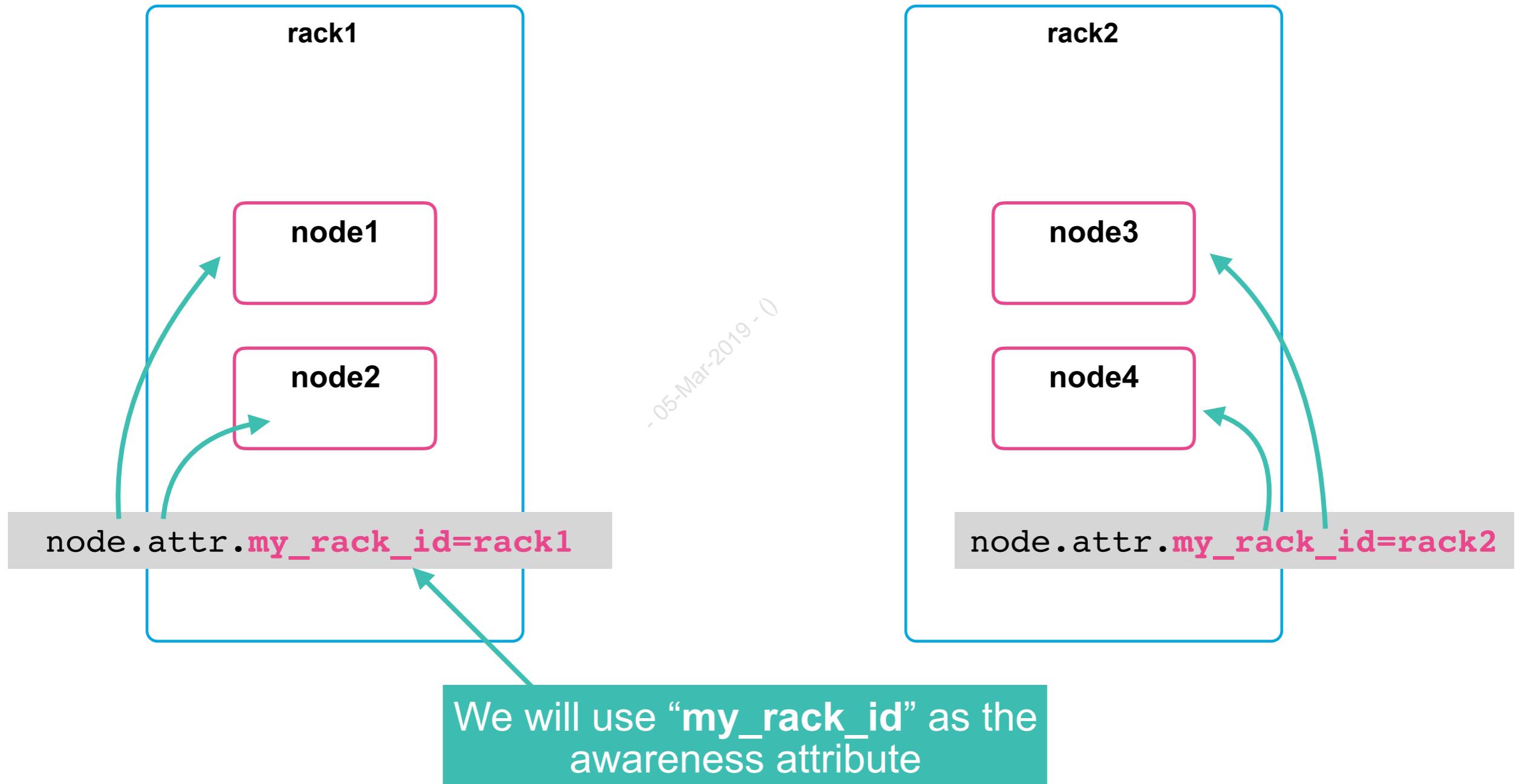
- You can make Elasticsearch aware of the physical configuration of your hardware
 - using **cluster.routing.allocation.awareness**, which is a cluster level setting
 - referred to as ***shard allocation awareness***
- Useful when more than one node shares the same resource (disk, host machine, network switch, rack, etc.)

05-Mar-2019



Step 1: Label Your Nodes

- Use `node.attr` to label your nodes:



Step 2: Configure Your Cluster

- You have to tell Elasticsearch which attribute (or attributes) are being used for awareness:

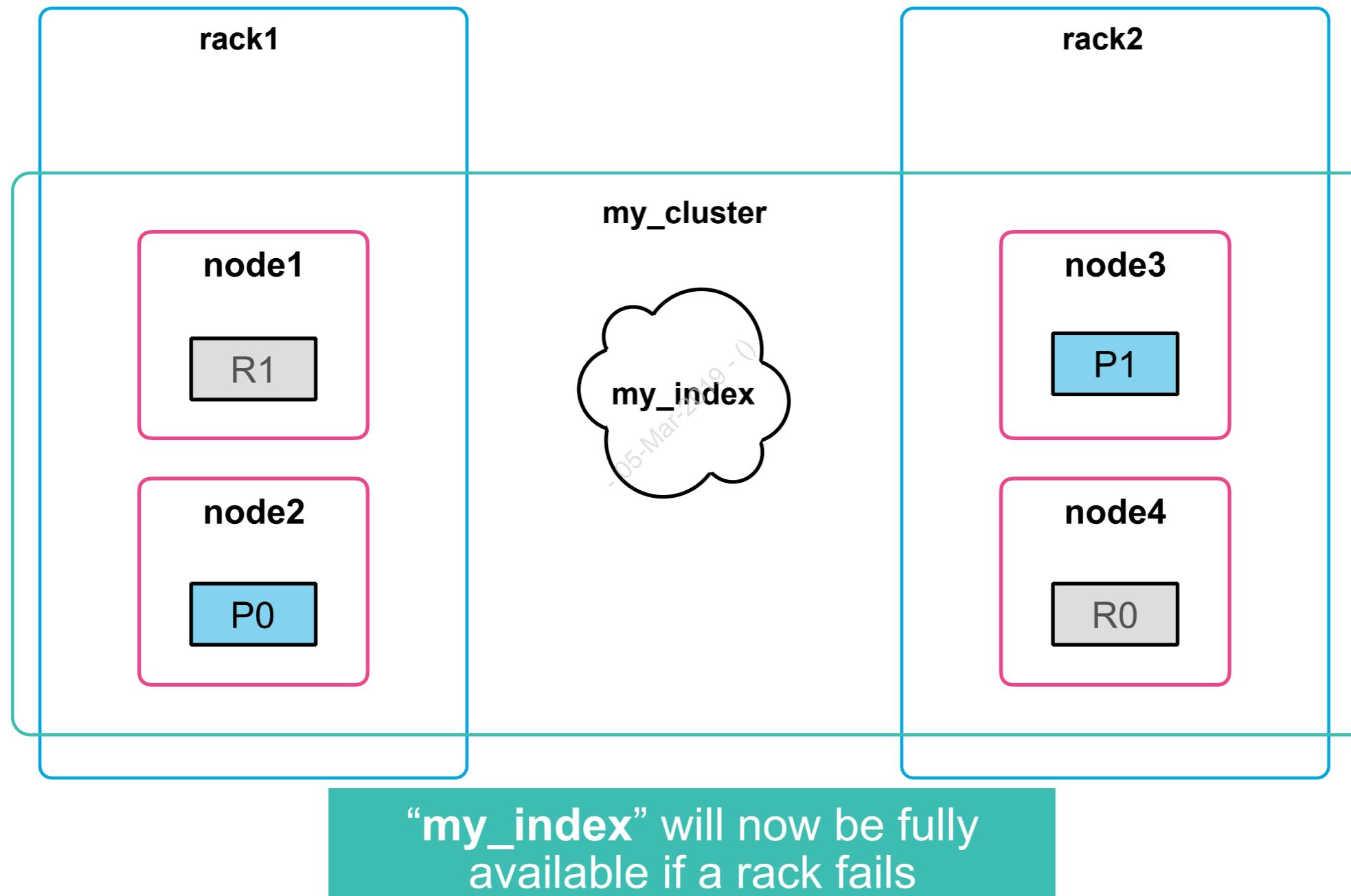
```
PUT _cluster/settings
{
  "persistent": {
    "cluster.routing.allocation.awareness.attributes": "my_rack_id"
  }
}
```

The name of the node
attribute you defined for
awareness



Awareness Example

- Now you are guaranteed that at least one copy of all shards will exist in each rack for each index:

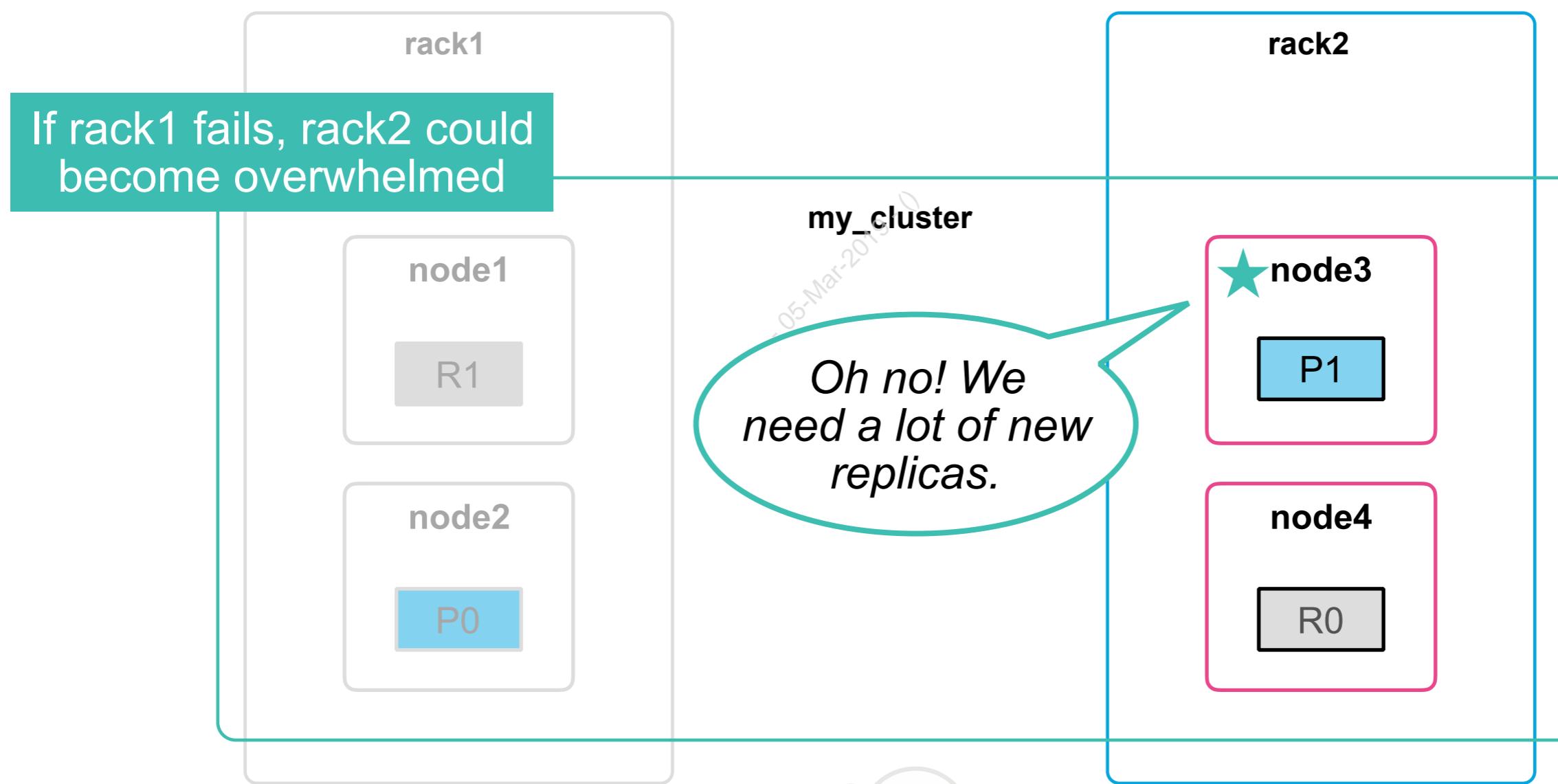


Forced Awareness

~05-Mar-2010

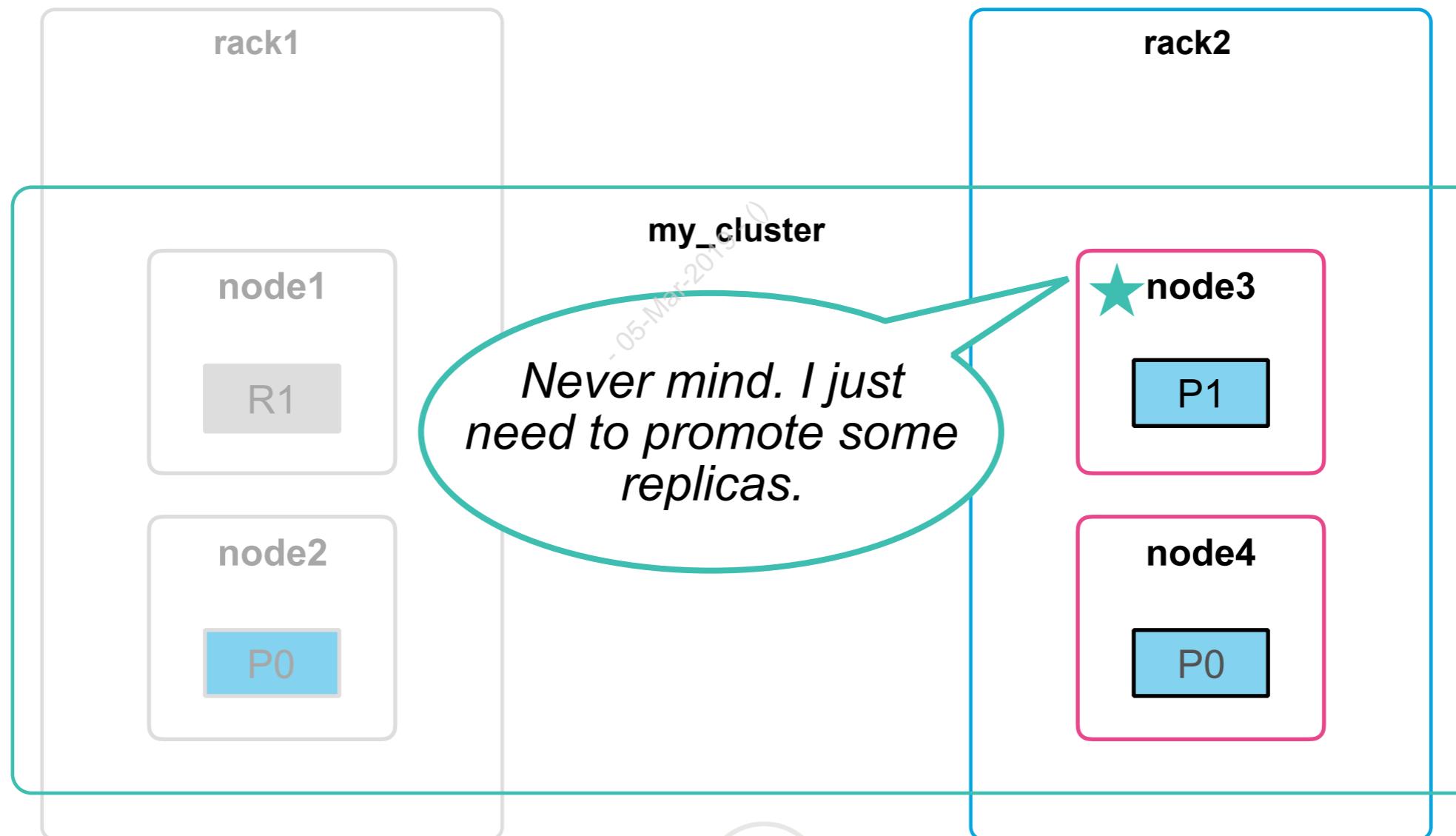
Why Forced Awareness?

- Suppose you have a rack (or zone) that fails
 - the remaining rack will try to reassign all the missing replicas
 - the single rack might not be able to handle that type of volume



Forced Awareness

- You can configure **forced awareness** to avoid overwhelming a zone
 - never allows copies of the same shard to be in the same zone



Configure Forced Awareness

- You have to tell Elasticsearch which attributes and values to use for forced awareness:
 - In this example, no copies of the same shard will appear on **rack1** and **rack2**

```
PUT _cluster/settings
{
  "persistent": {
    "cluster": {
      "routing": {
        "allocation.awareness.attributes": "my_rack_id",
        "allocation.awareness.force.my_rack_id.values": "rack1,rack2"
      }
    }
  }
}
```

this setting is the **name** of the attribute

this setting contains the **values** of the attribute



Chapter Review

~05-Mar-2019~

Summary

- Dedicated nodes can help you to better utilize hardware
- Coordinating-only nodes lightens the load on data nodes in some use cases
- You can use shard filtering to configure a ***hot/warm architecture*** for your cluster
- ***Shard filtering*** refers to the ability to control to which nodes an index is allocated
- You can make Elasticsearch aware of the physical configuration of your hardware using **cluster.routing.allocation.awareness**
- You can configure ***forced awareness*** to avoid overwhelming a rack or zone of servers



Quiz

1. Suppose you created a new index every day for that day's log files. How could this scenario benefit from a hot/warm architecture?
2. What happens if you configure an index's shard filtering with a scenario that is impossible for the cluster to implement?
3. Why configure shard allocation awareness if you have already configured shard allocation filtering?
4. What is the benefit of forced awareness over simply configuring shard allocation awareness?



Lab Cluster Architecture

Node	Server	Type	Tags
node1	server1	dedicated master-eligible	none
node2	server2	data and ingest	hot, rack1
node3	server3	dedicated data	warm, rack1
node4	server4	data and ingest	hot, rack2
node5	server5	dedicated data	warm, rack2



Lab 5

Cluster Management

.05-Mar-2019.0



Chapter 6

Capacity Planning

.05-Mar-2019-0

- 1 Elasticsearch Internals
- 2 Field Modeling
- 3 Fixing Data
- 4 Advanced Search & Aggregations
- 5 Cluster Management
- 6 Capacity Planning
- 7 Document Modeling
- 8 Monitoring and Alerting
- 9 From Dev to Production



Topics covered:

- Designing for Scale
- Capacity Planning
- Scaling with Replicas
- Scaling with Indices
- Capacity Planning Use Cases

~05-Mar-2019~0



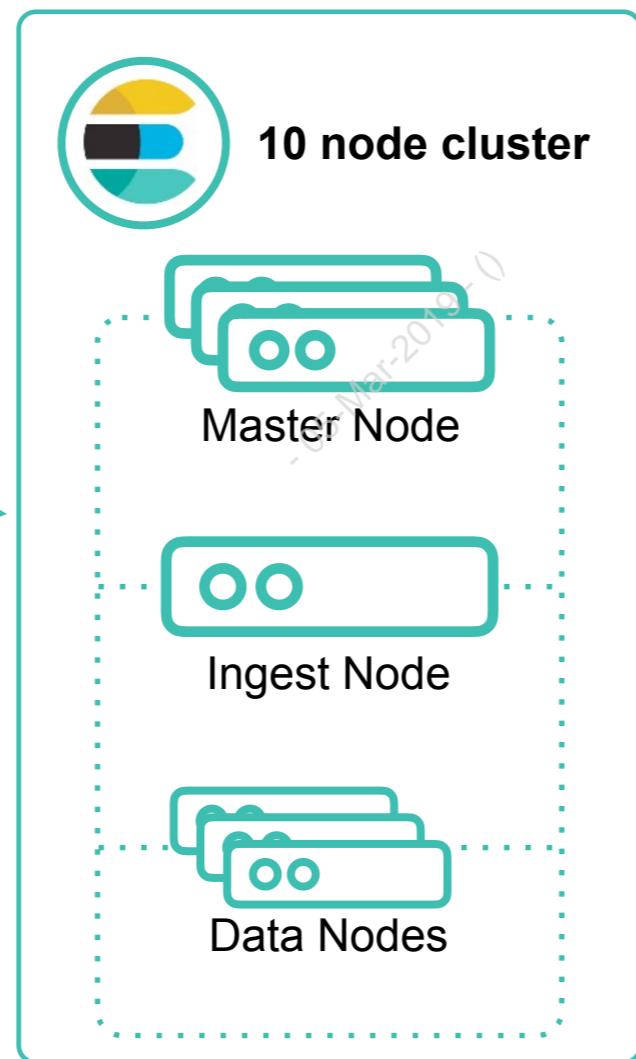
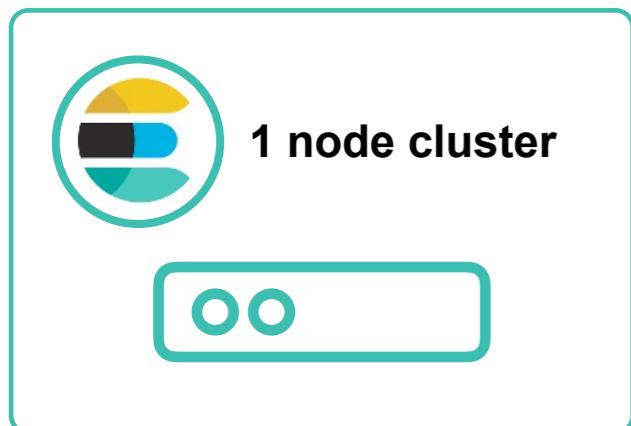
Designing for Scale

~05-Mar-2016 ~

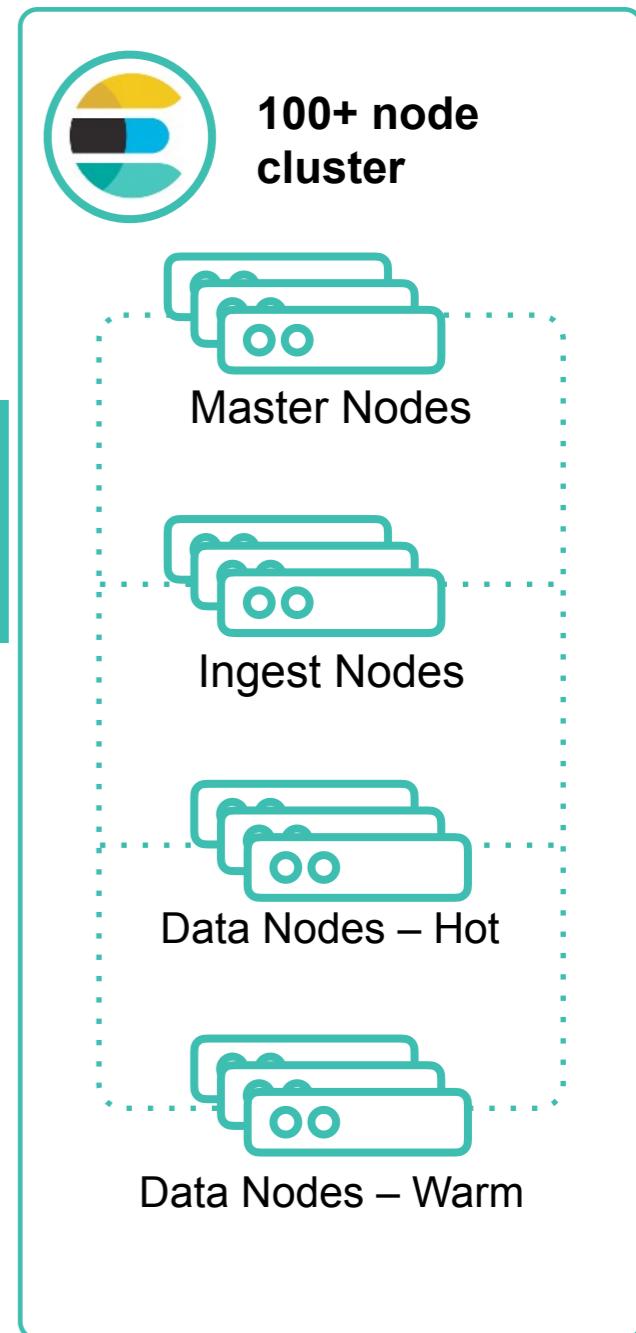
Designing for Scale

- Elasticsearch is built to scale
 - and the default settings can take you a long way
- Proper design can make scaling easier

Growing from 1 to 10
can be easy...

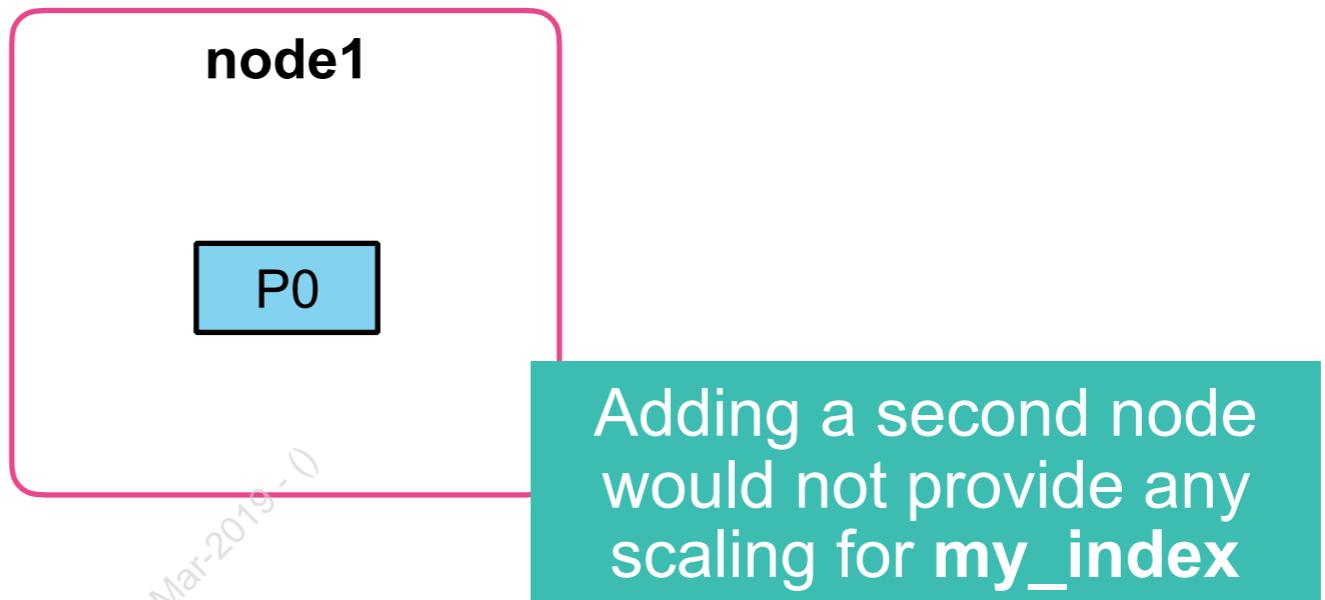


...and so can
growing from
10 to 100



One Shard...

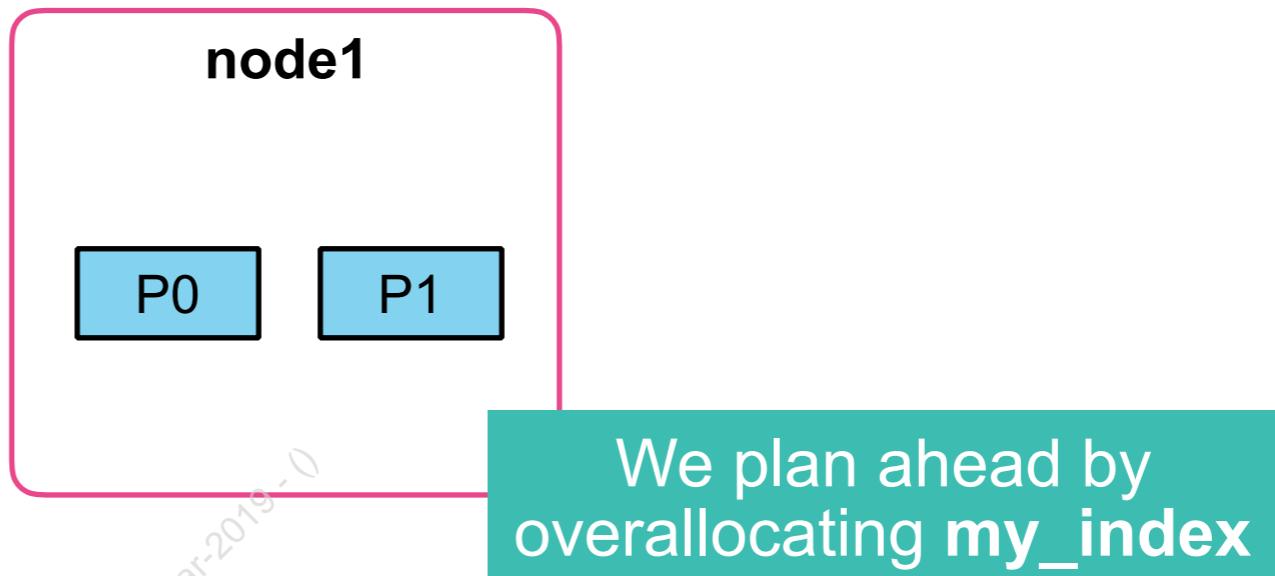
- ...does not scale very well:



```
PUT my_index
{
  "settings": {
    "number_of_shards": 1,
    "number_of_replicas": 0
  }
}
```

Two Shards...

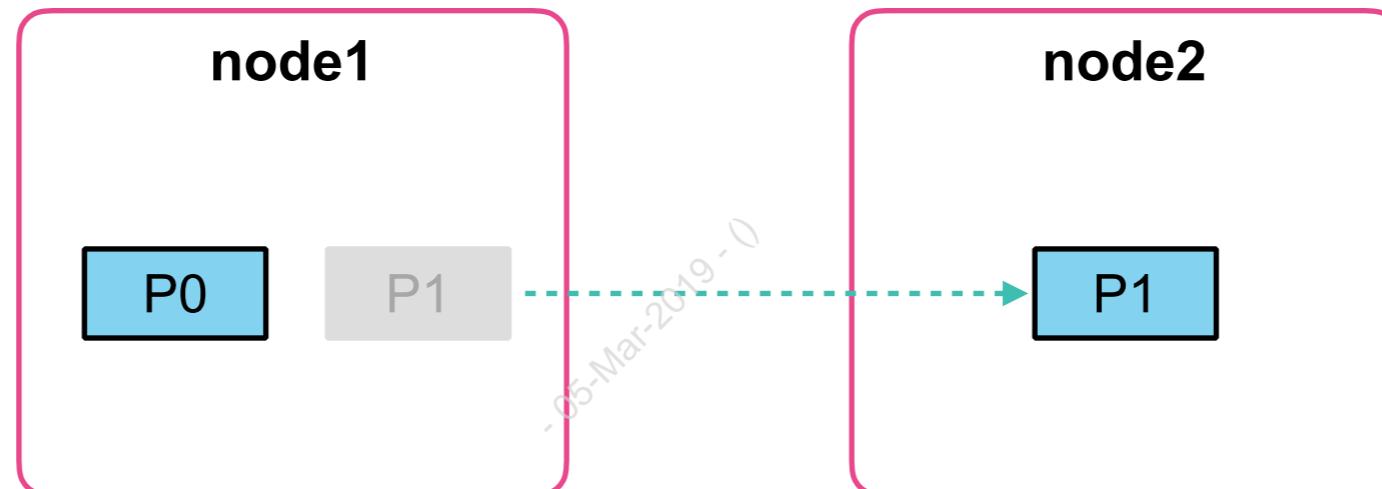
- ...can scale if we add a node:



```
PUT my_index
{
  "settings": {
    "number_of_shards": 2,
    "number_of_replicas": 0
  }
}
```

Balancing of Shards

- Elasticsearch automatically balances the shards:

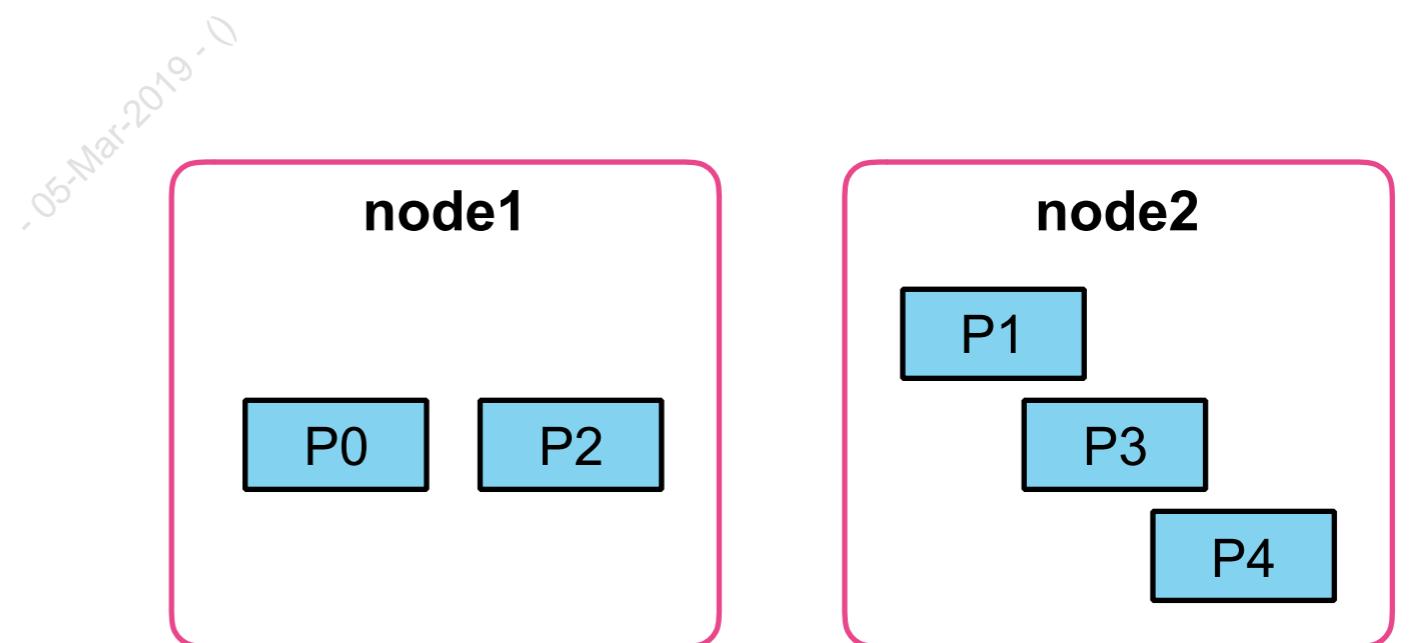


Adding a node causes
the cluster to rebalance

Shard Overallocation

- If you are expecting your cluster to grow, then it is good to plan for that by overallocating shards:
 - $\#shards > \#nodes$
- Shards can move between nodes quickly as the cluster grows
 - and there is no downtime during shard relocation

```
PUT my_index
{
  "settings": {
    "number_of_shards": 5,
    "number_of_replicas": 0
  }
}
```



Be careful

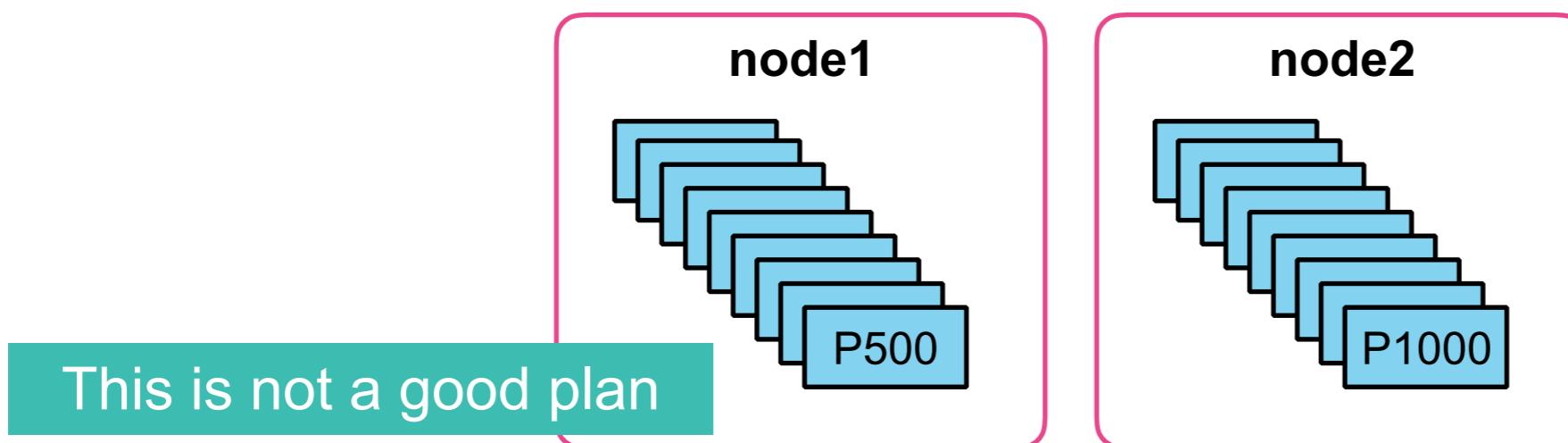
- The scaling unit is the shard
- But you should NOT have too many shards
 - specially sub-utilized shards
- Each shard uses resources from the machine
 - if you have too many it will seriously slow things down
- And if you have too many shards, Elasticsearch will be slow

05-Mar-2019



“Too Much” Overallocation

- A little overallocation is good
- A “kagillion” shards is not:
 - each shard comes at a cost (Lucene indices, file descriptors, memory, CPU)
 - also, a search request needs to hit every shard in the index
- A shard can optimally hold from 10 to 40 gigabytes
 - optimal depends on the use case
 - a 1GB shard is sub-utilized



Do not Overshard

- Business Requirements

- 1GB per day
 - 6 months retention
 - **~180GB**

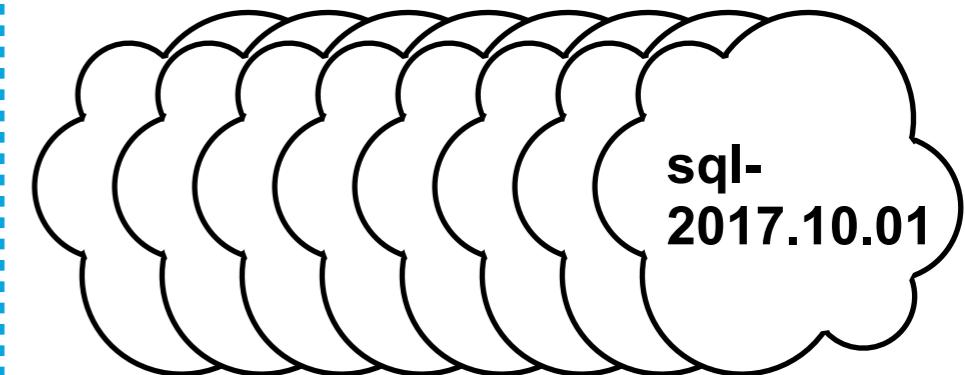
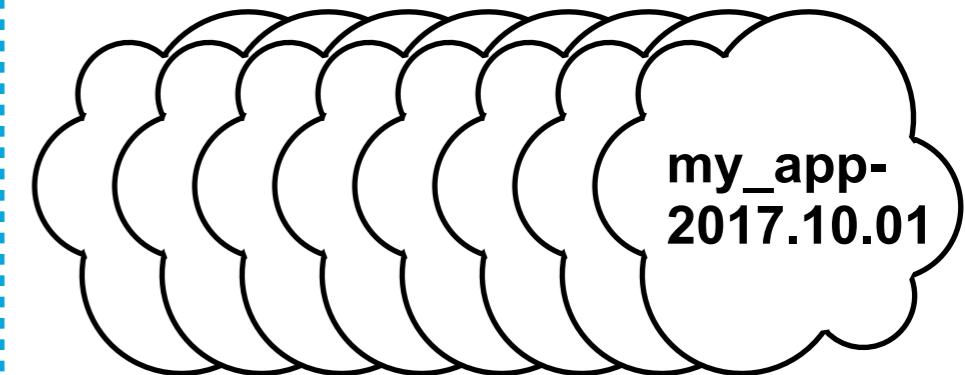
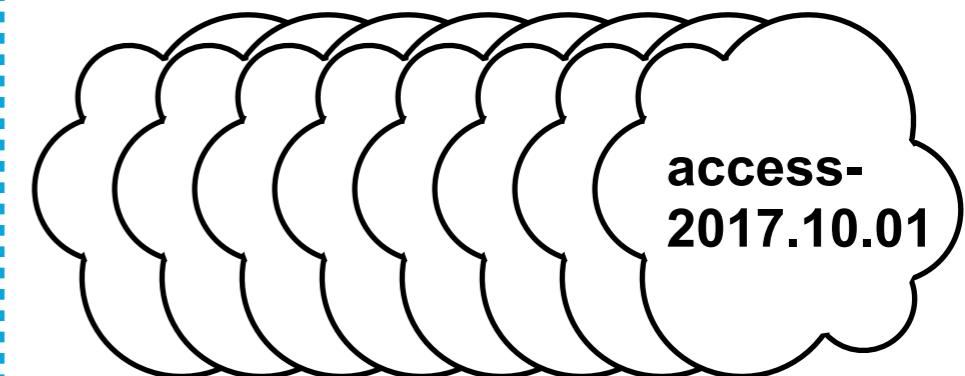
we could easily have this data in 10 shards

- Common Scenario

- 3 different logs
 - 1 index per day each
 - 5 shards (default)
 - 6 months retention
 - **~2700 shards**

too many shards for no good reason!

Cluster *my_cluster*



Capacity Planning

· 05-Mar-2019 · 0

Capacity Planning

- So how many shards should I configure for my index?
 - no simple formula to it
 - “It depends!”
- Too many factors involved:
 - use case: metrics, logging, search, apm, etc.
 - hardware
 - # of documents
 - size and complexity of your documents
 - how you index the data
 - how you search and aggregate the data
 - how many indices the data will be spread across



Capacity Planning

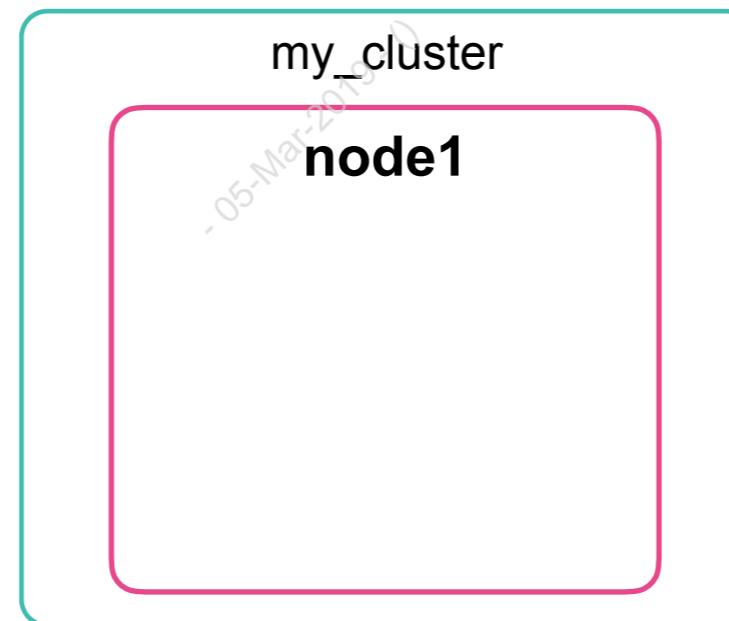
- Before trying to determine your capacity, you need to determine your SLA(s):
 - How many ***docs/second*** do you need to index?
 - How many ***queries/second*** do you need to process?
 - What is the ***maximum response time*** for queries?
- Get some production data
 - ***actual documents*** you are going to index
 - ***actual queries*** you are going to run in production
 - ***actual mappings*** you are going to use



Maximum Shard Capacity

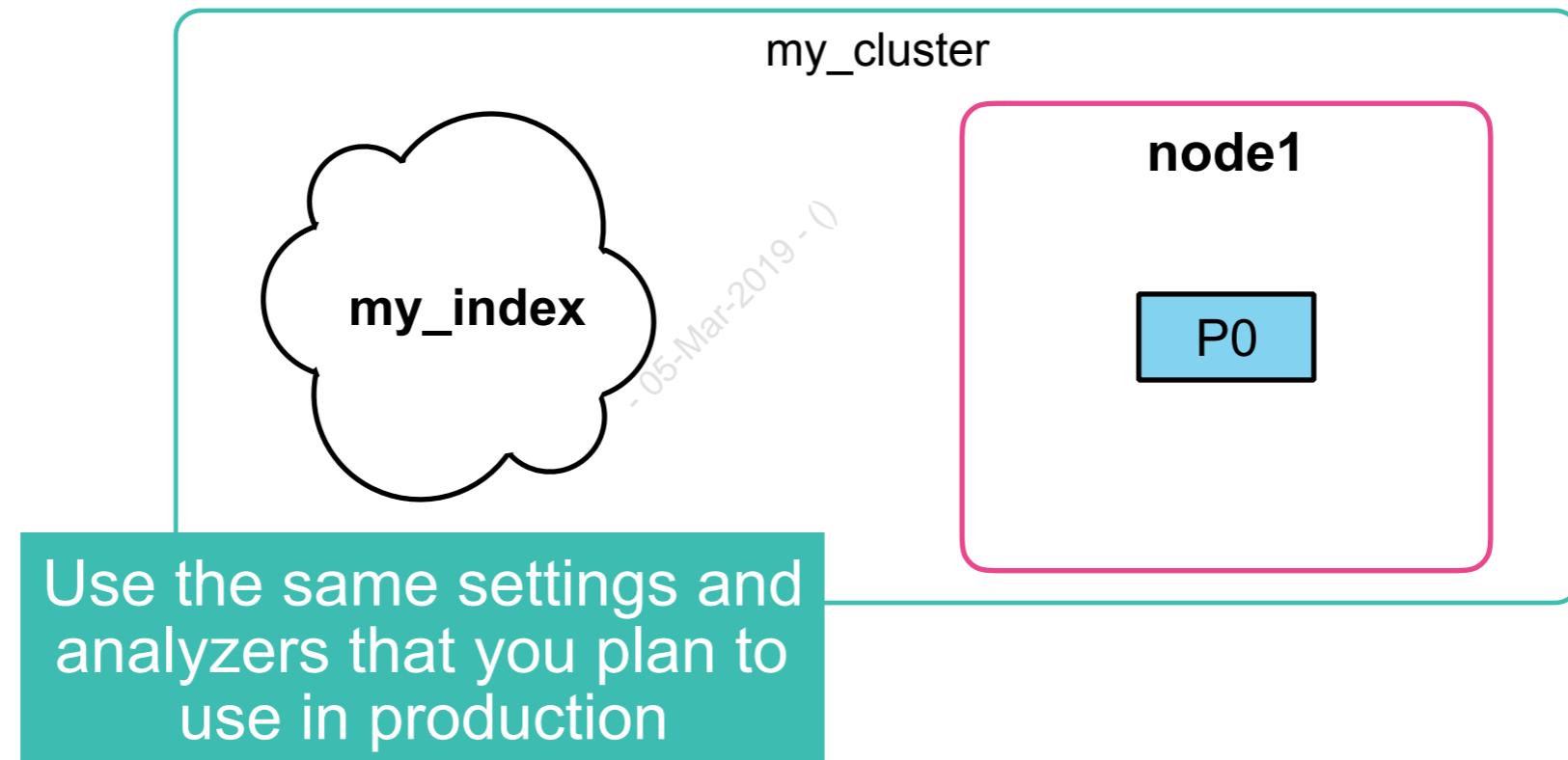
- You can evaluate the maximum shard size for your particular use case

1. Create a 1-node cluster using your production hardware



Maximum Shard Capacity

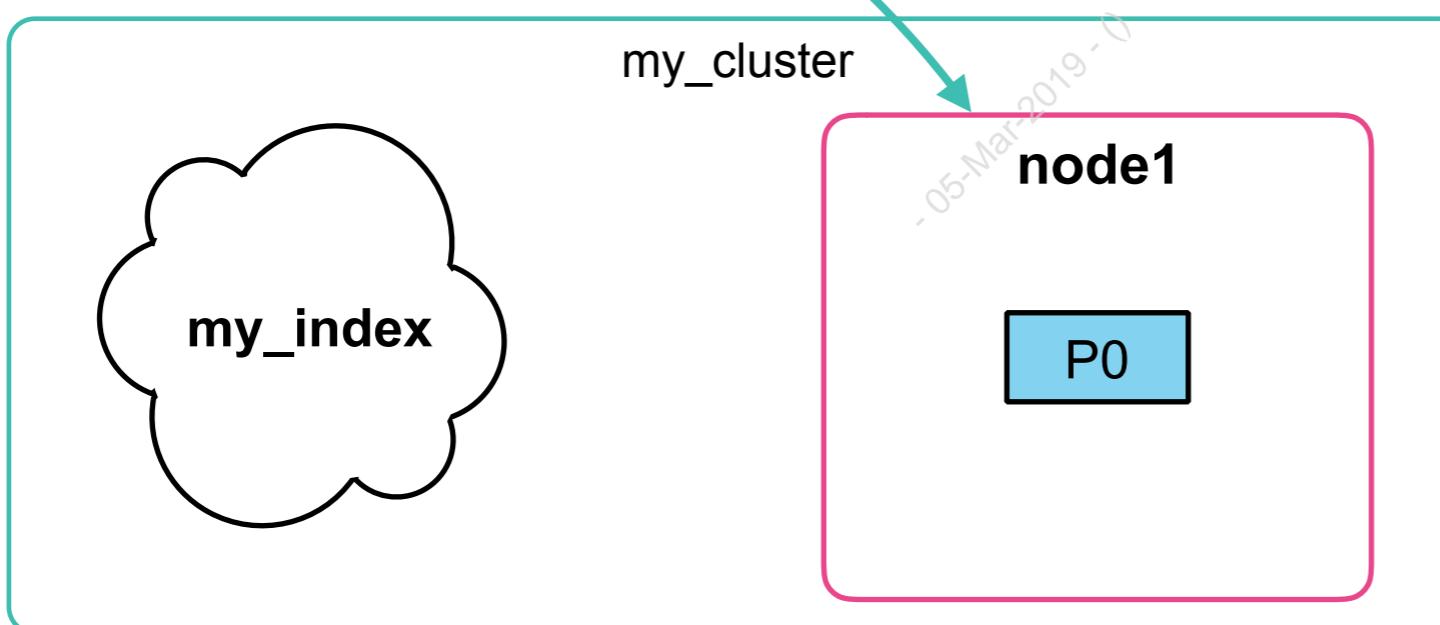
2. Create a single index with 1 shard and no replicas



Maximum Shard Capacity

```
GET my_index/_search
{
  ...
}
```

3. Incrementally index documents and run your searches and aggregations. Push this shard and index docs until it “breaks” and you will find its max capacity



“**breaks**” depends on your own definition - ingestion rates, search rates, latency, etc.

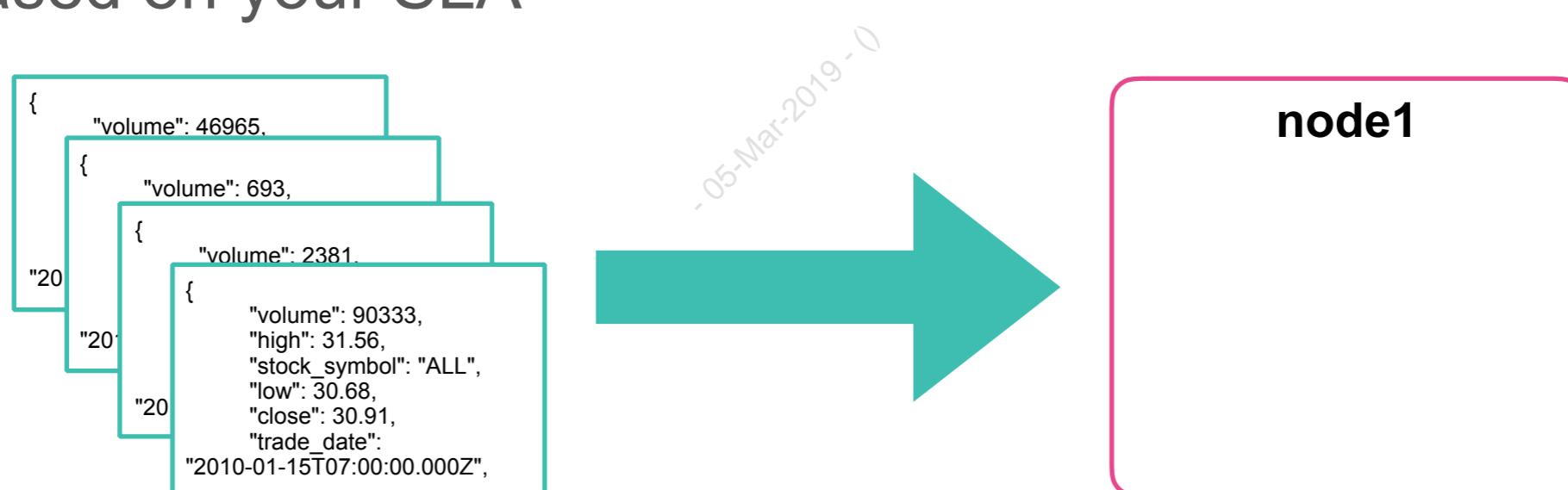
Number of Primary Shards

- It is not an exact science, but you can:
 - estimate the total amount of data for your index
 - leave room for growth (if applicable)
 - divide by the maximum capacity of a single shard
- The number of primary shards is usually determined by indexing speed
- For searching, the total number of shards is the measurement (agnostic of index)
 - note we are not talking about replicas here, just number of primary shards
 - we will talk about replicas next...



Measure Indexing Capacity of a Node

- You can calculate the indexing capacity of a single node
 - index documents in a similar way as your application will
 - index in parallel until 429 responses begin to come back
 - use this to calculate the number of **docs/second**
 - you can now calculate how to scale your nodes for ingestion based on your SLA



Index docs until a 429 response, which mean ES can not keep up with indexing



Scaling with Replicas

~05-Mar-2019 CQ



Scaling with Replicas

- Adding *replicas* provides an additional level of scaling
 - We know replicas provide high availability
 - They also provide scaling for reads and searches (if you add additional hardware)

Suppose our index has
one shard on one node

```
PUT my_index
{
  "settings": {
    "number_of_shards": 1,
    "number_of_replicas": 0
  }
}
```

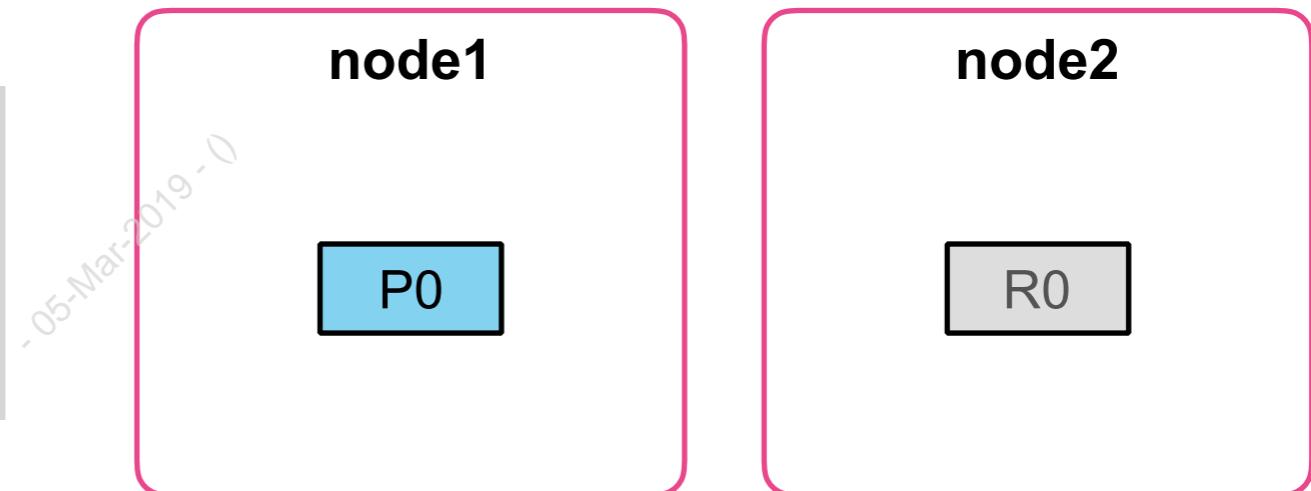
node1

P0

Scaling with Replicas

- If you add another node, you can increase the number of replicas:
 - “`number_of_replicas`” is a dynamic setting

```
PUT my_index/_settings
{
  "number_of_replicas": 1
}
```



Adding a node *and* replicas adds compute power to the cluster



The Cost of Replicas

- Replicas have a cost associated with them as well, including:
 - slower indexing speed (although replicas are indexed in parallel, so it is not a cumulative cost)
 - more storage on disk
 - larger associated heap memory footprint of the extra shard(s)

~05-Mar-2019~0



Scaling with Indices

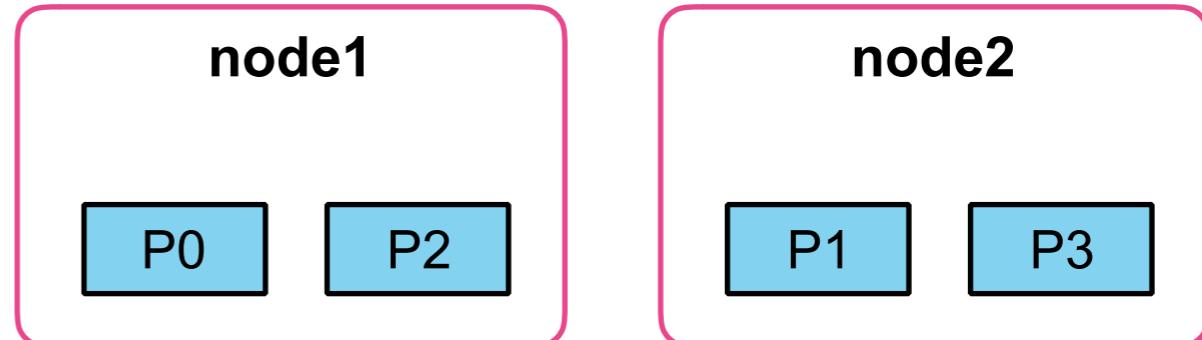
~05-Mar-2019 ~10

Scaling with Indices

- Using *multiple indices* also provides scaling
 - If you need to add capacity, consider just creating a new index
- Then search across both indices to search “new” and “old” data
 - you could even define a single alias for the multiple indices
- Searching 1 index with 50 shards is equivalent to searching 50 indices with 1 shard each

Scaling with Indices

```
PUT my_index
{
  "settings": {
    "number_of_shards": 4
  }
}
```

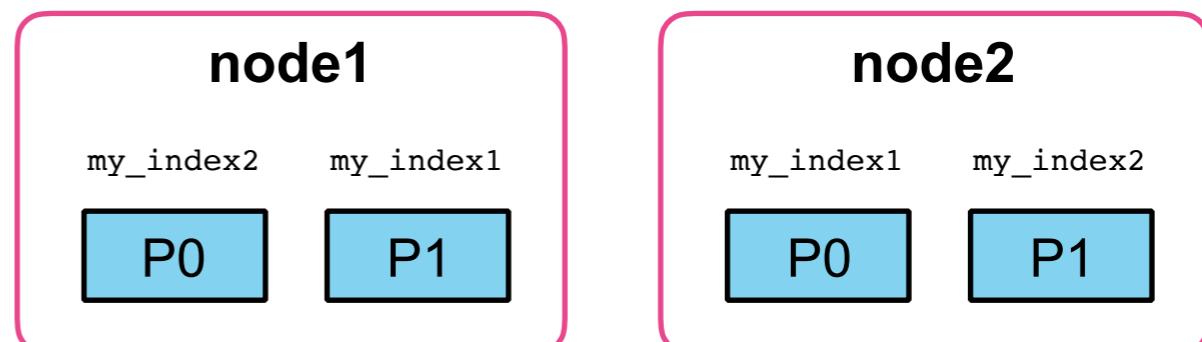


```
PUT my_index1
{
  "settings": {
    "number_of_shards": 2
  }
}

PUT my_index2
{
  "settings": {
    "number_of_shards": 2
  }
}
```

05-Mar-2019

Searching **my_index** hits 4 shards.
So does a search over
my_index1,my_index2



Capacity Planning Use Cases

25-Mar-2019 - 0

Capacity Planning Use Cases

- When planning your cluster and designing indices, it is important to understand:
 - what your data looks like
 - how that data is going to be searched
- Two very common use cases are:
 - **searching fixed-size data:** searching a large dataset that may grow slowly
 - **time-based data:** data that grows rapidly, like log files



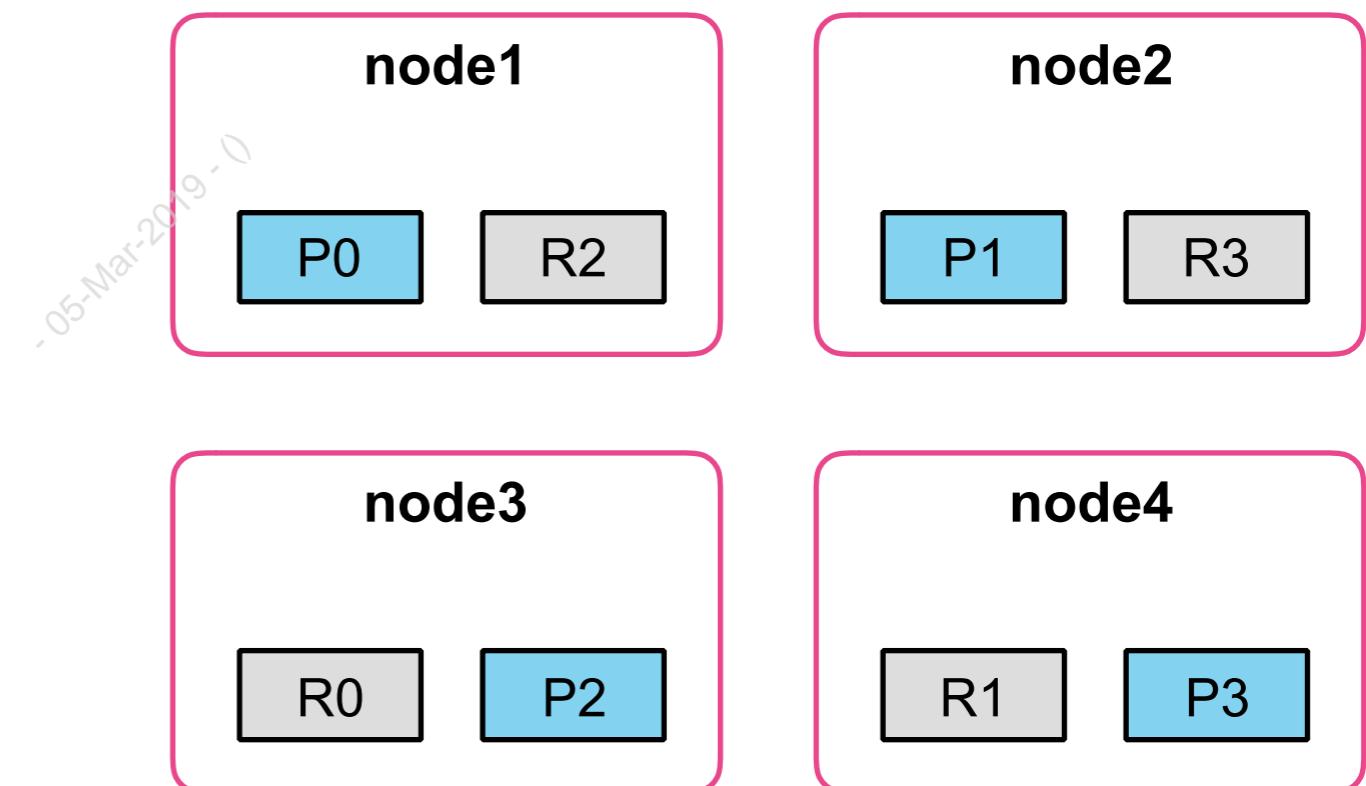
Fixed-size Data

~05-Mar-2019 (~)

Fixed-size Data

- Your use case may involve searching a large dataset that only gradually grows and/or changes
 - relatively **fixed-size** collection of documents
 - search for relevant documents, no matter their age

```
PUT hotels
{
  "settings": {
    "number_of_shards": 4,
    "number_of_replicas": 1
  }
}
```

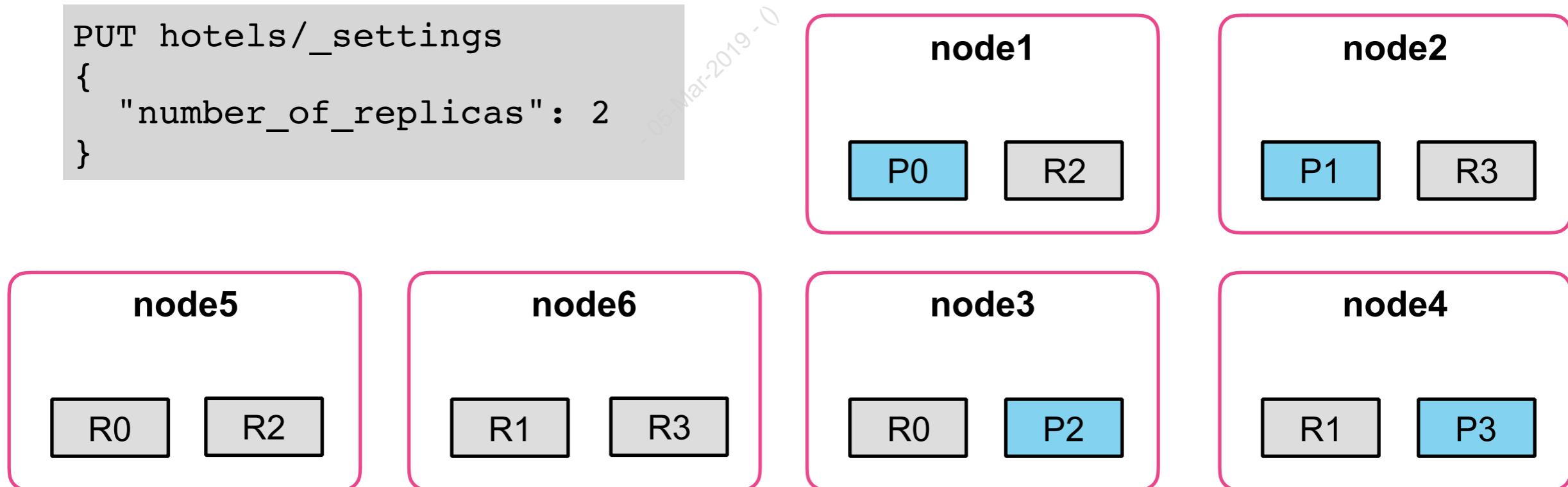


Planning for Fixed-size Data

- In this case, size your indices based on the maximum shard capacity vs. the amount of data you anticipate
 - If you need to increase capacity, either reindex or add multiple indices (but preferably not very often)
 - Easy to increase throughput by adding more nodes and replicas

```
PUT hotels/_settings
{
  "number_of_replicas": 2
}
```

© Seminar-2019 - Ø



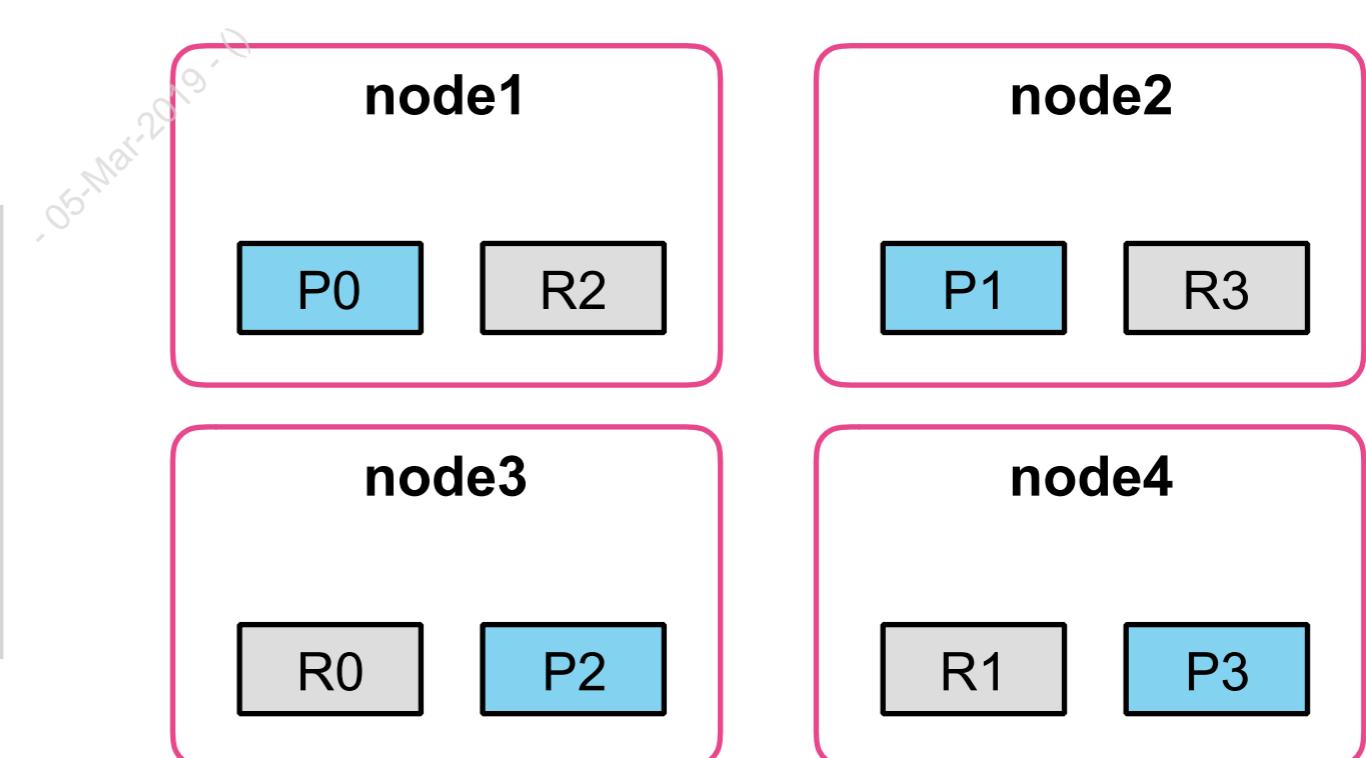
Time-based Data

~05-Mar-2019 (~)

Time-based Data

- *Time-based data* includes:
 - logs
 - social media streams
 - time-based events
- These documents have a timestamp, and likely do not change

```
PUT tweets-2017-02-05
{
  "settings": {
    "number_of_shards": 4,
    "number_of_replicas": 1
  }
}
```



Planning for Time-based Data

- ***Searching*** on time-based data usually involves a timestamp
 - you typically search for recent events
 - older documents become less important
- ***Data ingestion*** is another key factor to consider
 - you typically have a lot of data coming in
 - you do not want indexing to become a bottleneck (it has to keep up)



Index per Time Frame

- Time-based data is best organized using *time-based indices*
 - create a new index each day, week, month, year (whatever is appropriate for the amount of data being ingested)
 - add the date to the name of the index

```
PUT tweets-2017-02-05
{
  "settings": {
    "number_of_shards": 4,
    "number_of_replicas": 1
  }
}

PUT tweets-2017-02-06
{
  "settings": {
    "number_of_shards": 4,
    "number_of_replicas": 1
  }
}

PUT tweets-2017-02-07
{
  "settings": {
    "number_of_shards": 4,
    "number_of_replicas": 1
  }
}
```



Searching Time-based Indices

- Searches can use **wildcards** or **aliases** to search over multiple timeframes:

I want to search all tweets in February, 2017

```
GET tweets-2017-02*/_search
```

Data Ingestion for Time-based Indices

- Hardcoding the index name in the application is not scalable
- There are three main options to define the index name for data ingestion in time-based indices:
 - date library that calculates the correct index name
 - date math in index names
 - aliases

, 05-Mar-2019 - 0

Data Ingestion for Time-based Indices

- Date math in index names:

- simple to define (e.g. <tweets-{now/d}>)
- less work on a daily basis
- redeploy the application when the unit changes (e.g. from monthly indices to daily indices)

If now = 2018-03-22T11:56:22	
<logstash-{now/d}>	logstash-2018.03.22
<logstash-{now{YYYY.MM}}>	logstash-2018.03
<logstash-{now/w}>	logstash-2018.03.19

Special characters should be URI encoded

```
# GET /<logstash-{now/d}>/_search
GET /%3Clogstash-%7Bnow%2Fd%7D%3E/_search
```



Data Ingestion for Time-based Indices

- Alias:
 - very simple to define (e.g. tweets-write)
 - needs an external tool to update the alias (e.g. every midnight)
 - no need to redeploy your application

```
POST _aliases
{
  "actions": [
    {
      "add": {
        "index": "<tweets-{now/d}>",
        "alias": "tweets_write"
      }
    },
    {
      "remove": {
        "index": "<tweets-{now/d-1d}>",
        "alias": "tweets_write"
      }
    }
  ]
}
```

Managing Time-based Indices

- For optimal ingest rates:
 - Spread the shards of your active index over as many nodes as possible
 - for example, 20 nodes = 20 primary shards on the active index
- For optimal search and low resource usage:
 - shrink the older indices down to the optimal number of shards
 - close indices that are no longer being searched
- Use a hot/warm architecture
 - use hot nodes for indexing and warm nodes for querying



Chapter Review

~05-Mar-2019~

Summary

- If you are expecting your cluster to grow, then it is good to plan for that by **overallocating** shards
- A little overallocation is good. A “kagillion” shards is not
- You can attempt to calculate the **maximum shard size** for your particular use case by pushing the limits of your index using one primary shard on a one-node cluster
- You can scale the query workload of your cluster by adding more nodes and **increasing the number of replicas** of your indices
- You can similarly provide scaling by distributing your documents across **multiple indices**



Quiz

- True or False:** The number of primary shards of an index is fixed at the time the index is created.
- Is it more optimal to search over 1 index with 10 primary shards, or 10 indices with 1 primary shard each?
- If you have a two node cluster, why would you ever create an index with more than two primary shards?
- True or False:** Creating an index with only one primary shard is not a good design.
- Suppose you calculated the max shard size for your dataset to be about 100,000 documents. How many shards should you use for a relatively fixed-size dataset of 900,000 documents?



Lab 6

~05-Mar-2019~0

Capacity Planning

Chapter 7

Document Modeling

· 05-Mar-2019 · 0

- 1 Elasticsearch Internals
- 2 Field Modeling
- 3 Fixing Data
- 4 Advanced Search & Aggregations
- 5 Cluster Management
- 6 Capacity Planning
- 7 Document Modeling
- 8 Monitoring and Alerting
- 9 From Dev to Production



Topics covered:

- The Need for Document Modeling
- Denormalization
- The Need for Nested Types
- Nested Types
- Querying a Nested Type
- The Nested Aggregation
- Parent/Child Relationship
- The has_child Query
- The has_parent Query

05-Mar-2019 - 0



The Need for Document Modeling

.05-Mar-2019.0

It's all about relationships...

- If you come from the SQL world, you will likely need to change your thought process for modeling data for Elasticsearch
 - In SQL, you typically normalize your data
- Search requires different considerations
 - In Elasticsearch, you typically ***denormalize*** your data!
- A flat world has its advantages
 - Indexing and searching is fast
 - No need to join tables or lock rows



...and sometimes relationships matter

- There are times when relationships matter
 - We need to bridge the gap between normal and flat
- Four common techniques for managing relational data in Elasticsearch
 - **Denormalizing:** flatten your data (typically the best solution)
 - **Application-side joins:** run multiple queries on normalized data
 - **Nested objects:** for working with arrays of objects
 - **Parent/child relationships**



Denormalization

~05-Mar-2019

Denormalization

- *Denormalizing* your data refers to “*flattening*” your data
 - storing redundant copies of data in each document, instead of using some type of relationship
 - **_source** is compressed which reduces the disk “waste”
- Denormalization provides the best performance out of Elasticsearch
 - no need to perform expensive joins

~05-Mar-2019



Example of Denormalization

- Suppose you are indexing users and tweets
 - **users** in one index, and **tweets** in another
- When searching for **tweets**, suppose you want to search by the **username** also

users

```
{  
  "username" : "harrison",  
  "userid" : 1,  
  "city" : "Los Angeles",  
  "state" : "California"  
}
```

tweets

```
{  
  "body" : "My favorite movie is Star Wars",  
  "time" : "2017-01-24T02:32:27",  
  "userid" : 1  
}
```



Example of Denormalization

- Duplicate the desired **users** data in the **tweets** document:

tweets

```
PUT tweets/_doc/123
```

```
{  
  "body" : "My favorite movie is Star Wars",  
  "time" : "2017-09-24T02:32:27",  
  "user": {  
    "userid" : 1,  
    "username" : "harrison",  
    "city" : "Los Angeles",  
    "state" : "California"  
  }  
}
```

```
PUT tweets/_doc/456
```

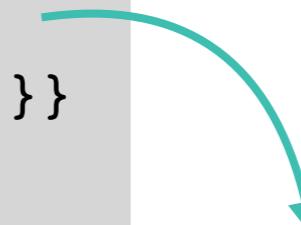
```
{  
  "body" : "Laugh it up, fuzzball.",  
  "time" : "1980-06-20T00:00:00",  
  "user": {  
    "userid" : 1,  
    "username" : "harrison",  
    "city" : "Los Angeles",  
    "state" : "California"  
  }  
}
```



Example of Denormalization

- Now you can search **tweets** and **username** all in a single query:

```
GET tweets/_search
{
  "query": {
    "bool": {
      "must": [
        {"match": {"body": "movie"}},
        {"match": {"user.username": "harrison"}}
      ]
    }
  }
}
```

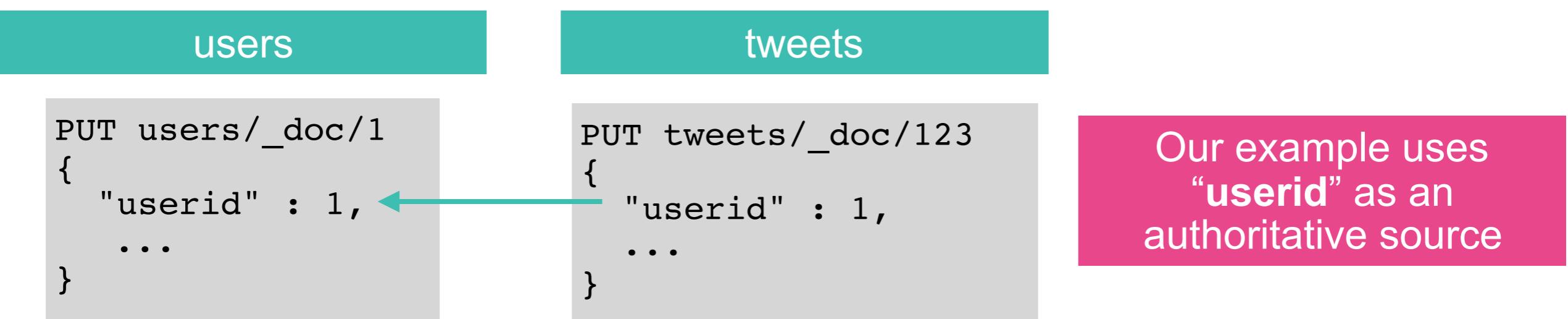


```
"_source": {
  "body": "My favorite movie is Star Wars",
  "time": "2017-09-24T02:32:27",
  "user": {
    "userid": 1,
    "username": "harrison",
    "city": "Los Angeles",
    "state": "California"
  }
}
```



Tips for Denormalizing Data

- Denormalize data to *optimize for reads*
- There is no mechanism to keep the denormalized data consistent with the original document
 - avoid denormalizing data that is likely to change frequently
- When denormalizing data that changes, you should ensure that there is 1 and only 1 **authoritative source** for the data
 - If you do denormalize data that might change, it can help to also denormalize a field that does not change, like an `_id`



The Need for Nested Types

.05-Mar-2019 - 0

Inner Objects

- Suppose we have a scenario where denormalization is not an option (or would be difficult to implement)
- For example, suppose we are indexing metadata of image files, and users can tag an image with any tags they want
 - To maintain strict mappings, we use a key/value pair design:

```
PUT photos/_doc/1
{
  "filename": "img1.jpg",
  "tags": [
    {"key": "event", "value": "Christmas"},  

    {"key": "folder", "value": "December2017"}
  ]
}
```

A photo can be tagged with any key/value pair...

```
PUT photos/_doc/2
{
  "filename": "img2.jpg",
  "tags": [
    {"key": "event", "value": "vacation"},  

    {"key": "holiday", "value": "Christmas"}
  ]
}
```

...and can have any number of tags, so we use an array



Searching inner objects...

- There is an interesting (and confusing) scenario that can arise with arrays of JSON inner objects
 - Which of the two documents on the previous slide is a hit for the following query?

```
GET photos/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "tags.key": "event"
          }
        },
        {
          "match": {
            "tags.value": "Christmas"
          }
        }
      ]
    }
  }
}
```



...can have surprising results

- Both documents are a hit?

```
GET photos/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "tags.key": "event"
          }
        },
        {
          "match": {
            "tags.value": "Christmas"
          }
        }
      ]
    }
  }
}
```



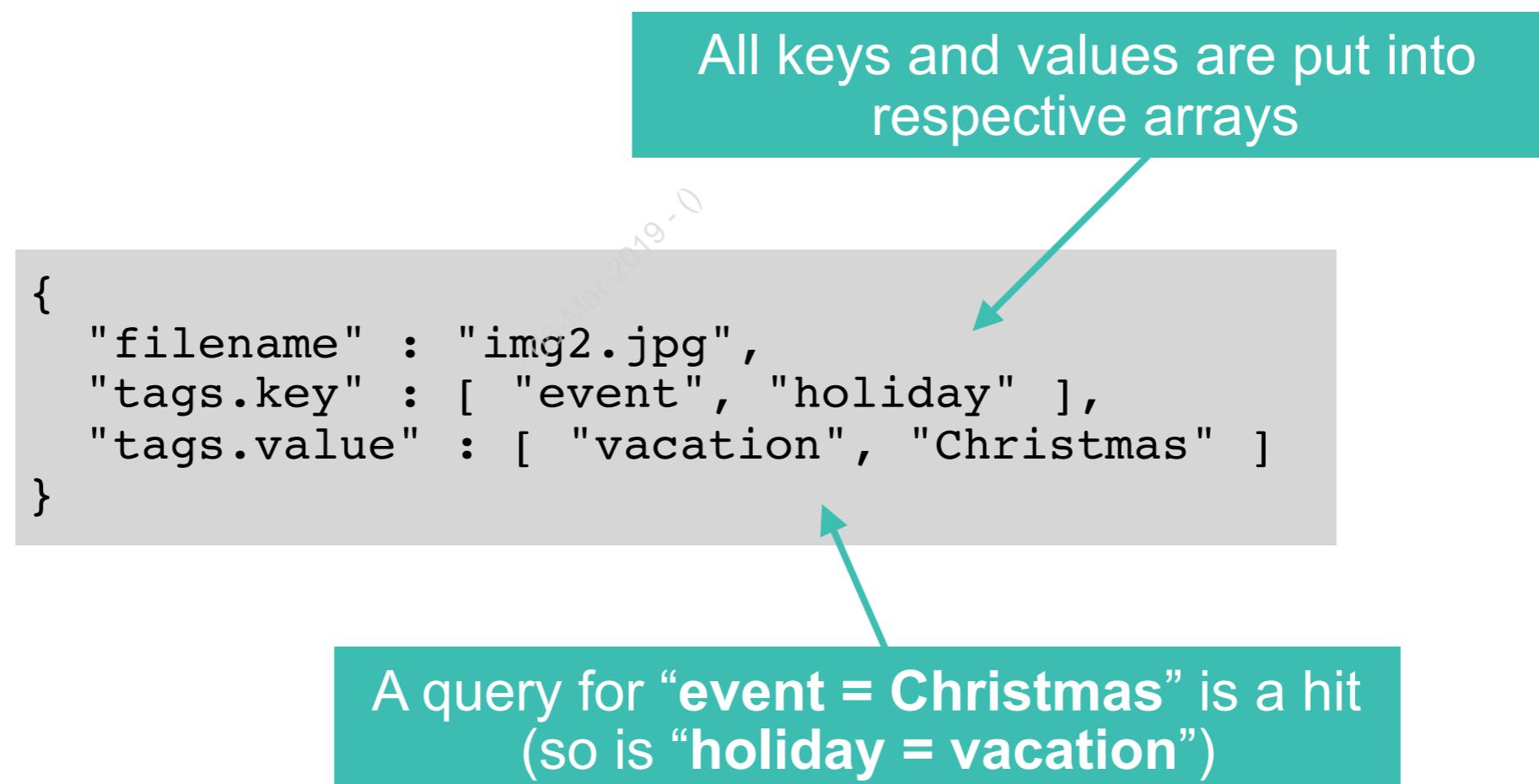
```
"filename": "img1.jpg",
"tags": [
  {"key": "event", "value": "Christmas"},
  {"key": "folder", "value": "December2017"}]
```

```
"filename": "img2.jpg",
"tags": [
  {"key": "event", "value": "vacation"},
  {"key": "holiday", "value": "Christmas"}]
```



Why the confusing results?

- The “key” and “value” fields are JSON inner objects of the “tags” array field
 - When the JSON object got flattened in the array, we lost the relationship between “key” and “value”



Nested Types

~05-Mar-2018

Nested Data Type

- The **nested** type allows arrays of objects to be indexed and queried independently of each other
 - Use **nested** if you need to maintain the relationship of each object in the array
- The mapping syntax looks like:

```
"mappings": {  
  "_doc": {  
    "properties": {  
      "outer_object        "type": "nested",  
        "properties": {  
          "inner_field          ...  
        }  
      }  
    },  
  },  
},
```



The Photo Metadata Example

- Define **tags** as **nested** to maintain the relationship between the **key/value** fields within **tags**:

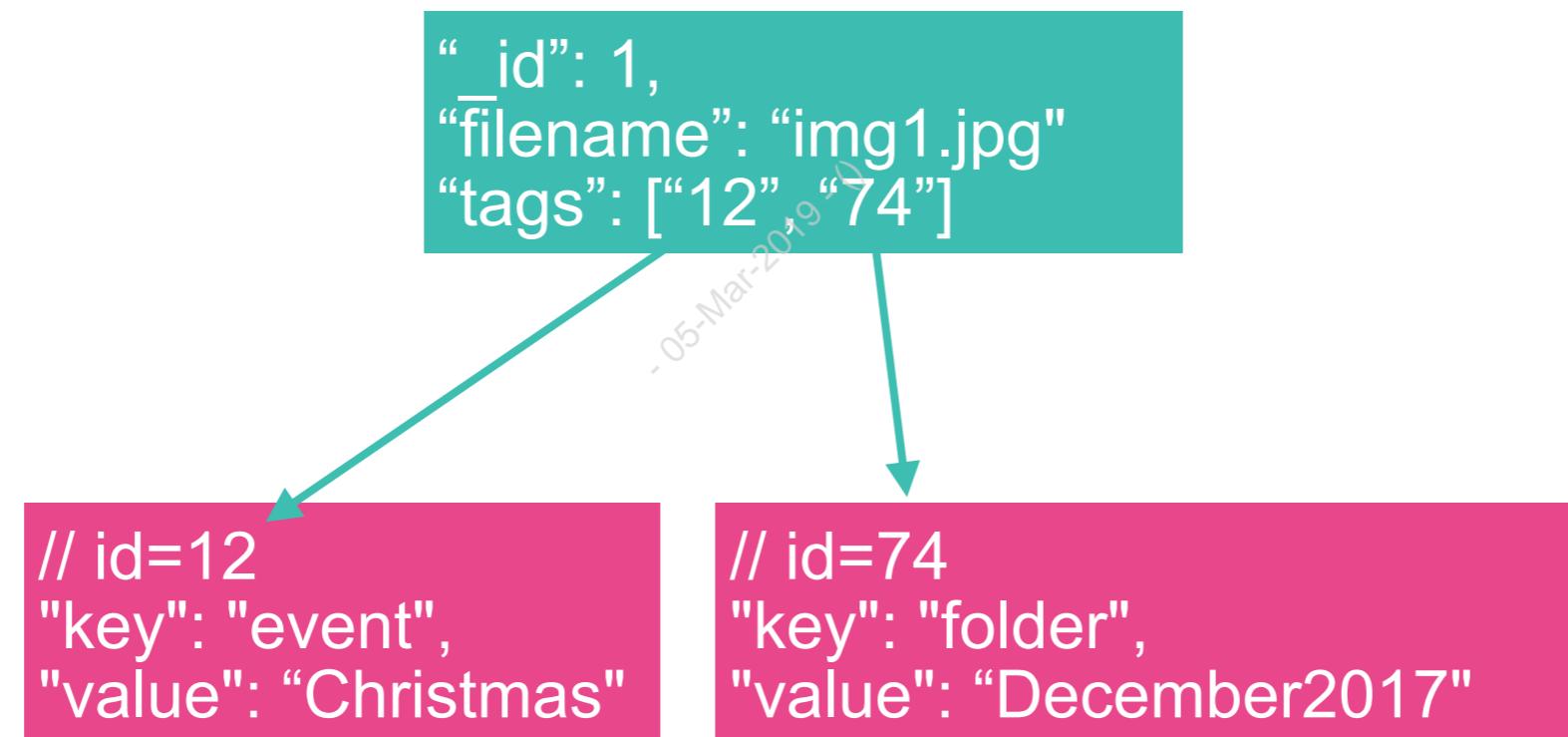
```
PUT photos
{
  "mappings": {
    "_doc": {
      "properties": {
        "filename": {
          "type": "keyword"
        },
        "tags": {
          "type": "nested",
          "properties": {
            "key": {
              "type": "keyword"
            },
            "value": {
              "type": "text"
            }
          }
        }
      }
    }
  }
}
```

Making an array field **nested** maintains the relationship between its nested fields



Nested Objects are Stored Separately

- Internally, nested documents are stored as **separate Lucene documents** that are *joined at query time*
 - Joins always come at a performance cost, so use nested types only when denormalization is not an option



Querying a Nested Type

05 Mar 2019 - 0

Querying a nested Type

- To query a **nested** data type, you have to use the “**nested**” query:

```
GET photos/_search
{
  "query": {
    "nested": {
      "path": "tags",
      "query": {
        "bool": {
          "must": [
            {"match": {"tags.key": "event"}},
            {"match": {"tags.value": "Christmas"}}
          ]
        }
      }
    }
  }
}
```

Specify the “**path**” to the nested object

“filename”: “img1.jpg”,
“tags”: [
 {“key”: “**event**”, “value”: “**Christmas**”},
 {“key”: “folder”, “value”: “December2017”}
]



Finding the Cause of a Hit

- Suppose we search for “tags” that have a “value” named “Christmas”
 - both **photos** are a hit, but the response does not reveal specifically which “key” has the value “Christmas”

```
GET photos/_search
{
  "query": {
    "nested": {
      "path": "tags",
      "query": {
        "match": {
          "tags.value": "Christmas"
        }
      }
    }
  }
}
```



“img1.jpg”

“img2.jpg”



The inner_hits Query

- The **inner_hits** query reveals which “key” generated the hit for the “Christmas”
 - returned in a separate “**inner_hits**” clause in the response
 - **inner_hits** is added to your **nested** query:

```
GET photos/_search
{
  "query": {
    "nested": {
      "path": "tags",
      "query": {
        "match": {
          "tags.value": "Christmas"
        }
      },
      "inner_hits": {}
    }
  }
}
```

You can add “**inner_hits**”
to a “**nested**” query



“img1.jpg”
“key”: “event”
“value”: “Christmas”

“img2.jpg”
“key”: “holiday”
“value”: “Christmas”



The Nested Aggregation

.05-Mar-2019 - 0

The nested Aggregation

- The **nested** bucket aggregation puts nested objects into a bucket
 - Useful for performing sub-aggregations
- The following simple example puts all **tags** into a bucket:

```
GET photos/_search
{
  "size": 0,
  "aggs": {
    "my_tags": {
      "nested": {
        "path": "tags"
      }
    }
  }
}
```



```
"aggregations": {
  "my_tags": {
    "doc_count": 4
  }
}
```



Example of a nested Aggregation

- Suppose we want to put all tags with the same value into the same bucket:

```
GET photos/_search
{
  "size": 0,
  "aggs": {
    "my_tags": {
      "nested": {
        "path": "tags"
      },
      "aggs": {
        "tag_terms": {
          "terms": {
            "field": "tags.key",
            "size": 10
          }
        }
      }
    }
  }
}
```



```
"buckets": [
  {
    "key": "event",
    "doc_count": 2
  },
  {
    "key": "folder",
    "doc_count": 1
  },
  {
    "key": "holiday",
    "doc_count": 1
  }
]
```



Parent/Child Relationship

-05-Mar-2019-

The Need for Parent/Child Relationships

- Updating a nested object requires:
 - a complete reindexing of the root object, AND
 - a complete reindexing of all its nested objects
- Using a ***join datatype***, you can completely separate two objects while maintaining their relationship
 - the parent and children are completely separate documents
 - the parent can be updated without reindexing the children
 - children can be added/changed/deleted without affecting the parent or any other children
- Configured in your mappings using the ***join datatype***



Parent/Child and Shards

- Behind the scenes, the *parent and all its children must live on the same shard*
 - This makes the query-time join faster
- Remember how documents are routed using the hash function?
 - They will not work for a child document - it must get routed to the same shard as its parent
 - The parent's id is used as the routing value for the child document
- Therefore, every time you refer to a child document, you must specify its parent's id
 - We will see how this is implemented next...

Defining a Parent/Child Relationship

- Let's go through the steps of defining a parent/child relationship
 - Define the mapping
 - Index some parent documents
 - Index some child documents
 - Query the documents

~05-Mar-2019~0

1. Define the mapping

- The parent and child relationship is defined using a field of type **join**
 - The following “**my_join_relation**” field defines “**company**” as a parent of “**employee**”:

```
PUT companies
{
  "mappings": {
    "_doc": {
      "properties": {
        "my_join_relation": {
          "type": "join",
          "relations": {
            "company": "employee"
          }
        },
        ...
      }
    }
  }
}
```

“**company**” is a parent of “**employee**”

A mapping can only define one **join** field, but that **join** field can define multiple relations



2. Index some parent documents

- When indexing a parent, specify its relation
 - In our join field, we used “company” as the parent relation name:

```
PUT companies/_doc/c1
{
  "company_name" : "Stark Enterprises",
  "my_join_relation": {
    "name": "company"
  }
}
PUT companies/_doc/c2
{
  "company_name" : "NBC Universal",
  "my_join_relation": {
    "name": "company"
  }
}
```

name of the **join** field

This document is a
“company” document



3. Index some child documents

- Specify the relation name of the **join** field
- Use the “**routing**” parameter to ensure the child document is indexed on the same shard as its parent

```
PUT companies/_doc/emp1?routing=c1
{
  "first_name" : "Tony",
  "last_name" : "Stark",
  "my_join_relation": {
    "name": "employee",
    "parent": "c1"
  }
}
```

A child document has to be on the same shard as its parent

name of the join field

This document is an “employee” document

This employee works for “Stark Enterprises”



4. Query the documents

- Notice the parent and child documents are in the same index:

```
GET companies/_search
```



~05-Mar-2019~0

```
{  
  "took": 1,  
  "timed_out": false,  
  "_shards": {  
    "total": 5,  
    "successful": 5,  
    "failed": 0  
  },  
  "hits": {  
    "total": 3,  
    "max_score": 1,  
    "hits": [  
      ...  
    ]  
  }  
}
```

- Typical parent/child queries involve **has_child** and **has_parent**...



The has_child Query

~05-Mar-2016 ~

The has_child Query

- The **has_child** query is a filter that accepts a query and the child type to run against
 - It results in parent documents that have child docs matching the query
 - Parent and child documents are routed to the same shard, so joining them when searching will be in-memory and efficient
- The syntax looks like:

```
GET my_index/_search
{
  "query": {
    "has_child": {
      "type": "relation_name",
      "query": {}
    }
  }
}
```

the child relation name



Example of has_child

I want all companies who have an employee named "Stark"

```
GET companies/_search
{
  "query": {
    "has_child": {
      "type": "employee",
      "query": {
        "match": {
          "last_name": "Stark"
        }
      }
    }
  }
}
```

```
"hits": [
  {
    "_index": "companies",
    "_type": "_doc",
    "_id": "c1",
    "_score": 1,
    "_source": {
      "company_name": "Stark Enterprises",
      "my_join_relation": {
        "name": "company"
      }
    }
  }
]
```



The inner_hits Query

- Notice in the previous query that “Stark Enterprises” has an employee with the last name “Stark”, but we do not know which employee caused the hit
 - Use the **inner_hits** query to get the relevant children from the **has_child** query:

```
GET companies/_search
{
  "query": {
    "has_child": {
      "type": "employee",
      "query": {
        "match": {
          "last_name": "Stark"
        }
      },
      "inner_hits" : {}
    }
  }
}
```



```
...
  "inner_hits": {
    "employee": {
      "hits": {
        "total": 1,
        "max_score": 0.6931472,
        "hits": [
          {
            "_type": "_doc",
            "_id": "emp1",
            "_score": 0.6931472,
            "_routing": "c1",
            "_source": {
              "first_name": "Tony",
              "last_name": "Stark",
              "my_join_relation": {
                "name": "employee",
                "parent": "c1"
              }
            }
          }
        ]
      }
    }
  }
}
```



The has_parent Query

~05-Mar-2018~

The has_parent Query

- The **has_parent** query accepts a parent and
 - Returns child documents which associated parents have matched
- The syntax looks like:

```
GET my_index/_search
{
  "query": {
    "has_parent": {
      "parent_type": "relation_name",
      "query": {}
    }
  }
}
```

the parent relation name

Example of has_parent

*I want all employees
that work for "NBC"*

```
GET companies/_search
{
  "query": {
    "has_parent": {
      "parent_type": "company",
      "query": {
        "match": {
          "company_name": "NBC"
        }
      }
    }
  }
}
```

```
"hits": [
  {
    "_index": "companies",
    "_type": "_doc",
    "_id": "emp3",
    "_score": 1,
    "_routing": "c2",
    "_source": {
      "first_name": "Tony",
      "last_name": "Potts",
      "my_join_relation": {
        "name": "employee",
        "parent": "c2"
      }
    }
  }
]
```

Accessing a Child Document

- The Document APIs require the **routing** property when working with child documents
 - The APIs need to know the routing id of the parent to find the child

```
GET companies/_doc/emp1
```

```
{  
  "_index": "companies",  
  "_type": "_doc",  
  "_id": "emp1",  
  "found": false  
}
```

```
GET companies/_doc/emp1?routing=c1
```

```
{  
  "_index": "companies",  
  "_type": "_doc",  
  "_id": "emp1",  
  "_version": 1,  
  "_routing": "c1",  
  "found": true,  
  "_source": {  
    "first_name": "Tony",  
    "last_name": "Stark",  
    "my_join_relation": {  
      "name": "employee",  
      "parent": "c1"  
    }  
  }  
}
```



Updating Child Documents

- One of the key benefits of a parent/child relationship is the ability to modify a child object independent of the parent
 - For example, changing an employee has no effect on its parent (or any of its siblings either)

```
POST companies/_doc/emp1/_update?routing=c1
{
  "doc" : {
    "first_name" : "Anthony"
  }
}
```

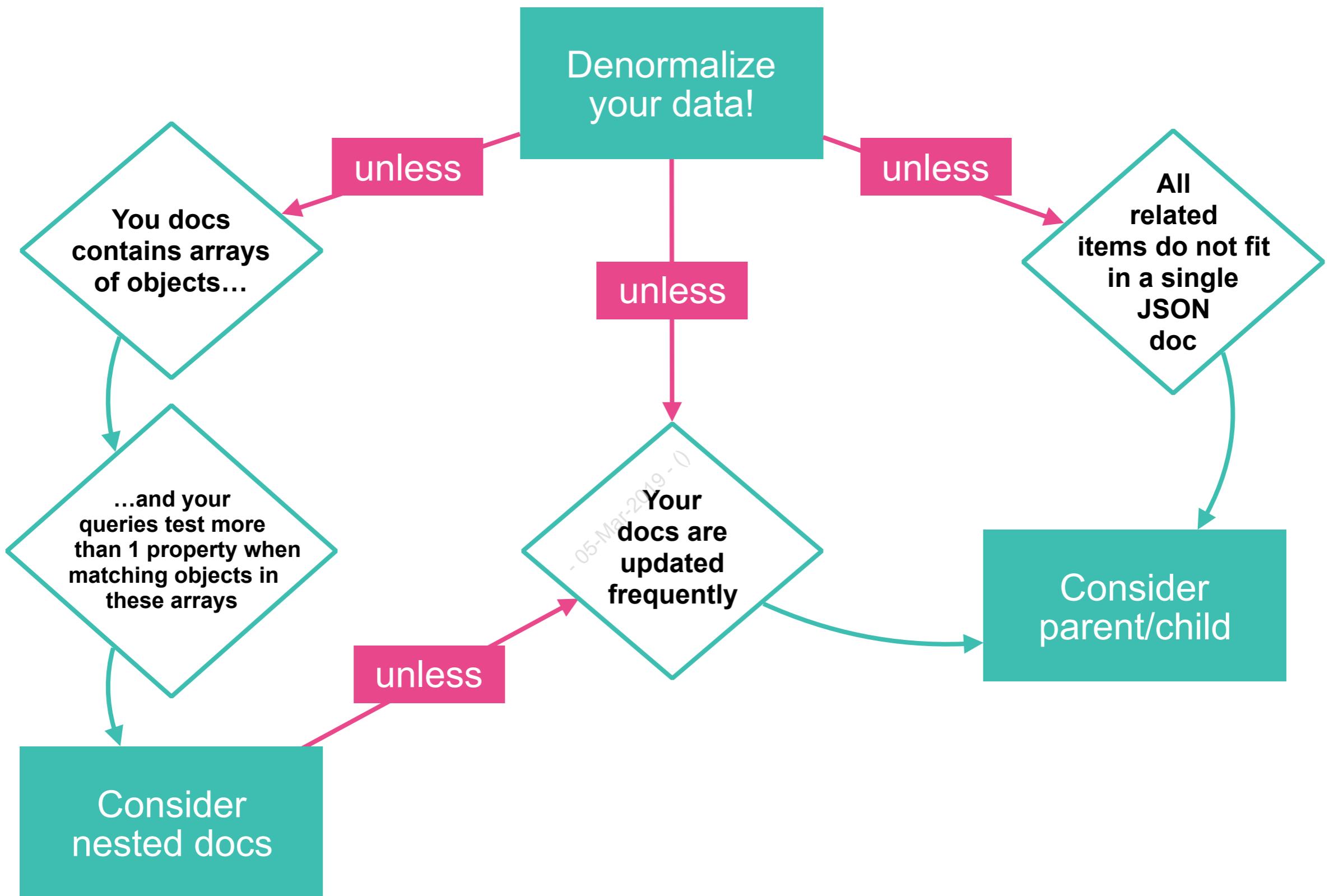
Change “Tony” to “Anthony”



Choosing a Technique

~05-Mar-2010

Choosing a Technique for Relationships



Kibana Considerations

- Kibana currently *has very limited support* for nested types or parent/child relationships
 - important to consider this limitation if you are using your indices for dashboards and visualizations in Kibana
- Kibana may better support these relationships in the future
 - but for now, you will have to decide which is more important: using nested or parent/child relationships, or being able to visualize your data in Kibana



Chapter Review

~05-Mar-2019~

Summary

- ***Denormalizing*** your data refers to “flattening” your data, and typically provides the best performance in terms of how your data is modeled
- The ***nested*** type allows arrays of objects to be indexed and queried independently of each other
- Updating a nested object requires a complete reindexing of the root object AND all other of its nested objects
- Using a ***parent/child data type***, you can completely separate two objects while maintaining their relationship
- One of the key benefits of a parent/child relationship is the ability to modify a child object independently of the parent
- When modeling your documents, prefer denormalization whenever possible



Quiz

- True or False:** Updating a **nested** inner object causes the root object and all other nested objects to be reindexed.
- True or False:** Deleting a child object causes the parent object and all other siblings to be reindexed.
- Why not just use a parent/child relationship all the time (as opposed to **nested** types) when dealing with relational objects?
- True or False:** Child objects must be routed to the same shard as its parent object.
- True or False:** Deleting a parent object causes all child objects to be deleted.
- True or False:** You can index a child object whose parent does not exist.



Lab 7

Document Modeling

~05-Mar-2019~0

Chapter 8

Monitoring and Alerting

05-Mar-2019 - 0

- 1 Elasticsearch Internals
- 2 Field Modeling
- 3 Fixing Data
- 4 Advanced Search & Aggregations
- 5 Cluster Management
- 6 Capacity Planning
- 7 Document Modeling
- 8 Monitoring and Alerting**
- 9 From Dev to Production



Topics covered:

- Monitoring Options
- The Stats APIs
- Task Monitoring
- The cat API
- Diagnosing Performance Issues
- The Elastic Monitoring Component
- The Monitoring UI
- Alerting

Monitoring Options

~05-Mar-2019

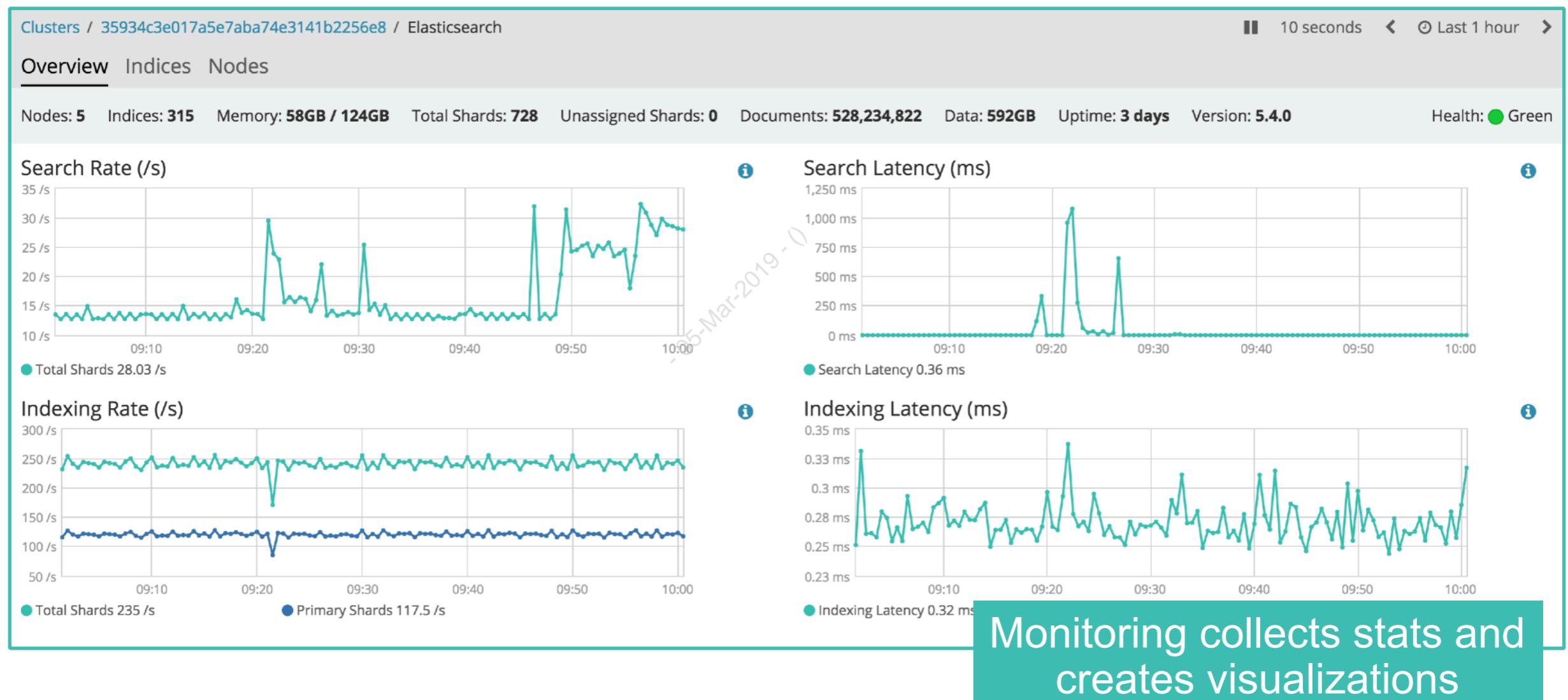
Monitoring Options

- Elasticsearch has several APIs for monitoring:
 - **Node Stats:** `_nodes/stats`
 - **Cluster Stats:** `_cluster/stats`
 - **Index Stats:** `my_index/_stats`
 - **Pending Cluster Tasks API:** `_cluster/pending_tasks`
- The above APIs return JSON objects (no surprise)
 - **cat API:** a human-readable alternative to the JSON APIs above
 - similar results, just formatted differently



Monitoring (the Component)

- While useful, the stats APIs are *point-in-time*
 - That is where *Monitoring* comes in to play (not the verb “monitoring”, but the Elastic component called “Monitoring”)



The Stats APIs

~05-Mar-2016~

Cluster Stats

- The **Cluster Stats API** provides stats at the cluster level:

GET `_cluster/stats`

Only a very small subset of
the large amount of stats

```
{  
  "_nodes": {  
    "total": 2,  
    "successful": 2,  
    "failed": 0  
  },  
  "cluster_name": "my_cluster",  
  "timestamp": 1486701112018,  
  "status": "red",  
  "indices": {  
    "count": 26,  
    "shards": {  
      "total": 162,  
      "primaries": 87,  
      "replication": 0.8620689655172413,  
      "index": {  
        "shards": {  
          "min": 2,  
          "max": 10,  
          "avg": 6.230769230769231  
        },  
        "primaries": {  
          "min": 1,  
          "max": 6,  
          "avg": 3.3461538461538463  
        },  
        "replication": {  
          "min": 1,  
          "max": 6,  
          "avg": 3.3461538461538463  
        }  
      }  
    }  
  }  
}
```



Node Stats

- The **Node Stats API** provides stats at the node level:

GET `_nodes/stats`

You can specify a list of nodes also

GET `_nodes/node1/stats`

```
{  
  "_nodes": {  
    "total": 2,  
    "successful": 2,  
    "failed": 0  
  },  
  "cluster_name": "my_cluster",  
  "nodes": {  
    "OmWJPhToQ0iyNfz-Qd9i8g": {  
      "timestamp": 1486701705225,  
      "name": "node1",  
      "transport_address": "192.168.1.6:9300",  
      "host": "192.168.1.6",  
      "ip": "192.168.1.6:9300",  
      "roles": [  
        "master",  
        "data"  
      ],  
      "attributes": {  
        "temp": "hot",  
        "server_size": "small",  
        "zone": "zoneA"  
      },  
      ...  
    }  
  }  
}
```



Indices Stats

- The *Indices Stats API* provides stats at the index level:

GET `my_index/_stats`

You can get the stats from all indices in one request:

GET `_stats`

```
{  
  "_shards": {  
    "total": 10,  
    "successful": 10,  
    "failed": 0  
  },  
  "_all": {  
    "primaries": {  
      "docs": {  
        "count": 2,  
        "deleted": 0  
      },  
      "store": {  
        "size_in_bytes": 7399,  
        "throttle_time_in_millis": 0  
      },  
      "indexing": {  
        "index_total": 0,  
        "index_time_in_millis": 0,  
        "index_current": 0,  
        "index_failed": 0,  
        "delete_total": 0,  
        "delete_time_in_millis": 0,  
        ...  
      }  
    }  
  }  
}
```

Task Monitoring

~05-Mar-2018~Q

Pending Tasks

- The *Pending Tasks API* shows cluster-level changes that have not been executed yet:

```
GET _cluster/pending_tasks
```

```
{  
  "tasks": [  
    {  
      "insert_order": 101,  
      "priority": "URGENT",  
      "source": "create-index [my_index], cause [api]",  
      "time_in_queue_millis": 86,  
      "time_in_queue": "86ms"  
    }  
  ]  
}
```

The response is often empty because cluster-level changes are fast

Task Management API

- The **Task Management API** shows tasks currently executing on the nodes
 - provides a nice view of how busy the cluster is

GET _tasks



You can also use this API to cancel a task

```
{  
  "nodes": {  
    "OmWJPhToQ0iyNfz-Qd9i8g": {  
      "name": "node1",  
      "transport_address": "192.168.1.6:9300",  
      "host": "192.168.1.6",  
      "ip": "192.168.1.6:9300",  
      "roles": [  
        "master",  
        "data"  
      ],  
      "tasks": {  
        "OmWJPhToQ0iyNfz-Qd9i8g:37432": {  
          "node": "OmWJPhToQ0iyNfz-Qd9i8g",  
          "id": 37432,  
          "type": "direct",  
          "action": "cluster:monitor/tasks/lists[n]",  
          "start_time_in_millis": 1486702376488,  
          "running_time_in_nanos": 2157349,  
          "cancellable": false,  
          "parent_task_id": "OmWJPhToQ0iyNfz-Qd9i8g:37431"  
        },  
        "OmWJPhToQ0iyNfz-Qd9i8g:37433": {  
          "node": "OmWJPhToQ0iyNfz-Qd9i8g",  
          "id": 37433,  
          "type": "direct",  
          "action": "cluster:monitor/tasks/lists[n]",  
          "start_time_in_millis": 1486702376488,  
          "running_time_in_nanos": 2157349,  
          "cancellable": false,  
          "parent_task_id": "OmWJPhToQ0iyNfz-Qd9i8g:37431"  
        }  
      }  
    }  
  }  
}
```



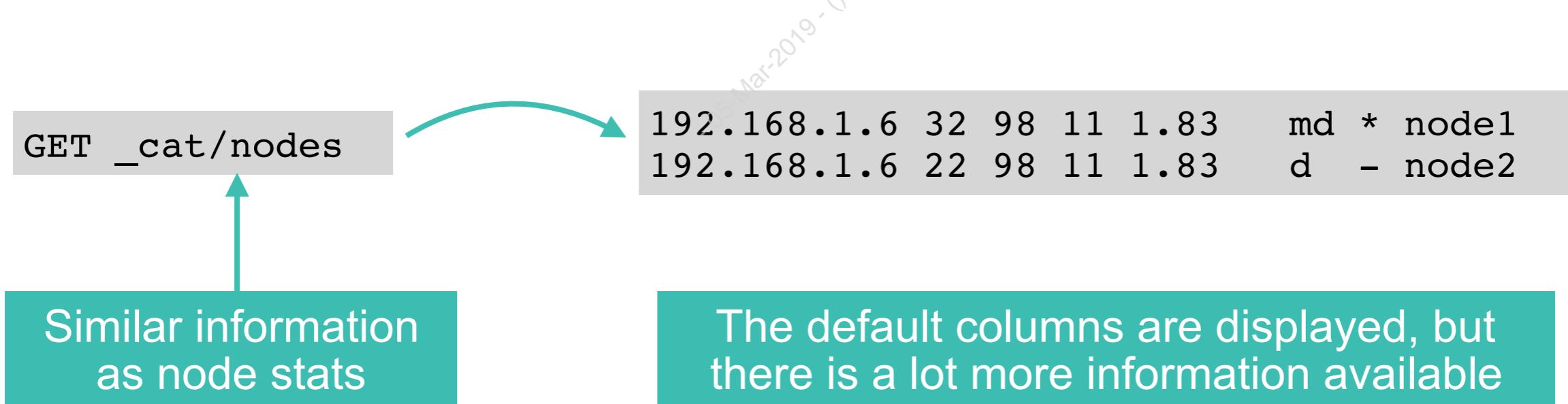
The cat API

~05-Mar-2013 2



The cat API

- You have seen the `_cat` API throughout the labs
 - it is a wrapper around many of the Elasticsearch JSON APIs, including the stats APIs discussed so far in this chapter
 - command-line tool friendly
 - helpful for simple monitoring (e.g. Nagios)



Specifying Columns

- Add the “h” parameter to specify which columns to return
 - use “*” to retrieve all columns
 - add the “v” parameter to display the column names in the response

```
GET _cat/nodes?v&h=name,disk.avail,search.query_total,heap.percent
```



name	disk.avail	search.query_total	heap.percent
node1	144.4gb	3800	43
node2	144.4gb	0	26



Diagnosing Performance Issues

.05 Mar 2019 - 0

Thread Pool Queues

- Many cluster tasks (bulk, index, get, search, etc.) use thread pools to improve performance
 - these thread pools are fronted by queues
 - when a queue is full, 429 status code is returned

GET `_nodes/thread_pool`

```
...  
"bulk": {  
  "type": "fixed",  
  "min": 8,  
  "max": 8,  
  "queue_size": 50  
}  
...
```

GET `_nodes/stats/thread_pool`

```
...  
"bulk": {  
  "threads": 8,  
  "queue": 0,  
  "active": 0,  
  "rejected": 0,  
  "largest": 8,  
  "completed": 177  
}  
...
```

Thread Pool Queues

- The `_cat/thread_pool?v` API provides a nice view of the thread pools:

```
GET _cat/thread_pool?v
```

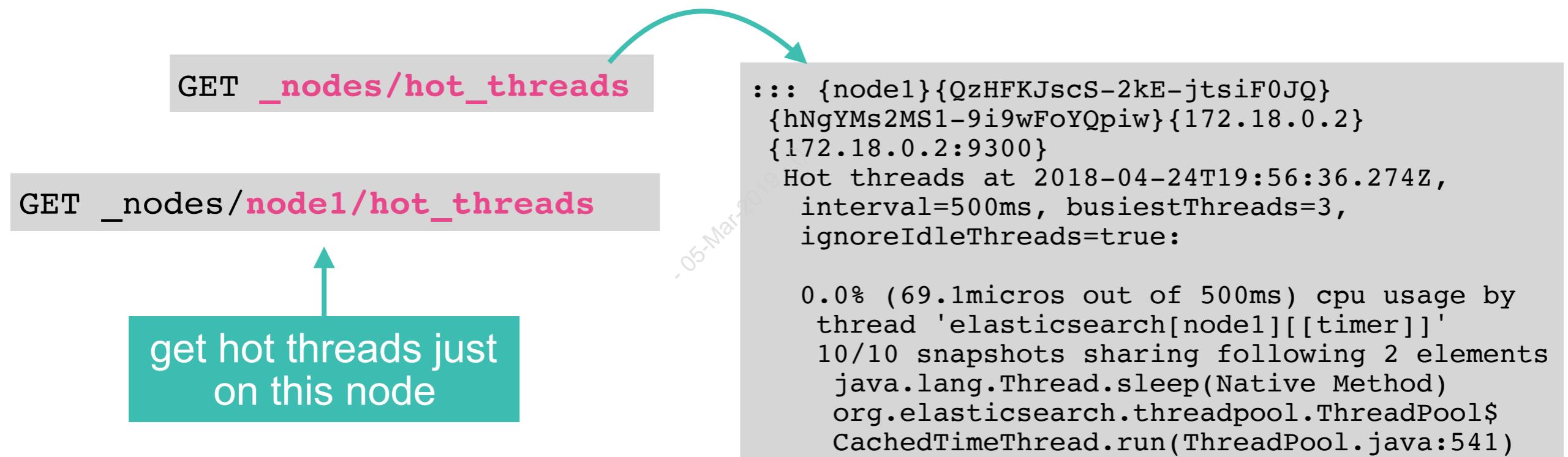
node_name	name	active	queue	rejected
node1	bulk	0	0	0
node1	get	0	0	0
node1	index	0	0	0
node1	management	1	0	0
node2	bulk	0	0	0
node2	get	0	0	0
node2	index	0	0	0
node2	management	1	0	0

- A full queue may be good or bad (“It depends!”)
 - OK if bulk indexing is faster than ES can handle
 - Bad if search queue is full



The hot_threads API

- The *Nodes hot_threads API* allows you get to view the current hot threads on each node
 - invoke it on all nodes or a specific node



The Indexing Slow Log

- The ***Indexing Slow Log*** captures information about long-running index operations into a log file
- Logs ***indexing events*** that take longer than configured thresholds
 - already configured in **log4j2.properties**

```
PUT my_index/_settings
{
  "index.indexing.slowlog" : {
    "threshold.index" : {
      "warn" : "10s",
      "info" : "5s",
      "debug" : "2s",
      "trace" : "0s"
    },
    "level" : "trace",
    "source" : 1000
  }
}
```

Request index slow log level

Current index slow log level

The first 1,000 characters of the document's source will be logged



The Search Slow Log

- The **Search Slow Log** captures information about *long-running searches* (query and fetch phases) into a log file
 - the log file is already configured in **log4j2.properties**, but disabled by default
 - useful, but can be limited since it logs per shard
 - Packetbeat may be a better solution

```
PUT my_index/_settings
{
  "index.search.slowlog": {
    "threshold": {
      "query": {
        "info": "5s"
      },
      "fetch": {
        "info": "800ms"
      }
    },
    "level": "info"
  }
}
```

This example sets level to info and defines thresholds

The Profile API

- Elasticsearch has a powerful **Profile API** which can be used to inspect and analyze your search queries
 - just set “profile” to true in your query:

```
GET crimes/_search
{
  "size": 20,
  "profile": true, ← Enable profiling for this search
  "query": {
    "bool": {
      "filter": {"match": {"incident.description": "handgun"}}
    }
  },
  "aggs": {
    "crimes_with_an_arrest": {
      "filter": {"match": {"arrest_made": "false"}},
      "aggs": {
        "types_of_handgun_crimes": {
          "terms": {"field": "incident.description.keyword"}
        }
      }
    }
  }
}
```



The Profiler Response...

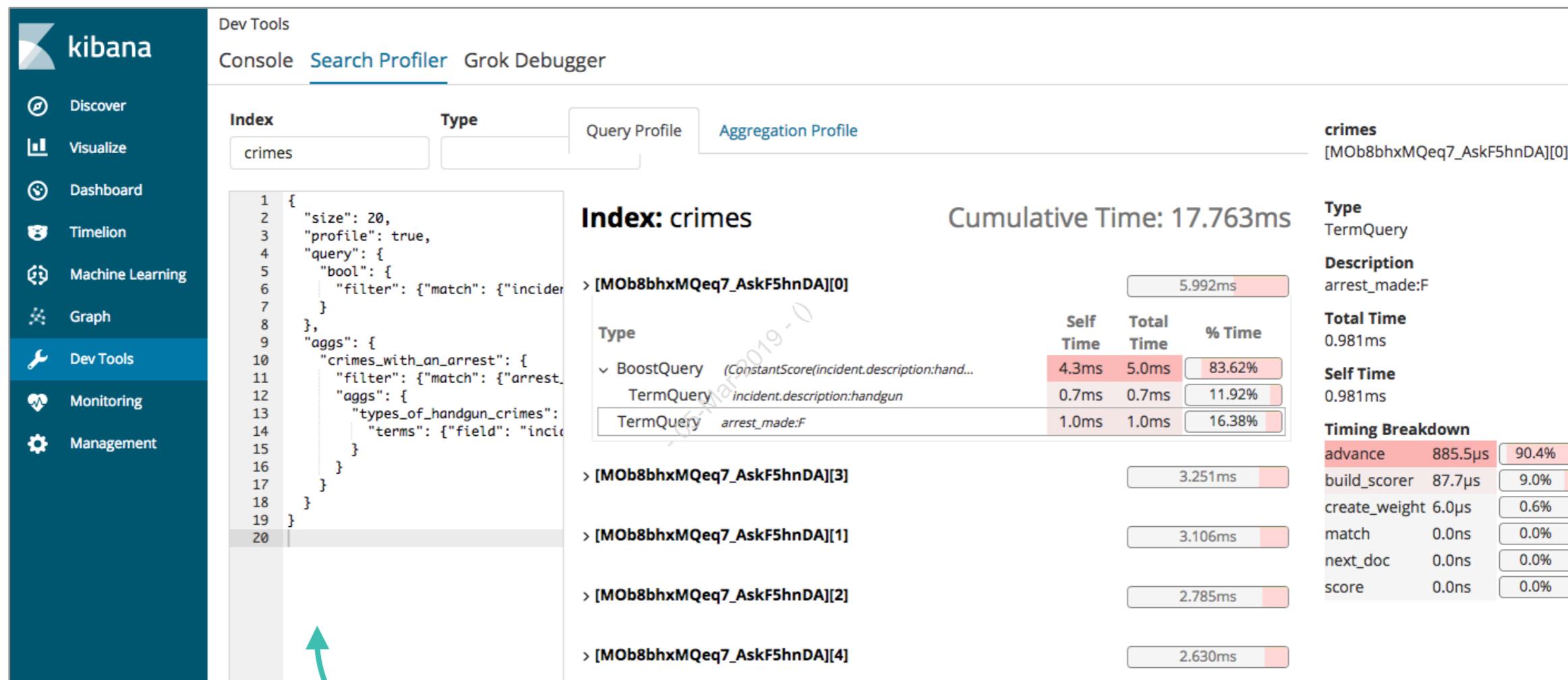
- ...is hard to read (it is a lot of JSON):

```
"profile": {  
  "shards": [  
    {  
      "id": "[M0b8bhxMQeq7_AskF5hnDA][crimes][0]",  
      "searches": [  
        {  
          "query": [  
            {  
              "type": "TermQuery",  
              "description": "arrest_made:F",  
              "time_in_nanos": 867806,  
              "breakdown": {  
                "score": 0,  
                "build_scorer_count": 14,  
                "match_count": 0,  
                "create_weight": 6393,  
                "next_doc": 0,  
                "match": 0,  
                "create_weight_count": 1,  
                "next_doc_count": 0,  
                "score_count": 0,  
                "build_scorer": 100244,  
                "advance": 759117,  
                "advance_count": 2037  
              }  
            },  
            {  
              "type": "BoostQuery",  
              "description":  
                "(ConstantScore(incident.description:handgun))^0.0",  
              "time_in_nanos": 1875651,  
              "breakdown": {  
                "score": 451486,  
                ...  
              }  
            }  
          ]  
        }  
      ]  
    }  
  ]  
}
```



The Search Profiler

- Search Profiler is a tool that transforms the JSON output into a visualization that is easy to navigate:



You can also copy-and-paste the output of a profiled query into this field



The Query Profile Tab

- The query times are shown per shard, along with the generated Lucene query

Query Profile Aggregation Profile

Index: crimes

Cumulative Time: 17.763ms

> [MOb8bhxMQeq7_AskF5hnDA][0]

Type

- BoostQuery (ConstantScore(incident.description:hand...))
- TermQuery incident.description:handgun
- TermQuery arrest_made:F

Self Time	Total Time	% Time
4.3ms	5.0ms	83.62%
0.7ms	0.7ms	11.92%
1.0ms	1.0ms	16.38%

> [MOb8bhxMQeq7_AskF5hnDA][3]

Type

- BoostQuery (ConstantScore(incident.description:hand...))
- TermQuery incident.description:handgun
- TermQuery arrest_made:F

Self Time	Total Time	% Time
1.7ms	2.3ms	71.11%
0.6ms	0.6ms	19.91%
0.9ms	0.9ms	28.89%

crimes
[MOb8bhxMQeq7_AskF5hnDA][1]

Type
TermQuery

Description
incident.description:handgun

Total Time
0.634ms

Self Time
0.634ms

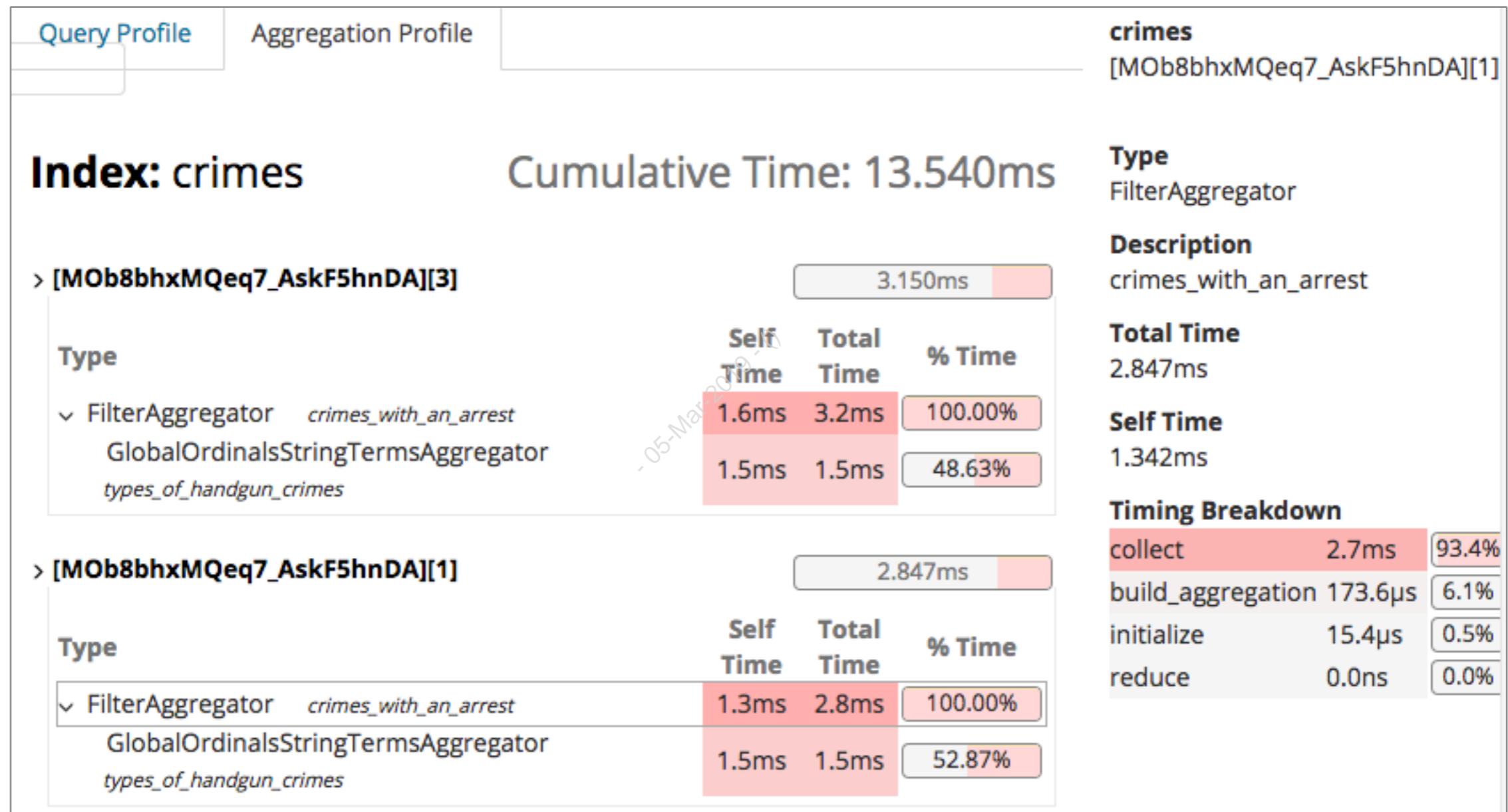
Timing Breakdown

next_doc	577.3 μ s	91.3%
build_scorer	51.1 μ s	8.1%
create_weight	3.7 μ s	0.6%
score	0.0ns	0.0%
match	0.0ns	0.0%
advance	0.0ns	0.0%



The Aggregation Profile Tab

- Shows query times per shard for each aggregation that was executed:



The Elastic Monitoring Component

.05-March-2019 - 0

Elastic Monitoring

- **Monitoring** uses Elasticsearch to monitor Elasticsearch
 - `xpack.monitoring.collection.enabled` defaults to `false`
 - the stats of all the nodes are indexed into Elasticsearch

The screenshot shows the Kibana interface with the left sidebar menu open. The 'Monitoring' option is selected and highlighted in blue. The main content area displays a message about cluster monitoring. At the top right, there are controls for time intervals: '10 seconds', 'Last 1 hour', and arrows for navigating between time periods. The message itself features a blue heart icon with a green line graph inside it. The text reads: 'Monitoring is currently off'. Below this, it says: 'Monitoring provides insight to your hardware performance and load.' A note below states: 'We checked the cluster defaults settings and found that `xpack.monitoring.collection.enabled` is set to `false`.'. At the bottom, a question asks: 'Would you like to turn it on?'. A blue button labeled 'Turn on monitoring' is centered at the bottom of the message box.



Configuring Monitoring

- Here are a few common Monitoring settings
 - which can be configured in `elasticsearch.yml`:

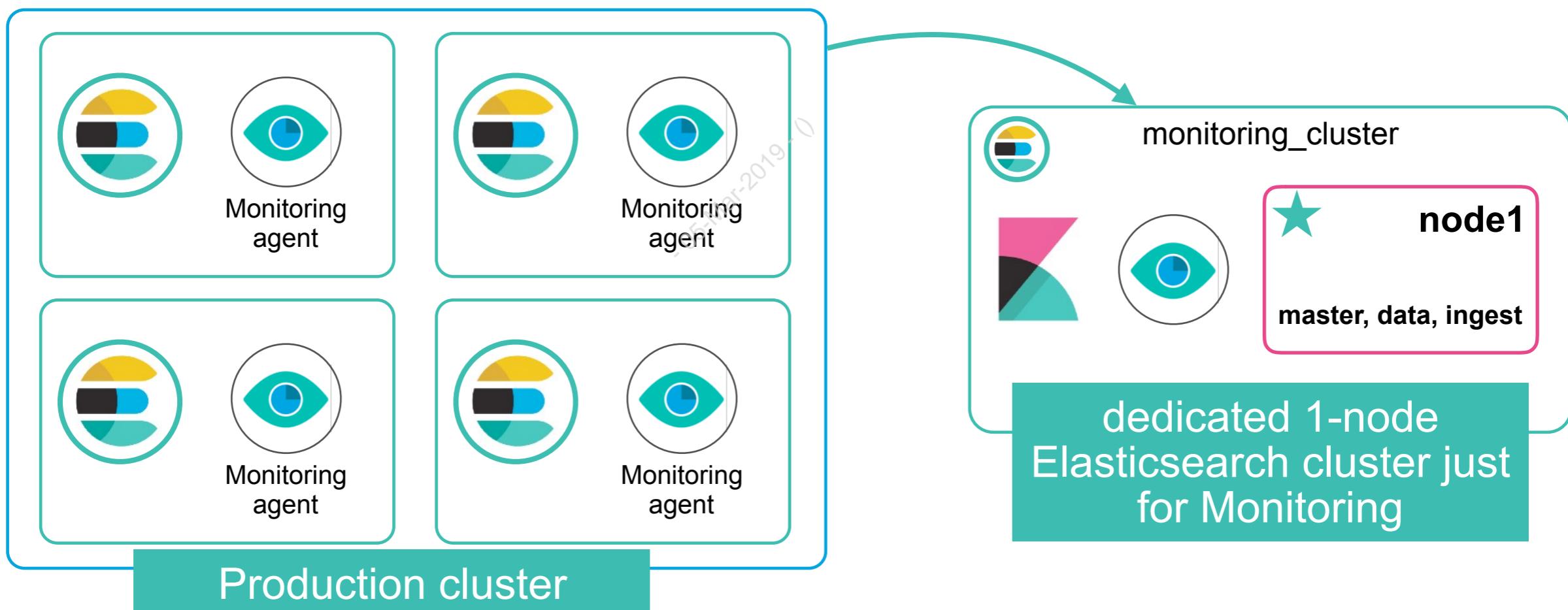
<code>xpack.monitoring.collection.indices</code>	The indices to collect data from. Defaults to all indices, but can be a comma-separated list.
<code>xpack.monitoring.collection.interval</code>	How often data samples are collected. Defaults to 10s
<code>xpack.monitoring.history.duration</code>	How long before indices created by Monitoring are automatically deleted. Defaults to 7d

- The complete list of Monitoring settings is at
 - <https://www.elastic.co/guide/en/elasticsearch/reference/current/monitoring-settings.html>



Dedicated Monitoring Cluster

- Recommend using a *dedicated cluster* for Monitoring
 - reduce the load and storage on your other clusters
 - access to Monitoring even when other clusters are unhealthy
 - separate security levels from Monitoring and production clusters



Configuring Dedicated Monitoring Cluster

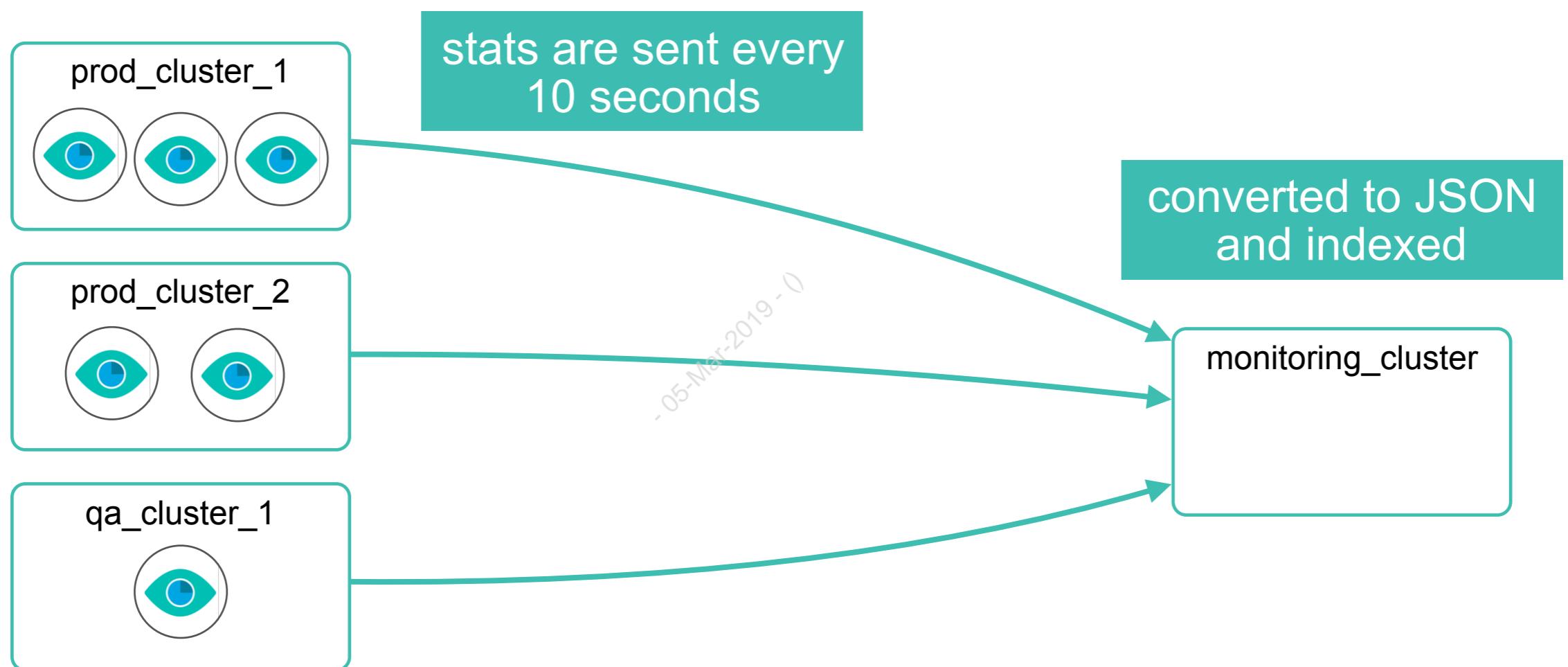
- If Monitoring is on a different cluster than the one being monitored, you need to tell the monitored cluster where to send its stats:
 - if Elastic Security is enabled on the Monitoring cluster, then provide credentials
 - you can also use SSL/TLS (see docs for details:
<https://www.elastic.co/guide/en/elasticsearch/reference/current/monitoring-settings.html#http-exporter-settings>)

configure in `elasticsearch.yml`
of each node

```
xpack.monitoring.exporters:  
  id1:  
    type: http  
    host: ["http://monitoring_cluster:9200"]  
    auth.username: username  
    auth.password: changeme
```

Monitoring Multiple Clusters

- Multiple clusters can be monitored in a single monitoring cluster (Gold/Platinum)

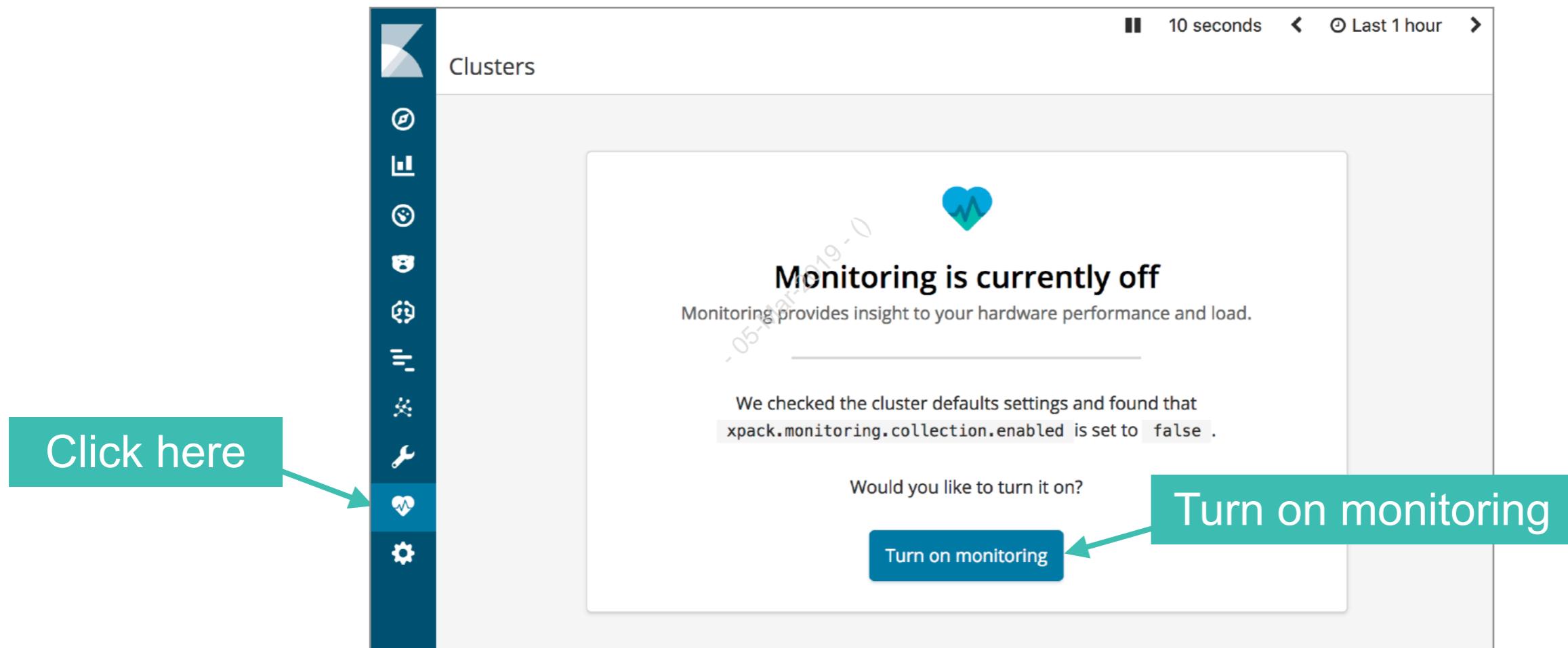


The Monitoring UI

~05-Mar-2019~0

The Monitoring UI

- Open up Kibana
- Click on the **Monitoring** shortcut in the left toolbar:



Clusters Dashboard

- The initial Monitoring page is the “Clusters” dashboard
 - shows all the clusters being monitored
 - notice it is monitoring your Kibana instances as well

The screenshot shows the Elasticsearch Monitoring interface. On the left is a dark sidebar with various icons for different monitoring features. The main area is titled "Clusters" and shows "my_cluster". A green arrow points from the text "shows all the clusters being monitored" in the slide notes to the "my_cluster" section. Below this, a message says "Your Trial license will expire on January 12, 2018." The "Top Cluster Alerts" section lists two items: "Elasticsearch cluster status is yellow. Allocate missing replica shards." (last checked December 13, 2017) and "Configuring TLS will be required to apply a Gold or Platinum license when security is enabled. See documentation for details." (last checked December 13, 2017). A teal callout box on the right states: "Note the free version of Monitoring can only monitor a single cluster". The "Elasticsearch" section provides an overview: Version 6.0.0, Uptime 42 minutes, and 0 jobs. It shows 1 node with 28GB available disk and 29.44% JVM heap usage. There are 14 indices with 4,942 documents, 3MB disk usage, 32 primary shards, and 0 replica shards. The "Health" status is "Yellow". The "Kibana" section also provides an overview: Requests 109, Max. Response Time 683 ms. It shows 1 instance with 340 connections and 11.34% memory usage. The "Health" status is "Green".

Clusters

my_cluster

Your Trial license will expire on [January 12, 2018](#).

Top Cluster Alerts

Elasticsearch cluster status is yellow. [Allocate missing replica shards.](#)
Last checked December 13, 2017 4:24:32 PM (since 41 min ago)

Configuring TLS will be required to apply a Gold or Platinum license when security is enabled. [See documentation for details.](#)
Last checked December 13, 2017 4:25:26 PM (since 47 min ago)

[View all alerts](#)

Elasticsearch Health: Yellow

Overview
Version: 6.0.0
Uptime: 42 minutes
Jobs: 0

Nodes: 1
Disk Available: 28GB / 234GB (12.10%)
JVM Heap: 29.44% (292MB / 991MB)

Indices: 14
Documents: 4,942
Disk Usage: 3MB
Primary Shards: 32
Replica Shards: 0

Kibana Health: Green

Overview
Requests: 109
Max. Response Time: 683 ms

Instances: 1
Connections: 340
Memory Usage: 11.34% (162MB / 1GB)

Note the free version of Monitoring can only monitor a single cluster

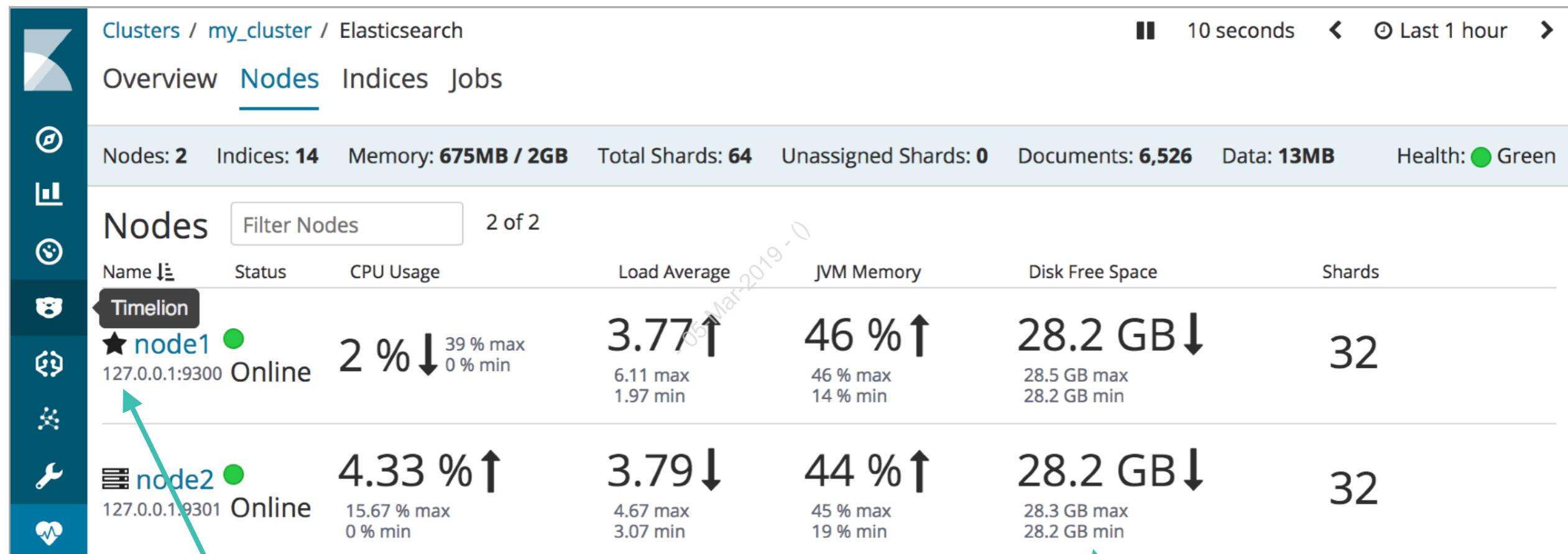


Nodes Dashboard

breadcrumbs for easy navigation

polling interval (and a handy “pause” button)

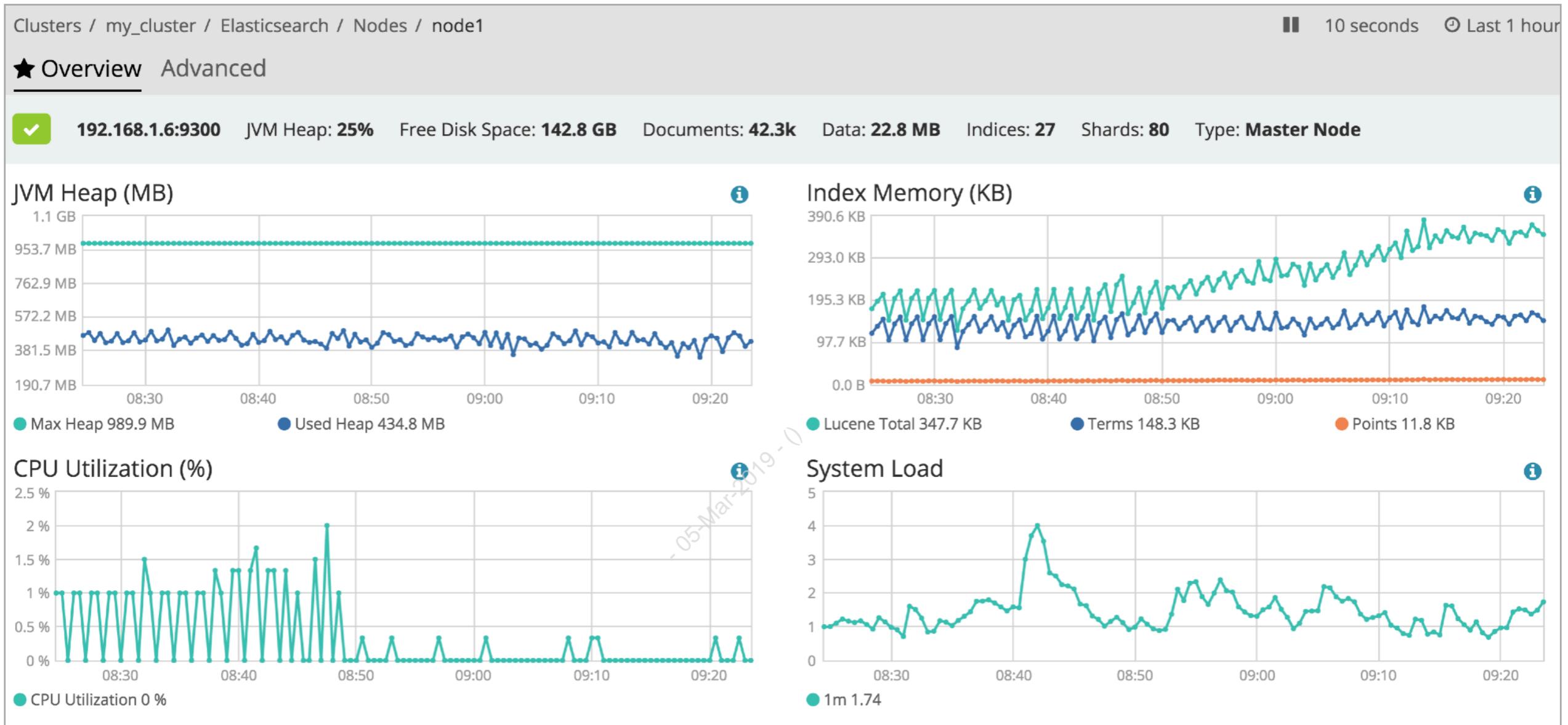
Time interval of what you are currently viewing



master node has the “star”

stats update in real-time

Individual Node Dashboard



the actual page has more details

Indices Dashboard

quick way to find unhealthy indices

Name	Status	Document Count	Data	Index Rate	Search Rate	Unassigned Shards
my_tweets	Yellow	4	28.2 KB	0 /s	0 /s	2
restored-logs-old	Green	2	15.7 KB	0 /s	0 /s	0
logs-old	Green	2	15.7 KB	0 /s	0 /s	0
my_new_index	Green	2	14.8 KB	0 /s	0 /s	0

Alerting

~05-Mar-2019~Q

Configure Alerts

- You should use some type of monitoring tool to ***fire alerts*** for unexpected or extreme situations
 - send an alert, email, page, etc.
 - Nagios is a popular tool
 - third-party tools available
- Or you can use ***Elastic Alerting***
 - <https://www.elastic.co/guide/en/elasticsearch-stack-overview/current/xpack-alerting.html>
 - part of the **Gold** Subscription

Enabling Elastic Alerting

The screenshot shows the Kibana Management interface. On the left is a sidebar with icons for Discover, Visualize, Dashboard, Timelion, APM, Dev Tools, Monitoring, and Management. The main area is titled "Management" and shows "Version: 6.3.1". It features a card for "Elasticsearch" and another for "Index Management" and "License Management", with the latter being circled in red. Below these are cards for "Kibana", "Index Patterns", "Saved Objects", "Reporting", and "Advanced Settings".

The screenshot shows the "Management / Elasticsearch / License Management" page. The sidebar on the left is identical to the one in the previous screenshot. The main content area displays a message: "Your Basic license is active" with a checkmark icon, followed by the text "Your license will never expire." Below this are two boxes: "Update your license" (with the sub-instruction "If you already have a new license, upload it now." and a "Update license" button) and "Start a 30-day trial" (with the sub-instruction "Experience what security, machine learning, and all our other [Platinum features](#) have to offer." and a "Start trial" button). A watermark "05-Mar-2019" is visible across the center of the screen.

Elastic Alerting

- A set of administrative features that enable you to:
 - watch for changes or anomalies in your data and
 - perform the necessary actions in response
- For example, you might want to:
 - open a helpdesk ticket when any servers are running out of free space
 - track network activity to detect malicious activity
 - send immediate notification if nodes leave the cluster

How Watches Work

- A **watch** is constructed from five simple building blocks:
- **Trigger**
 - determines when the watch is executed
- **Input**
 - loads data into the watch payload
- **Condition**
 - controls whether the watch actions are executed
- **Transform**
 - processes the watch payload to prepare it for the watch actions
- **Actions**
 - one or more actions to be executed if condition is true

How Watches Work

```
PUT _xpack/watcher/watch/log_error_watch
{
  "trigger": { "schedule": { "interval": "5m" } },
  "input": {
    "search": {
      "request": {
        "indices": [ "logs*" ],
        "body": {
          "query": { "bool": { "filter": [ { "range": {
              "@timestamp": {
                "gte": "{{ctx.trigger.scheduled_time}}|-5m"
              }
            },
            { "match": { "message": "error" } }
          ] } } }
        }
      }
    }
  },
  "condition": {
    "compare": { "ctx.payload.hits.total": { "gt": 0 } }
  },
  "actions": {
    "log_error": {
      "logging": {
        "text": "Found {{ctx.payload.hits.total}} errors."
      }
    }
  }
}
```

runs every 5 minutes

search for the term **error** in all indices that start with logs

any documents returned?

if true, log this message

How Watches Work

- Watches are stored in Elasticsearch

```
GET .watches/_doc/log_error_watch
```

```
GET .watches/_search
```

Alerting creates some watches
that you can view

- Also, every watch execution is stored in Elasticsearch
 - You can check the watch history to see execution details:

```
GET .watcher-history*/_search
{
  "sort" : [
    { "result.execution_time" : "desc" }
  ]
}
```

retrieves the last ten watch executions (watch records)

Watcher UI

- Enables you to monitor, manage, create and simulate watches
- Available when Alerting is enabled
- If Elastic Security is enabled make sure to create users with Watcher specific roles:
 - **watcher_admin** can perform **all** watcher-related actions
 - **watcher_user** can **view** all existing watches

Watcher UI

The screenshot shows the Kibana Management interface. On the left, a dark sidebar lists various tools: Discover, Visualize, Dashboard, Timelion, Machine Learning, APM, Graph, Dev Tools, Monitoring, Management (which is selected), elastic, Logout, and Collapse. The main area is titled "Management" and displays the version "Version: 6.3.1". It features several sections: "Security" (with "Users" and "Roles" sub-links), "Elasticsearch" (with "Index Management", "License Management", and "Watcher" sub-links, where "Watcher" is circled in red), "Kibana" (with "Index Patterns", "Saved Objects", "Reporting", and "Advanced Settings" sub-links), "Logstash", and "Pipelines".



Watcher UI

Management / Elasticsearch / Watcher

Watches

Create threshold alert

Create advanced watch

delete watches

watch execution

Search...

Delete

1-6 of 6

< >

ID ↑

Name

State

Comment

Last Fired

Last Triggered

WL7EBG2VR...

X-Pack Monit...

✓ OK

a few second...

WL7EBG2VR...

X-Pack Monit...

✓ OK

an hour ago

a few second...

WL7EBG2VR...

X-Pack Monit...

✓ OK

a few second...

WL7EBG2VR...

X-Pack Monit...

✓ OK

a few second...

WL7EBG2VR...

X-Pack Monit...

✓ OK

a minute ago

WL7EBG2VR...

X-Pack Monit...

✓ OK

a minute ago

05-Mar-2019 - 0

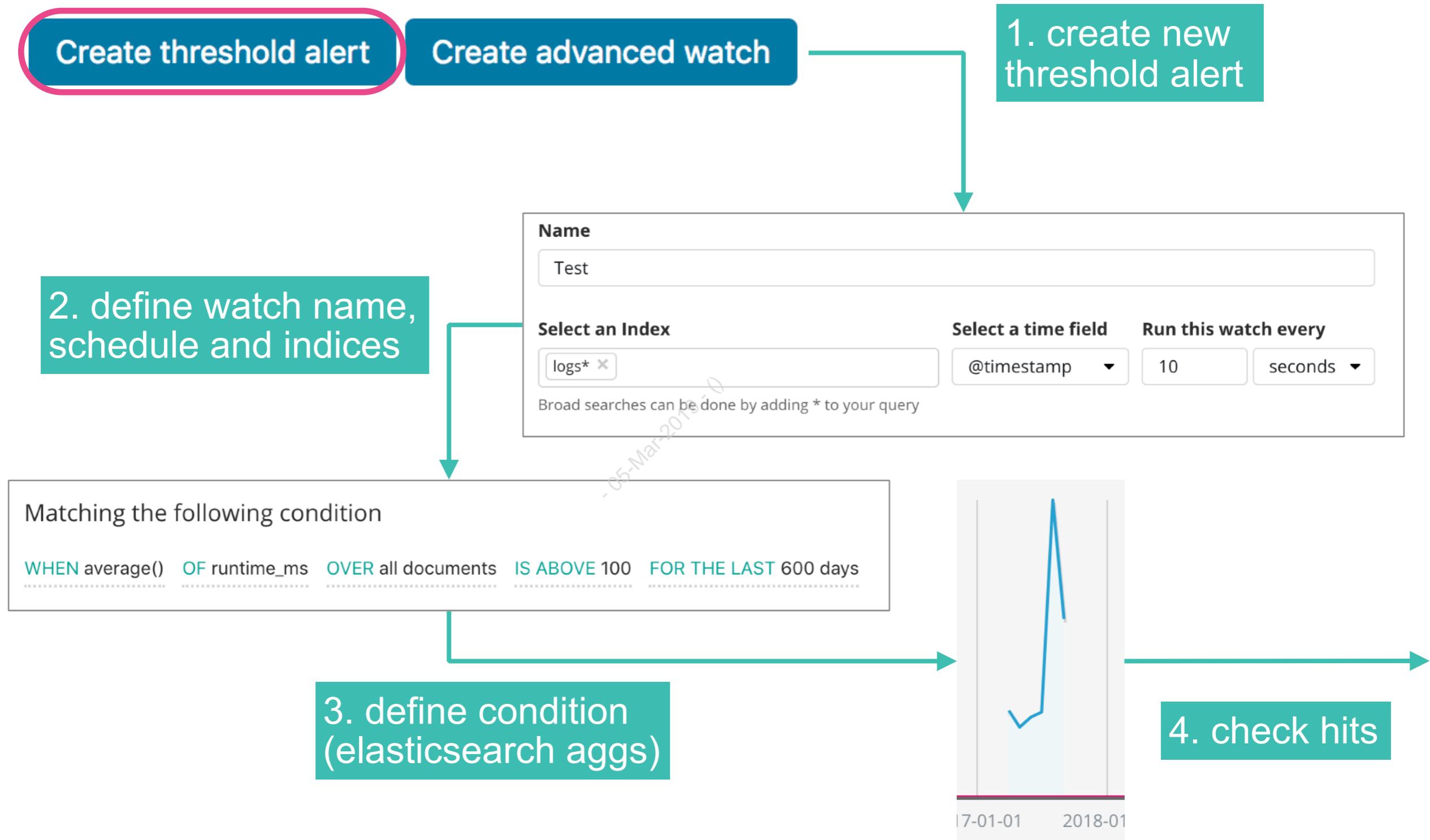
watcher state:
firing, error,
ok, disabled

action execution

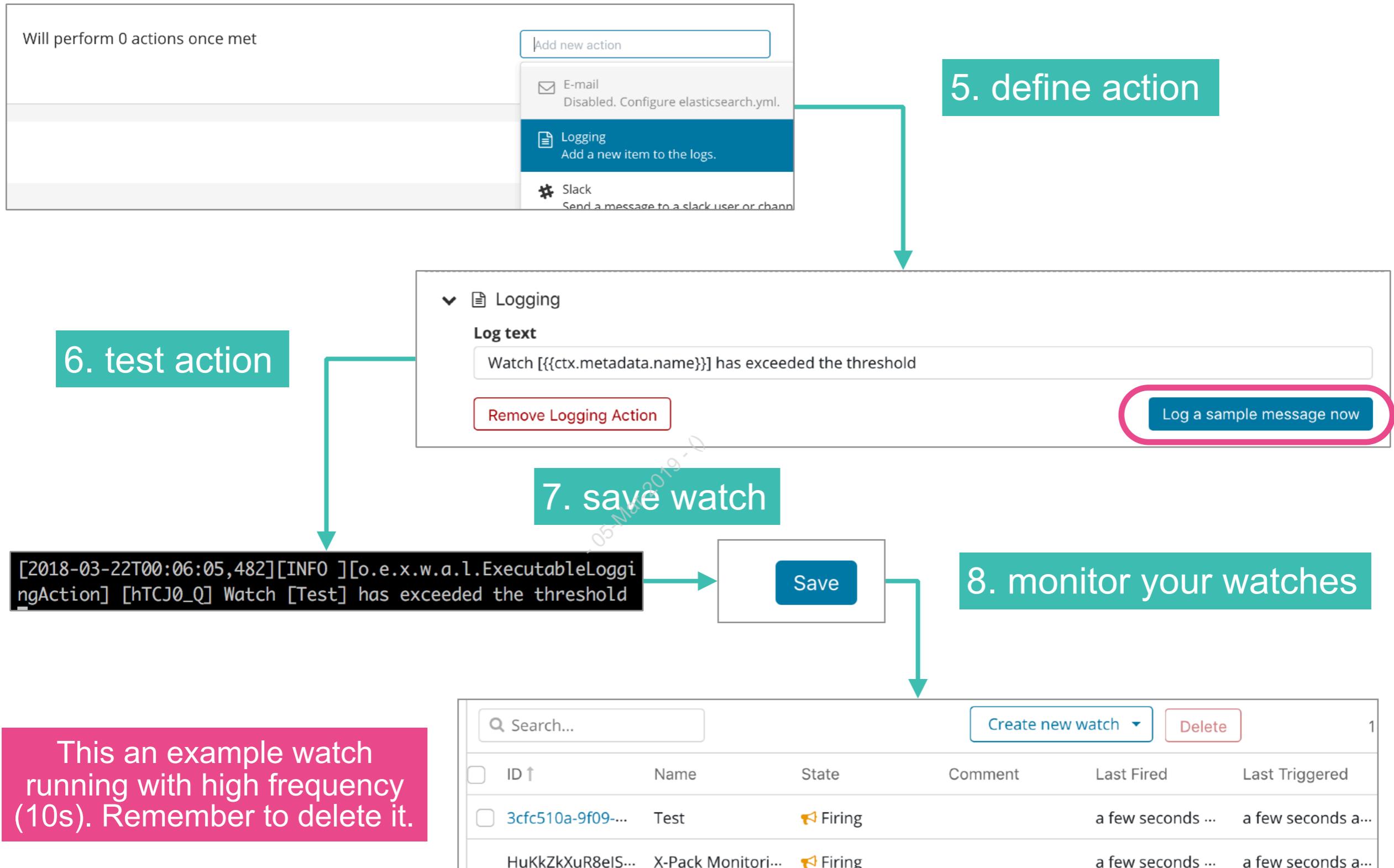
list of existing watches



Creating a Threshold Alert



Creating a Threshold Alert



Chapter Review

~05-Mar-2019~

Summary

- Elasticsearch has **Stats APIs** for retrieving statistics about your cluster, nodes, indices and pending tasks
- The **cat API** provides a human-readable wrapper around the Stats API (and other Elasticsearch APIs)
- Slow logs, thread pools, and hot threads can help you diagnose performance issues.
- The **Elastic Monitoring** component uses Elasticsearch to monitor Elasticsearch
- Best practice is to use a **dedicated cluster** for Monitoring
- Elastic Alerting (Gold license) allows you to create alerts based on different criteria
- Watchers can be configured using JSON or the Watcher UI



Quiz

1. **True or False:** The **Elastic Monitoring** can monitor multiple clusters.
2. What are the benefits of using a dedicated cluster for the **Monitoring** component?
3. The default **Monitoring** collection interval is _____ seconds.
4. How would you check to see if the queue for the index thread pool was full?
5. Name three of the five watch building blocks.
6. **True or False:** Any user can create alerts using the Watcher UI.
7. **True or False:** You can use **Elastic Alerting** to send an email if a node is running out of disk space?



Lab 8

Monitoring and Alerting

DATA-2019-0



Chapter 9

From Dev to Production

05-Mar-2019 - 0

- 1 Elasticsearch Internals
- 2 Field Modeling
- 3 Fixing Data
- 4 Advanced Search & Aggregations
- 5 Cluster Management
- 6 Capacity Planning
- 7 Document Modeling
- 8 Monitoring and Alerting
- 9 From Dev to Production

9

From Dev to Production



Topics covered:

- Disabling Dynamic Indexes
- Development vs. Production Mode
- Best Practices
- JVM Settings
- Common Causes of Poor Query Performance
- Cross Cluster Search
- Overview of Upgrades
- Cluster Restart

~05-Mar-2019~0



Disabling Dynamic Indexes

.05 Mar 2019 - 0

Dynamic Indexes

- An index will ***dynamically*** be created during a document index request if the index does not already exist:

a new **logs** index will be created if it is not defined yet

```
PUT logs/log/1
{
  "level" : "ERROR",
  "message" : "Unable to reach host"
}
```

Disabling Dynamic Indexes

- You can disable this dynamic behavior completely using the dynamic **action.auto_create_index** setting:

```
PUT _cluster/settings
{
  "persistent": {
    "action.auto_create_index" : false
  }
}
```

- Or, you can whitelist certain patterns:

```
PUT _cluster/settings
{
  "persistent": {
    "action.auto_create_index" : ".monitoring-es*,logstash-*"
  }
}
```

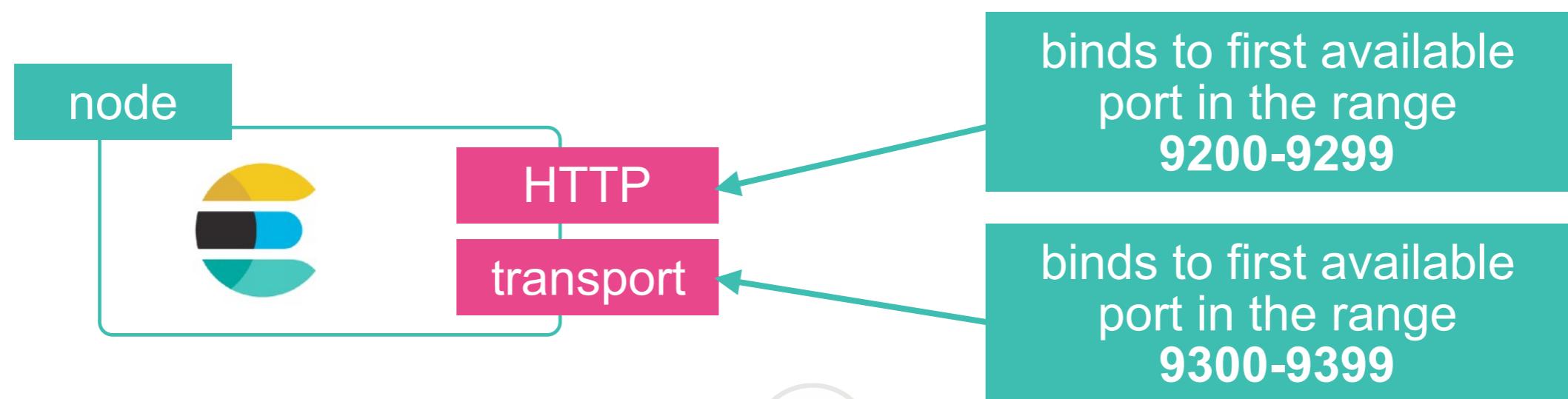


Development vs. Production Mode

.05-Mar-2019 - 0

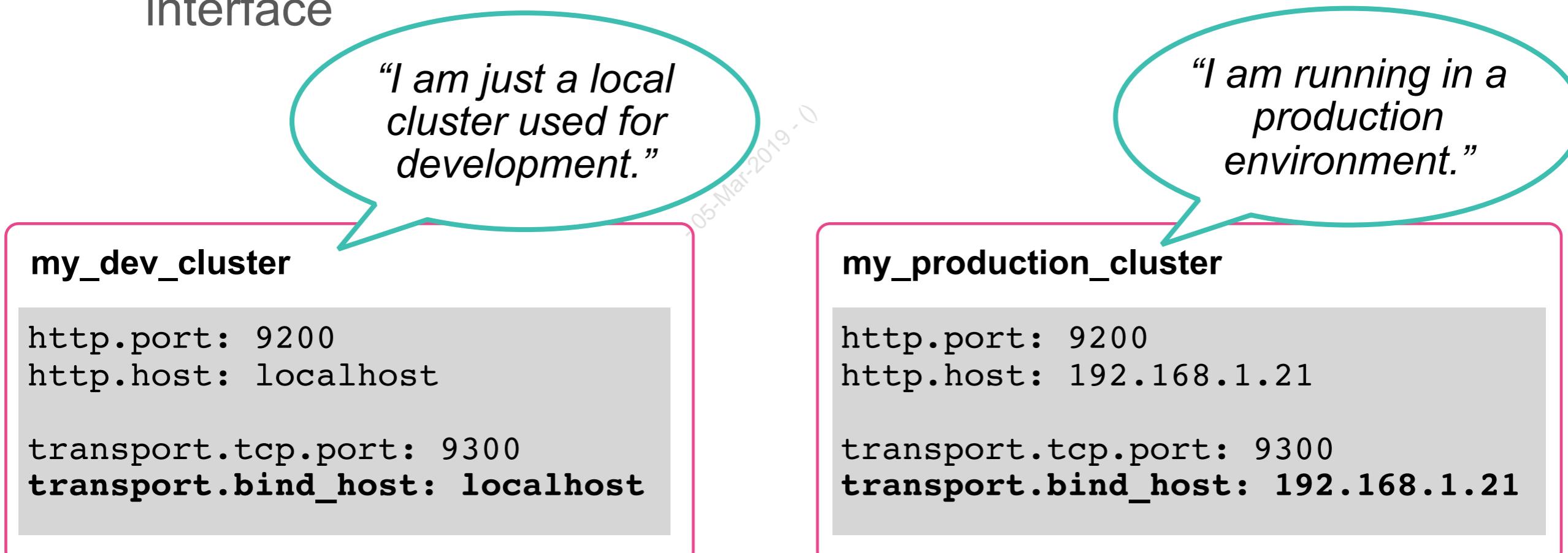
HTTP vs. Transport

- There are two important network communication mechanisms in Elasticsearch to understand:
 - **HTTP**: address and port to bind to for HTTP communication, which is how the Elasticsearch REST APIs are exposed
 - **transport**: used for internal communication between nodes within the cluster
- The defaults are fine for downloading and playing with Elasticsearch, but not useful for production systems
 - bind to **localhost** by default



Development vs. Production Mode

- Every Elasticsearch instance 5.x or later is either in development mode or production mode:
 - **development mode**: if it does *not* bind transport to an external interface (the default)
 - **production mode**: if it does bind transport to an external interface



Bootstrap Checks

- Elasticsearch has ***bootstrap checks*** upon startup:
 - inspect a variety of Elasticsearch and system settings
 - compare them to values that are safe for the operation of Elasticsearch
- Bootstrap checks behave differently depending on the mode:
 - **development mode**: any bootstrap checks that fail appear as ***warnings*** in the Elasticsearch log
 - **production mode**: any bootstrap checks that fail will cause Elasticsearch to ***refuse to start***
 - <https://www.elastic.co/guide/en/elasticsearch/reference/master/bootstrap-checks.html>



Bootstrap Checks

- A node in *production mode* must pass all of the checks, or the node will not start
 - the bootstrap checks fit into two categories

JVM Checks

- heap size
- disable swapping
- not use serial collector
- OnError and OnOutOfMemoryError
- server JVM

Linux Checks

- maximum map count
- maximum size virtual memory
- maximum number of threads
- file descriptor
- system call filter

Best Practices

~05-Mar-2010~Q

Networking Best Practices

- Avoid running over WAN links between datacenters
 - not officially supported by Elastic
- Try to have zero (or very few) hops between nodes
- If you have multiple network cards, separate **transport** and **http** traffic
 - bind to different network interfaces
 - use separate firewall rules for each kind of traffic
- Use long-lived HTTP connections
 - client libraries support this
 - or use a proxy/load-balancer



Storage Best Practices

- Prefer solid state disks (SSDs)
 - segments are immutable, so the *write amplification factor* approaches one and is a non-issue
- Local disk is king!
 - in other words, avoid NFS or SMB, AWS EFS, Azure filesystem
- Elasticsearch does not need redundant storage
 - replicas/software provide HA
 - local disks are better than SAN
 - RAID1/5/10 is not necessary



Storage Best Practices

- If you have multiple disks in the same server, you can set **RAID 0 or path.data**
- **RAID 0**
 - splits ("stripes") data evenly across two or more disks
 - perfect distribution of the data across the disks
 - if you lose one disk, you lose the data on all disks
- **path.data**
 - allows you to distribute your index across multiple SSDs
 - potential to an unbalanced distribution (all files belonging to a shard will be stored on the same data path)
 - if you lose one disk, the data on the other disks are preserved
 - may generate node level watermark issues if disks have different sizes



Storage Best Practices

- Use **noop** or deadline scheduler in the OS when using SSD
 - For details, see:
https://www.elastic.co/guide/en/elasticsearch/guide/current/hardware.html#_disks
- Spinning disks are OK for **warm** nodes
 - but, disable concurrent merges
- Trim your SSD's:
 - <https://www.elastic.co/blog/is-your-elasticsearch-trimmed>



Hardware Selection

- In general, choose ***medium machines over large machines***
 - loss of a large node has a greater impact
 - ***prefer*** six 4cpu x 64gb x 4 1tb drives
 - ***avoid*** 2 12cpu x 256gb x 12 1tb drives
- Avoid running multiple nodes on one server
 - one Elasticsearch instance can fully consume a machine
- Larger machines can be helpful as ***warm nodes***
 - configure shard allocation filtering as previously discussed



Cloud Strategies

- On Cloud, use the *discovery plugin*
 - because IPs can change frequently, the plugin dynamically configure the unicast hosts
- Span the cluster across more than one AZ
- Prefer ephemeral storage over network storage
- Snapshot to cloud storage with the **repository plugins**
- Use shard awareness and forced awareness
- Avoid instances marked with low networking performance



Throttles

- Elasticsearch has relocation and recovery throttles to ensure these tasks do not have a negative impact
- **Recovery:**

- for faster recovery, temporarily increase the number of concurrent recoveries:

```
PUT _cluster/settings
{
  "transient": {
    "cluster.routing.allocation.node_concurrent_recoveries": 2
  }
}
```

- **Relocation:**
- for faster rebalancing of shards, increase:

```
"cluster.routing.allocation.cluster_concurrent_rebalance" : 2
```



JVM Settings

~05-Mar-2019~

JVM Configuration

- Since Elasticsearch 6.0, only 64-bit JVMs are supported
- You can configure the Java Virtual Machine (JVM) two ways:
 - the **config/jvm.options** file (preferred)

```
-Xms30g  
-Xmx30g
```

- setting the **ES_JAVA_OPTS** environment variable

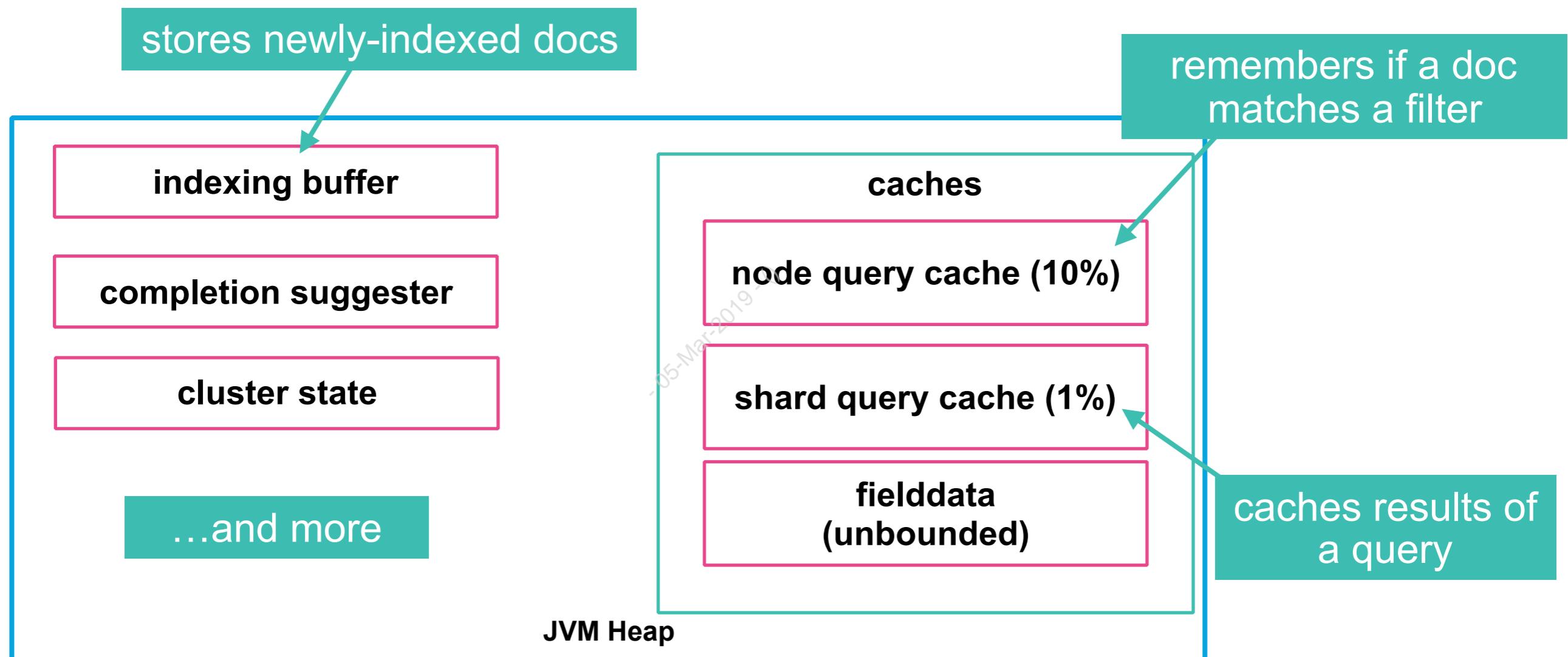
```
ES_JAVA_OPTS="-Xms30g -Xmx30g" bin/elasticsearch
```

- Elasticsearch has very good JVM defaults
 - avoid and be careful when changing them



What Goes on the Heap?

- Some of the major usage of the heap by Elasticsearch includes:



JVM Heap Size

- By default, the JVM heap size is 1 GB
 - likely not high enough for production
 - you can change it using **Xms** (min heap) and **Xmx** (max heap)

```
ES_JAVA_OPTS="-Xms8g -Xmx8g" ./bin/elasticsearch
```

- Some guidelines for configuring the heap size:
 - set **Xms** and **Xmx** to the same size (bootstrap check)
 - set **Xmx** to no more than 50% of your physical RAM
- Rule of thumb for setting the JVM heap is:
 - do not exceed more than 30GB of memory (to not exceed the compressed ordinary object pointers limit)
 - <https://www.elastic.co/blog/a-heap-of-trouble>



Production JVM Settings

1. JDKs have two modes of a JVM: *client* and *server*
 - *server* JVM is required in production mode
2. Configure the JVM to **disable swapping**
 - by requesting the JVM to lock the heap in memory through **mlockall** (Unix) or **virtual lock** (Windows)

~05-Mar-2019~0



Common Causes of Poor Query Performance

05 Mar 2019 10

Common Causes of Poor Query Performance

- Let's take a look at some common mistakes developers make that can have a negative effect on query performance
 - and discuss better ways to write your search queries!

~05-Mar-2019~0



How could we improve this query?

- Suppose we want to search for **blogs** that mention “elk” between 2016-2018
 - easy to find with a couple of **must** queries:

```
GET blogs/_search
{
  "query": {
    "bool": {
      "must": [
        {"match": {"content": "elk"}},
        {
          "range": {
            "publish_date": {
              "gte": 2016,
              "lte": 2018
            }
          }
        }
      ]
    }
  }
}
```

- Any thoughts on why this might not be the most efficient query?

Issue: Not Using Filters

- We could improve the performance of this query
 - a **range** query could take advantage of ***filter caching*** if it appeared in a **filter** clause (remember bit sets?)

```
GET blogs/_search
{
  "query": {
    "bool": {
      "must": [
        "match": {
          "content": "elk"
        }
      ],
      "filter": {
        "range": {
          "publish_date": {
            "gte": 2016,
            "lte": 2018
          }
        }
      }
    }
  }
}
```

If the **range** query gets cached,
it can execute faster on
subsequent searches



Issue: Aggregating Too Many Docs

- Aggregations are powerful, but they can consume a great deal of memory
- Whenever possible, ***limit the number of docs*** that an agg is computed over by using a **query or filter**
 - Let's look at a few examples of how to do this...

~05-Mar-2019~0



Solution: Limit the Scope with a Query

- Limit the scope of the aggregation by adding a **query**:

```
GET logs*/_search
{
  "size": 0,
  "query": {
    "bool": {
      "filter": {
        "range": {
          "runtime_ms": {
            "lt": 200
          }
        }
      }
    }
  },
  "aggs": {
    "my_aggs": {
      "range": {
        "field": "runtime_ms",
        "ranges": [
          {
            "from": 0,
            "to": 100
          },
          {
            "from": 100,
            "to": 200
          }
        ]
      }
    }
  }
}
```

Adding a **query** block
limits the scope of an agg

Solution: Use a Filter Bucket

- The **filter** bucket aggregation is useful when the **query** scope is different than the desired aggregation scope:

```
GET logs*/_search
{
  "size": 20,
  "query": {
    "match": {
      "language.url": "time-based indices"
    }
  },
  "aggs": {
    "not_found_requests": {
      "filter": {
        "match": {
          "status_code": 404
        }
      },
      "aggs": {
        "top_countries": {
          "terms": {
            "field": "geoip.country_name.keyword"
          }
        }
      }
    }
  }
}
```

The **query** is searching for requests to blogs that contain **time**, **based**, or **indices**

The **filter** bucket selects requests that returned a 404

The **agg** is over requests to blogs that contain **time**, **based**, or **indices** that returned 404

Solution: The Sampler Aggregation

- Another way to limit the scope of an agg is to use the **sampler** bucket aggregation
 - it filters out a sample of the top-scoring hits
 - can improve analytics by filtering out the long tail of low-quality matches
- The **sampler** aggregation is a great solution for scenarios where you do not want to analyze the entire dataset
 - and limiting the scope with a **query** or **filter** is not a viable option

Solution: The Sampler Aggregation

```
GET logs*/_search
{
  "size": 0,
  "query": {
    "bool": {
      "filter": {"match": {"language.url": "time-based indices"}}
    }
  },
  "aggs": {
    "my_sample": {
      "sampler": {
        "shard_size": 100
      },
      "aggs": {
        "top_countries": {
          "terms": {
            "field": "geoip.country_name.keyword"
          }
        }
      }
    }
  }
}
```

Requests to blogs that contain
time, *based*, or *indices*

Sample 100 of the top hits
from each shard

Aggregation executed on top of the best
results (likely an AND instead of an OR)

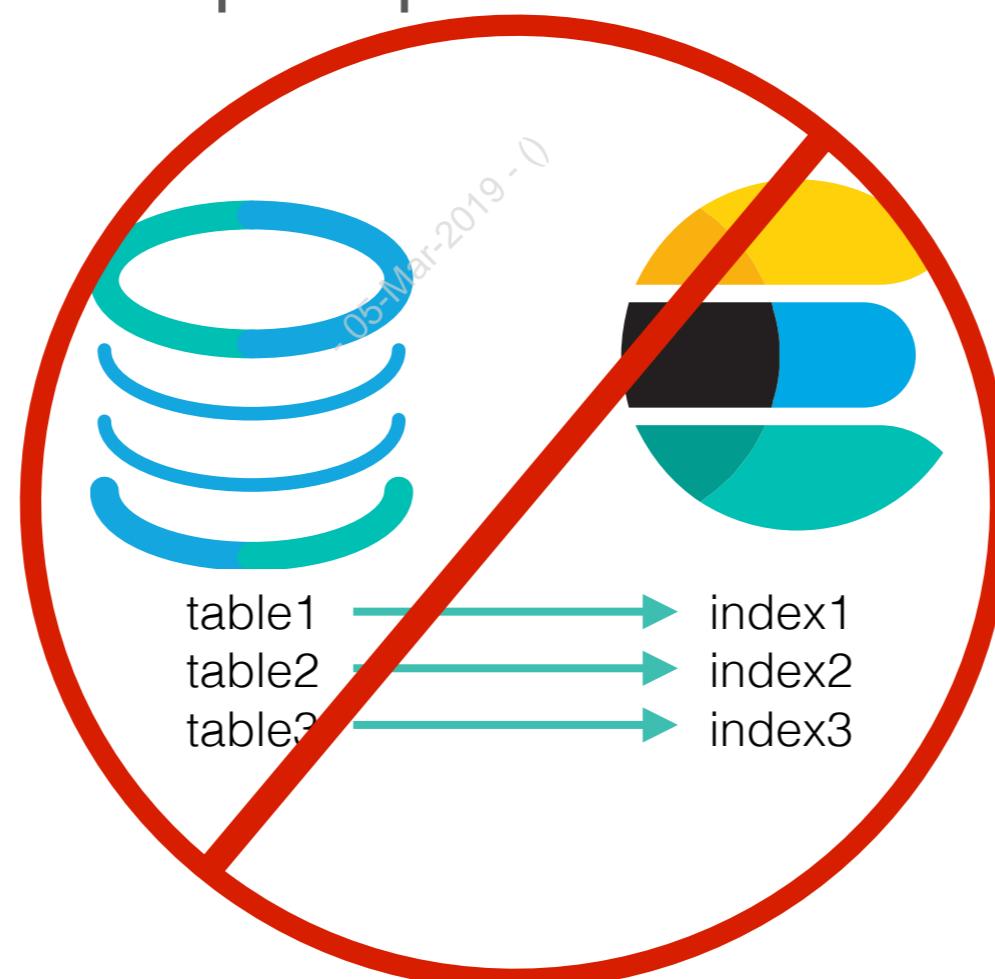
1500 out of 31063

```
"aggregations": {
  "my_sample": {
    "doc_count": 1500,
    "top_countries": {
      "doc_count_error_upper_bound": 4,
      "sum_other_doc_count": 251,
      "buckets": [
        ...
      ]
    }
  }
}
```



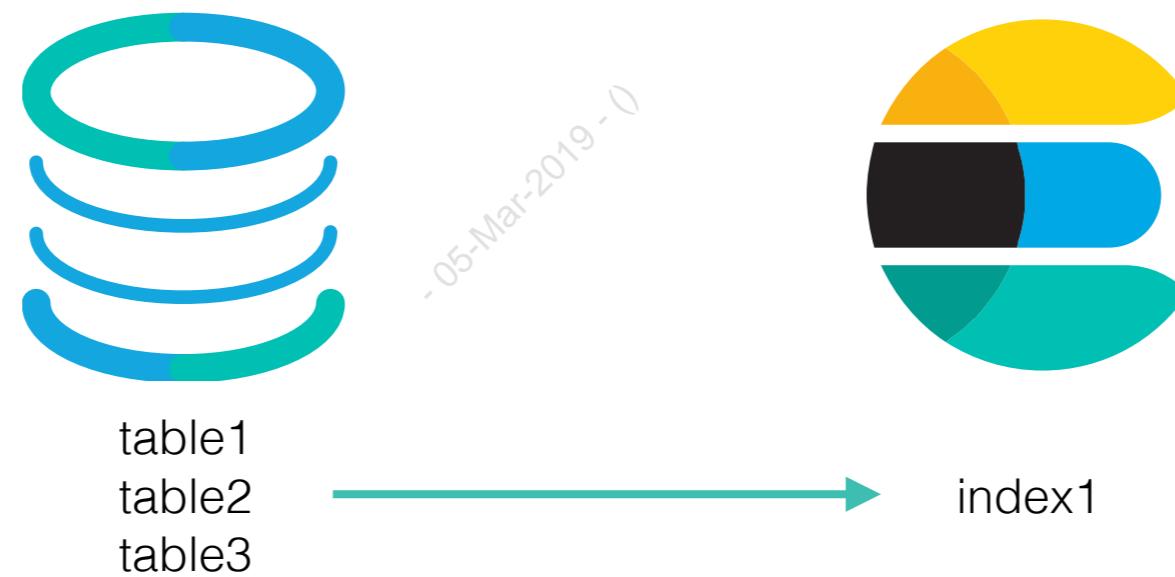
Issue: Confusing Elasticsearch w/ RDBMS

- Elasticsearch is *not* a relational database
 - So don't try to make it look like one!
- Trying to design indexes that look like relational tables in a database is not a good idea
 - and will only result in poor performance issues



Solution: Denormalize Your Data

- It is very important that you ***denormalize your data***
 - do not use **nested** types when it is not required
 - avoid using parent/child relationships (***denormalize*** instead!)



Issue: Too Many Shards

- Remember: a query has to hit every shard
 - More shards == slower query performance
- The default of 5 shards can actually be too high for some scenarios
 - Having thousands of 100MB indices with 5 shards each is not good
- **The solution?**
 - Shards can be fairly large (up to 40GB is good)...
 - ...so create as few of them as needed



Issue: Unnecessary Scripting

- In particular, using scripts at *query time* when *index time* might be a better option
 - For example, the length of the **title** field has to be calculated on each document each time this **query** is executed:

```
GET blogs/_search
{
  "query": {
    "bool": {
      "must": [
        "match": {"title": "network"}
      ],
      "filter": [
        {
          "script": {
            "script": {
              "source": "doc['title.keyword'].value.length() > 50"
            }
          }
        }
      ]
    }
  }
}
```



Solution: Index Common Computations

- Instead of using a search-time script, compute the value once and index it
 - The length of the **title** field can now be easily and quickly searched on

```
PUT _ingest/pipeline/comment_length
{
  "processors" : [
    {
      "script": {
        "lang": "painless",
        "source": "ctx.title_length = ctx.title.length();"
      }
    }
  ]
}
```

Compute the length once at index time (by perhaps using a pipeline)

Issue: Expensive Regular Expressions

- You have to be careful with searches that use regular expressions
 - These two **regexp** queries seem quite similar, but one is much more expensive. (Which one?)

```
GET blogs/_search
{
  "query": {
    "regexp": {
      "title": "net.*"
    }
  }
}
```

```
GET blogs/_search
{
  "query": {
    "regexp": {
      "title": ".*work"
    }
  }
}
```

Solution: Use regexp with Caution

- Avoid leading wildcards
 - Perhaps index your data in such a way as to avoid the need for leading wildcards
- And understand that, in general, regular expressions can be expensive

```
GET blogs/_search
{
  "query": {
    "regexp": {
      "title": ".*work"
    }
  }
}
```

A regex that has
lookarounds or short
prefixes can be expensive



Solution: Clever Indexing

- If you really need to search the end of your tokens, simply index them using the **reverse** token filter
 - and search them using a prefix regular expression:

```
GET blogs/_search
{
  "query": {
    "regexp": {
      "title.reversed": "krow.*"
    }
  }
}
```

"Brewing in Beats: New community
Beat for **network** devices"

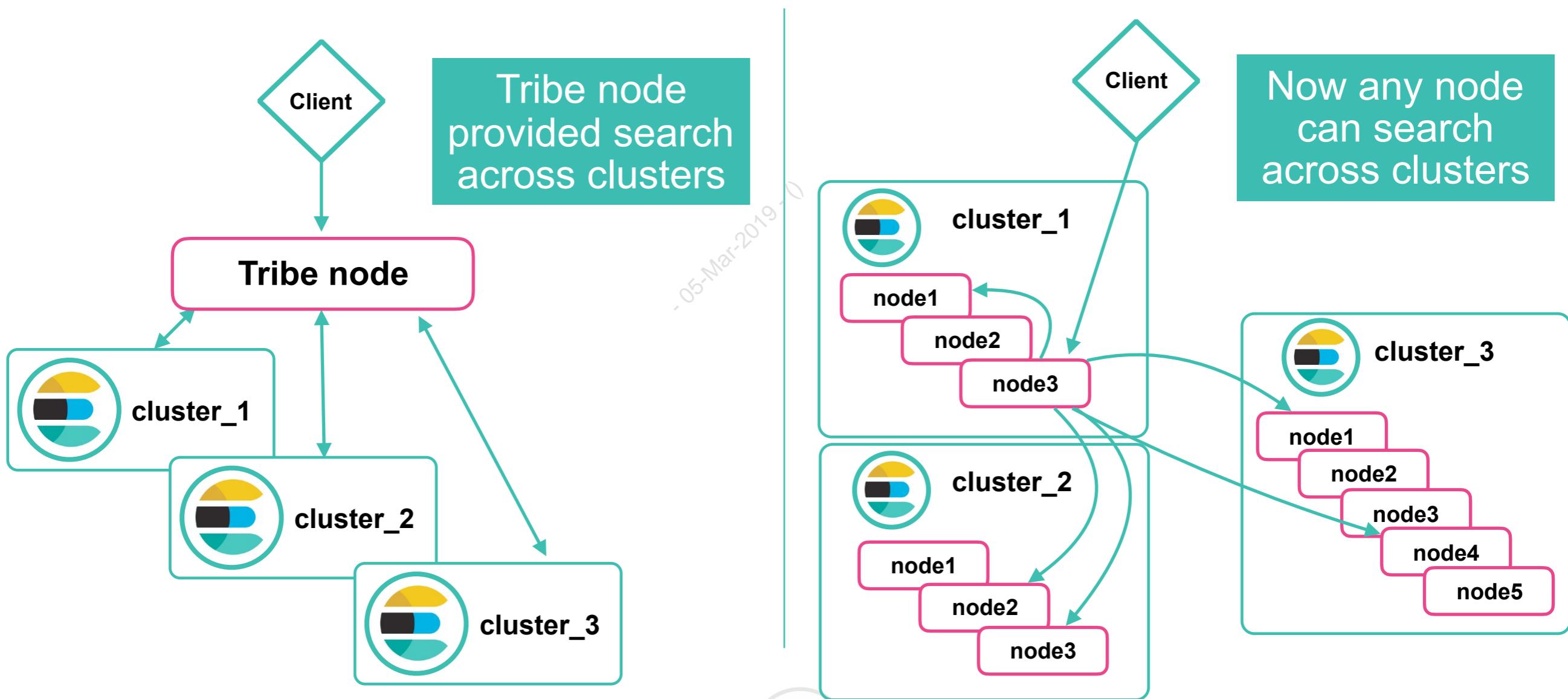


Cross Cluster Search

~05-Mar-2019 (~)

What is Cross Cluster Search?

- **Cross Cluster Search** is a feature that allows any node to act as a client for executing queries across multiple clusters
 - introduced in Elasticsearch 5.3, it replaces the tribe node functionality



Registering Remote Clusters

- Remote clusters are configured in the cluster settings
 - using the `cluster.remote` property
 - `seeds` is a list of nodes in the remote cluster used to retrieve the cluster state when registering the remote cluster

```
PUT _cluster/settings
{
  "persistent": {
    "cluster.remote" : {
      "germany_cluster" : {
        "seeds" : [ "my_server:9300", "64.33.90.170:9300" ]
      }
    }
  }
}
```

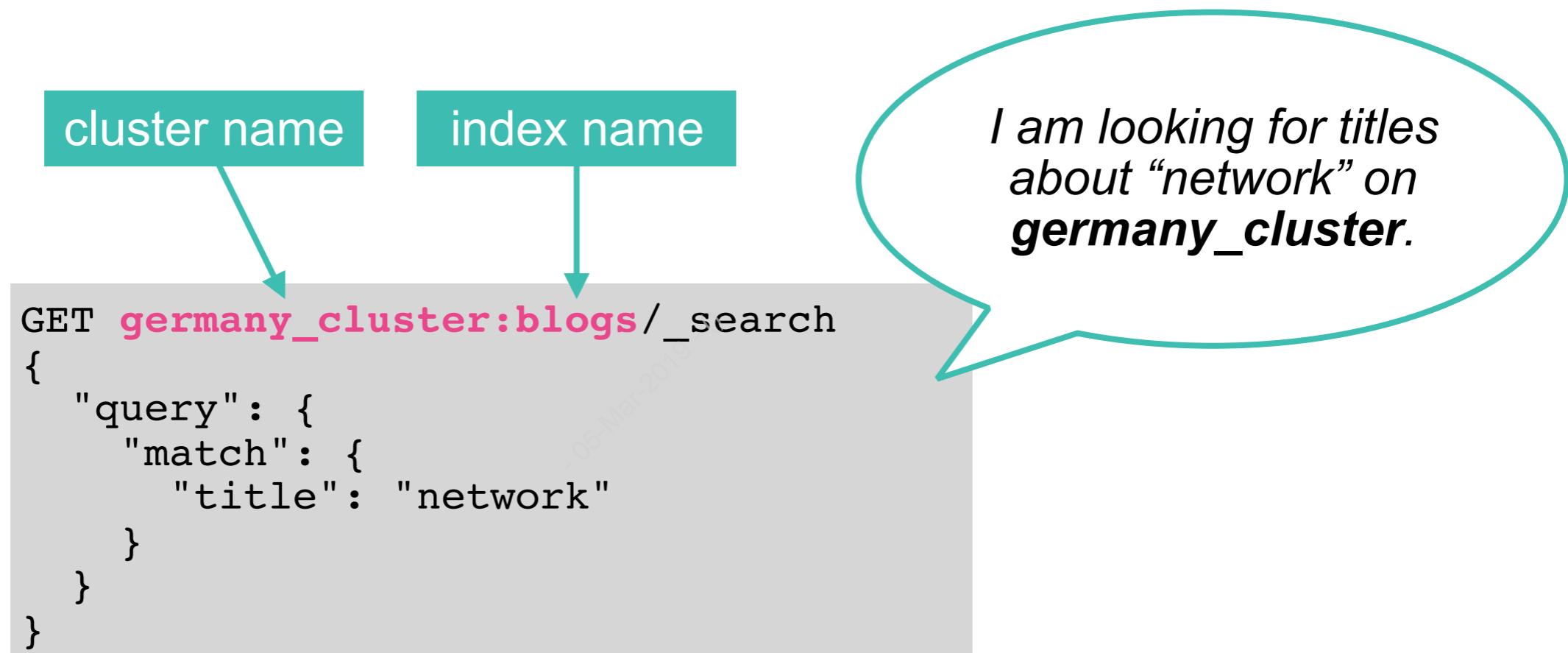
A name you assign to the remote cluster

List of seed nodes



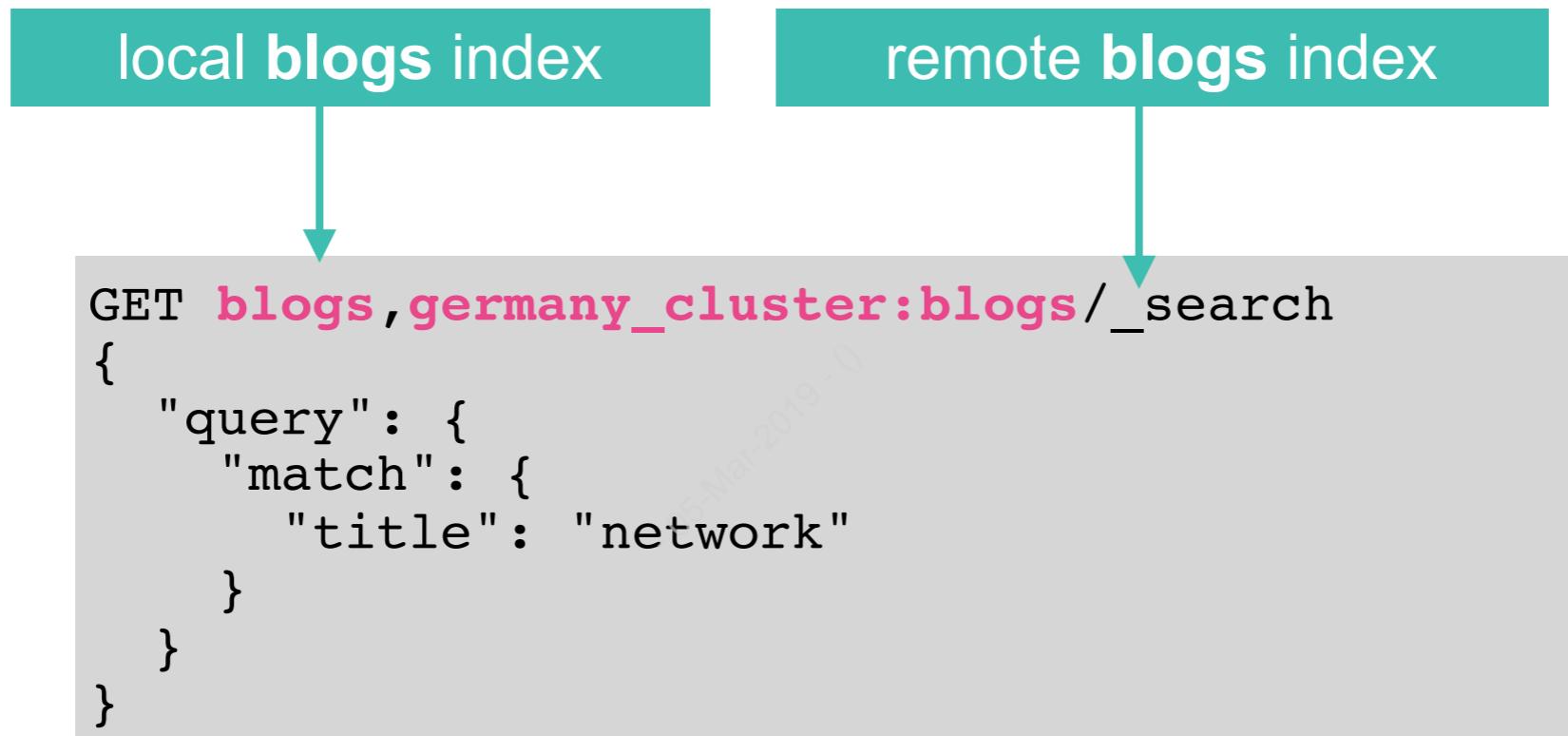
Searching Remotely

- To search an index on a remote cluster, prefix the index name with the remote cluster name:



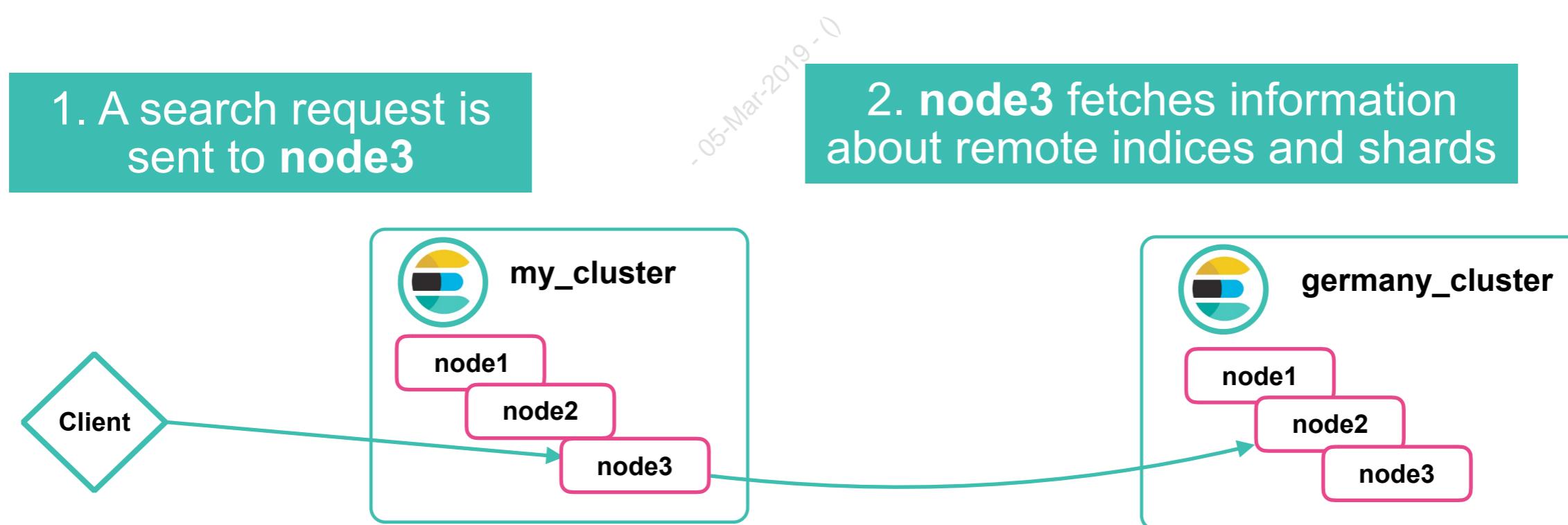
Cross Cluster Searching

- To perform a search across multiple clusters, simply list the cluster names and indices in the `_search`:



How It Works

- From a search execution perspective, there is no difference between local indices and remote indices
 - as long as the coordinating node can reach some nodes belonging to the remote clusters
 - the coordinating node resolves the shards of remote indices by sending one `_search_shards` request per cluster

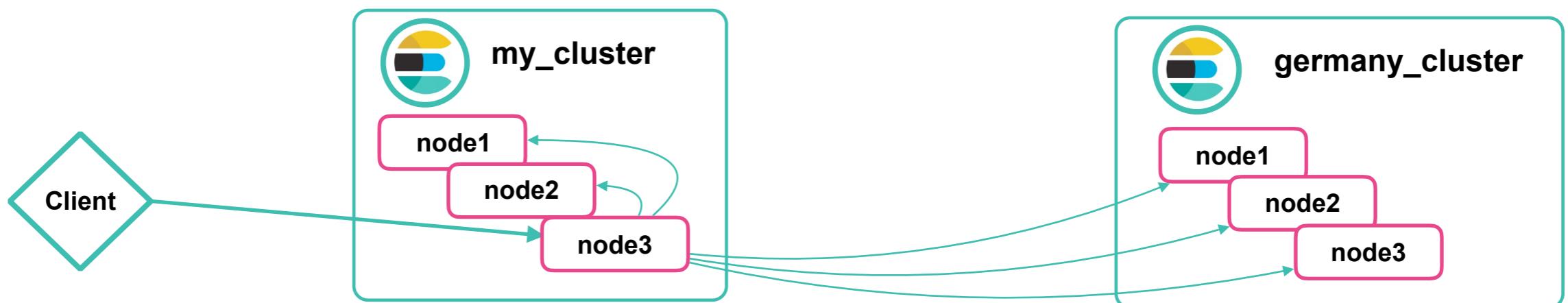


How It Works

- Once the details are retrieved of where the remote shards are located, the search is executed just like any other search

3. The query gets executed on the relevant local shards...

...and the query gets executed on the relevant remote shards

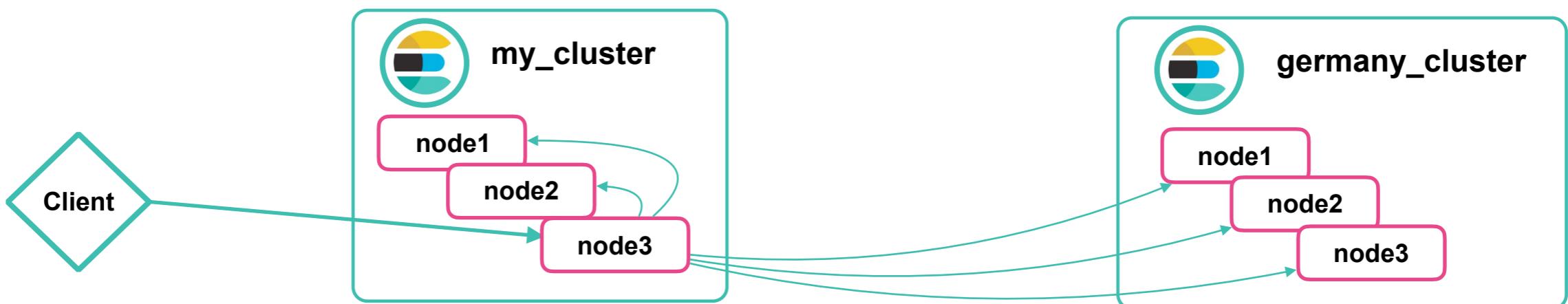


How It Works

- The coordinating node gets “size” hits from each shard (local and remote) and performs reduction
- Then the top hits are fetched from the relevant shards
 - and returned to the client

4. **node3** gets “size” hits from each shard and determines the top hits

5. The top hits are fetched by **node3** and returned to the client



The Response

- All results retrieved from a remote index will be prefixed with their remote cluster name

```
GET blogs,germany_cluster:blogs/_search
{
  "query": {
    "match": {
      "title": "network"
    }
  }
}
```

```
"hits": [
  {
    "_index": "germany_cluster:blogs",
    "_type": "doc",
    "_id": "3s1CKmIBCLh5xF6i7Y2g",
    "_score": 4.8329377,
    "_source": {
      "title": "Using Nmap + Logstash to Gain
Insight Into Your Network",
      ...
    }
  },
  {
    "_index": "blogs",
    "_type": "doc",
    "_id": "Mc1CKmIBCLh5xF6i7Y",
    "_score": 4.561167,
    "_source": {
      "title": "Brewing in Beats: New community
Beat for network devices",
      ...
    }
  },
  ...
]
```

Using Wildcards

- You can use wildcards for the names of the remote clusters
 - The following search queries **messages** on the local cluster and all registered remote clusters:

```
GET blogs,*:blogs/_search
{
  "query": {
    "match": {
      "title": "network"
    }
  }
}
```

Same hits as the query on
the previous slide



Overview of Upgrades

~05-Mar-2018

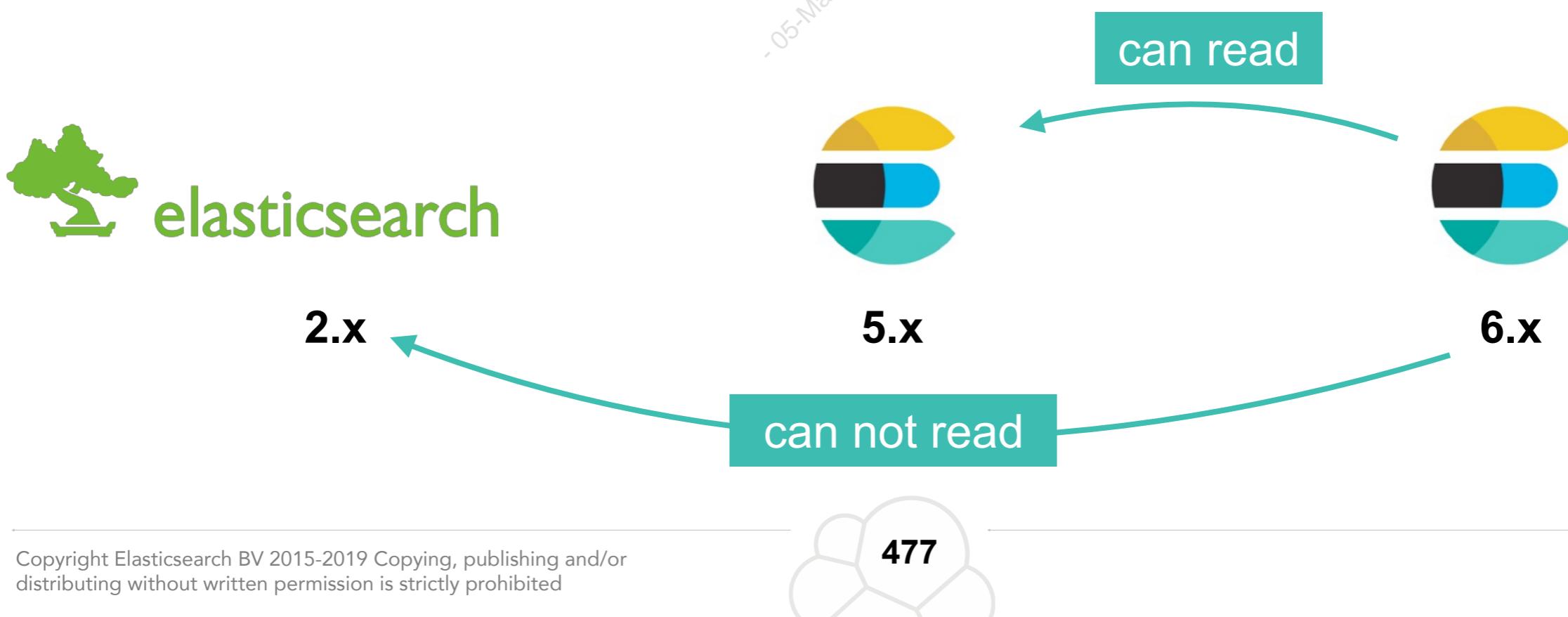
Versions

- Elasticsearch versions are denoted as **X.Y.Z**
 - **X** is the *major version*
 - **Y** is the *minor version*
 - **Z** is the *patch* level or *maintenance* release
- Elasticsearch can use indices created in the previous major version, but older indices must be reindexed or deleted.
 - 6.x can use indices created in 5.x
 - 6.x **cannot** use indices created in 2.x or before
 - 5.x can use indices created in 2.x
 - 5.x **cannot** use indices created in 1.x or before
 - ...



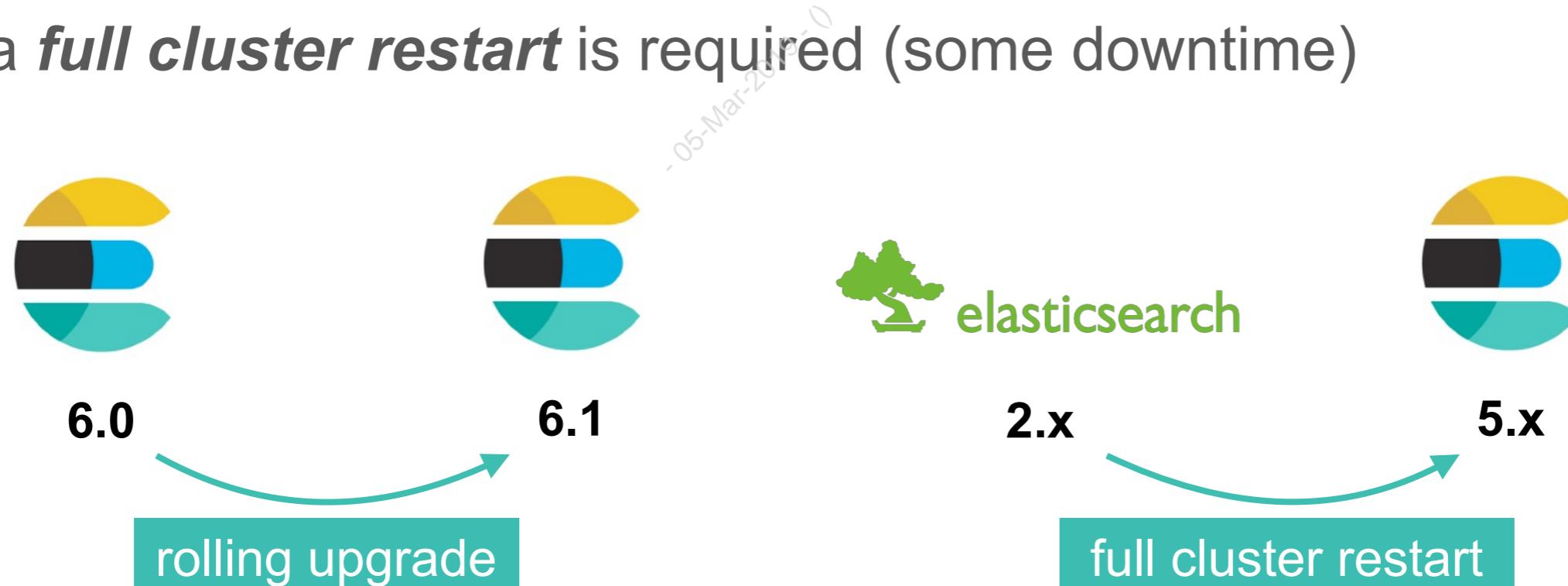
Upgrading Old Versions

- Elasticsearch can read indices from the *previous major version only*
 - so an index in 5.x can be used in a 6.x cluster
 - but an index in 2.x can *not* be used in a 6.x cluster
 - Elasticsearch will fail to start if
- Upgrading from 2.x or 1.x to 6.x *requires all your old indices to be reindexed*



Overview of Upgrades

- Elasticsearch has a fairly rapid release cycle
 - at some point, you will want to upgrade a cluster to a newer version
- In general, there are two possible scenarios that occur when upgrading a cluster:
 - a ***rolling upgrade*** can occur (no downtime)
 - a ***full cluster restart*** is required (some downtime)



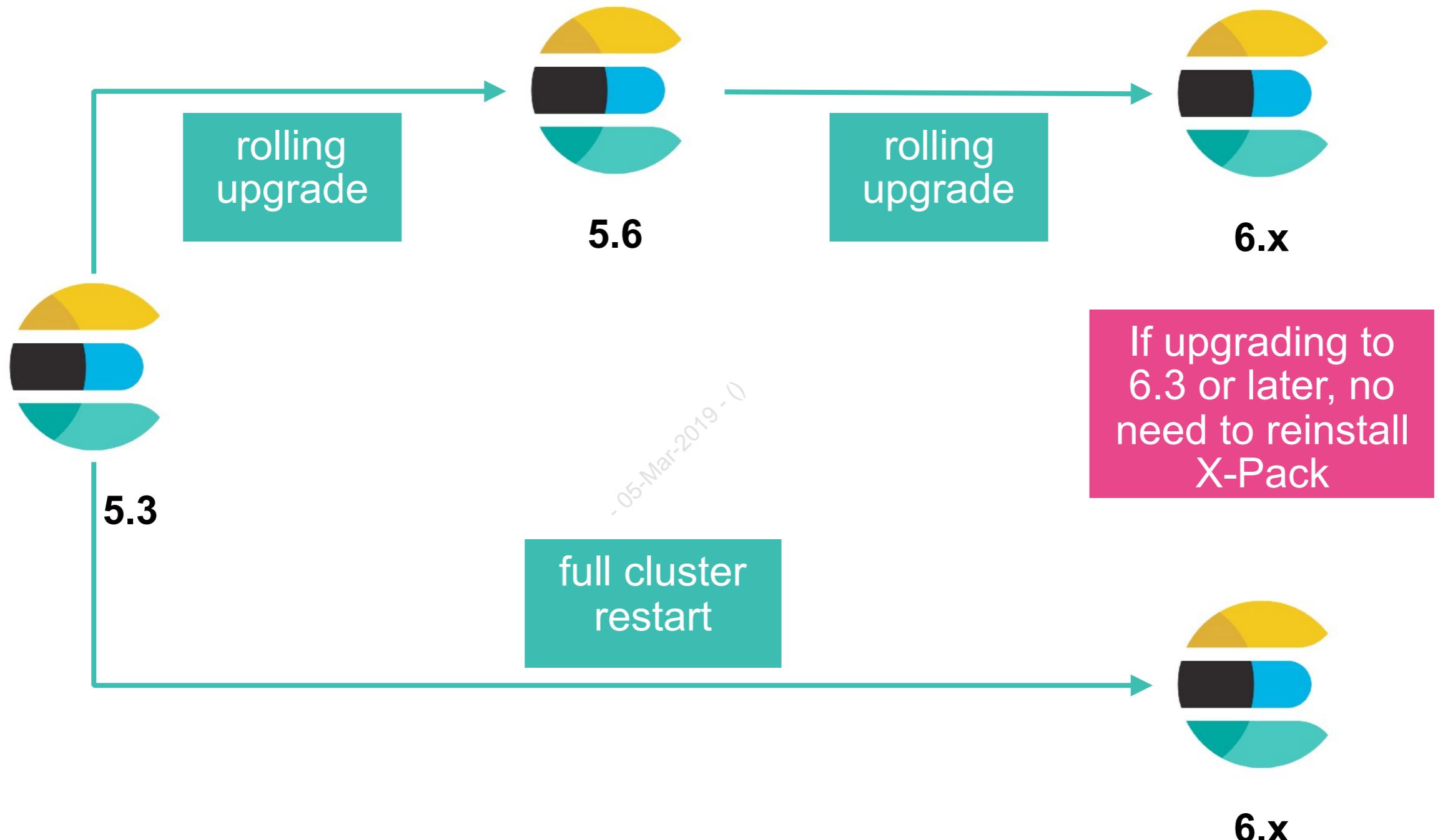
Overview of Upgrades

Upgrade From	Upgrade To	Supported Upgrade Type
5.x	5.y	Rolling upgrade
5.6	6.x	Rolling upgrade
5.0 - 5.5	6.x	Full cluster restart
<= 2.x	6.y	Full cluster restart
6.x	6.y	Rolling upgrade

Major version upgrade
without full cluster restart?



Upgrade 5.x to 6.x



Upgrading

- Unfortunately, there are too many options and details in the Elastic upgrade process
- It would be impossible to cover all possible combinations in a few minutes
- Elastic created an upgrade guide in which
 - you answer a few questions
 - and we build a list of the steps you should follow
 - https://www.elastic.co/products/upgrade_guide
- A cluster restart will be part of any list, so let's talk about it...



Cluster Restart

-05-Mar-2019-



Cluster Restart

- Rolling restart
 - zero downtime
 - reads and writes continue to operate normally
- Full cluster restart
 - cluster unavailable during update
 - updates are usually faster

~05-Mar-2019~0

Steps for a Rolling Restart

- A **rolling restart** allows the nodes in the cluster to be upgraded one at a time
 - by using a *rolling restart*
- To perform a rolling restart:
 1. stop non-essential indexing (if possible)
 2. disable shard allocation
 3. stop and update one node
 4. start the node
 5. re-enable shard allocation and wait
 6. GOTO step 2 (until all nodes are updated)

Step 0: backup your cluster!

→ 05-Mar-2019 - 0

Rolling Restart

- **Step 1:** stop indexing data (if possible)
 - if you still need to index, that is OK but shard recovery will take longer

~05-Mar-2019~0



Rolling Restart

- Step 2: Disable shard allocation
 - and perform a synced flush
 - recall the default is “1m”, but better to just disable it entirely:

```
PUT _cluster/settings
{
  "transient": {
    "cluster.routing.allocation.enable" : "none"
  }
}

POST _flush/synced
```

- Use **new_primaries** instead of **none** to allow the creation of new indices during the rolling restart

Rolling Restart

- **Step 3:** Update the node:
 - stop the node you want to update
 - update the machine/application

~05-Mar-2019~0



Rolling Restart

- **Step 4:** Start the node up again
 - and wait for it to join the cluster
 - you can use the following command to confirm: `GET _cat/nodes`

~05-Mar-2019~ Ø



Rolling Restart

- Step 5: Reenable shard allocation,

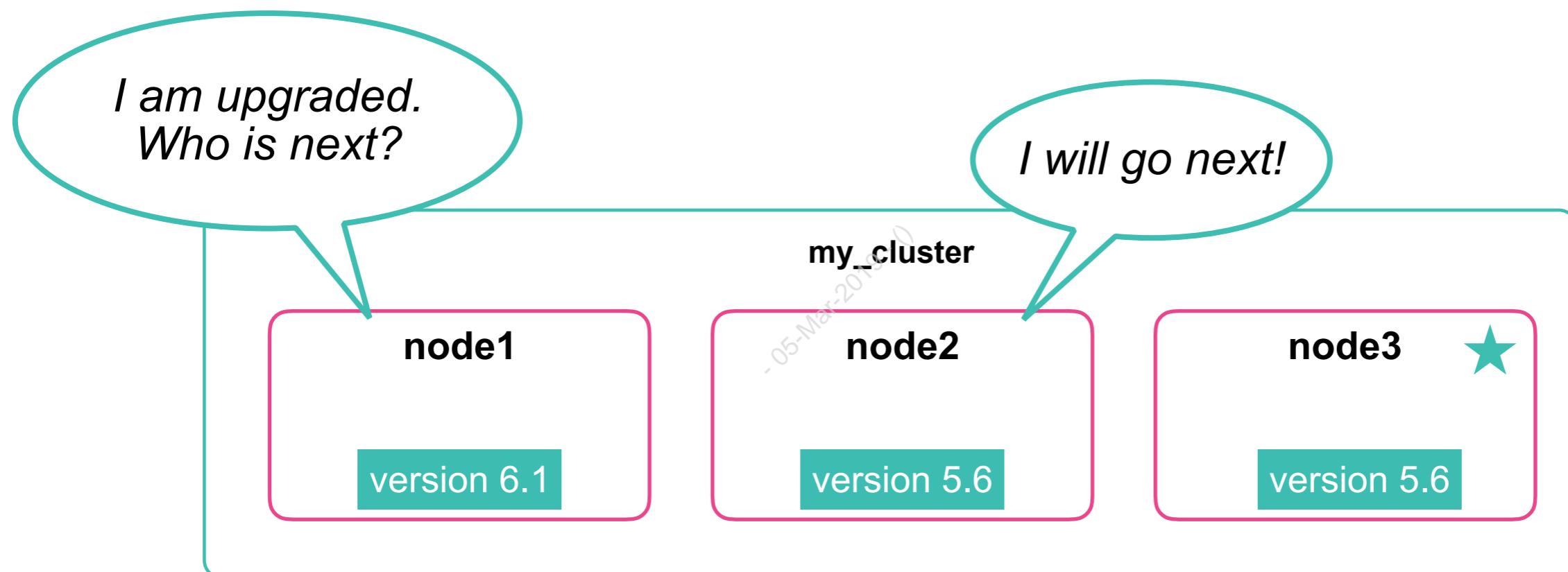
```
PUT _cluster/settings
{
  "transient": {
    "cluster.routing.allocation.enable" : "all"
  }
}
```

- then wait for the cluster to be green again:
 - if green is not possible, check there are no initializing or relocating shards before continuing

```
GET _cat/health
```

Repeat for Each Node

- Step 6 is to start the process over for the next node



Full Cluster Restart

- The cluster will be unavailable during the update
 - The steps are very similar to a rolling upgrade:
 1. stop indexing (e.g. disable writes, or make ES unreachable)
 2. disable shard allocation
 3. perform a synced flush
 4. shutdown and update **all** nodes
 5. start all dedicated master nodes
 6. start the other nodes
 7. wait for yellow
 8. reenable shard allocation
- © Mar-2019 - 0*
-
- The diagram illustrates the 8 steps of a full cluster restart. Step 0 is highlighted with a teal box and an arrow pointing to it from the text "Step 0: backup your cluster!". Step 4 is annotated with "make sure to use ‘persistent’". Step 5 is annotated with "downtime is here". Step 6 is annotated with "and wait for the election".

Chapter Review

~05-Mar-2019~

Summary

- A node in ***production mode*** must pass a series of checks, or the node will not start
- For best performance, choose ***SSD over spinning disks***
- Local disks are preferred - the software provides HA
- In general, choose ***medium machines over large machines***
- Cross cluster search allows you to search multiple clusters within the same request.
- Upgrading to a new major version of Elasticsearch usually requires a ***full cluster restart***, but can also be done with a ***rolling upgrade*** from 5.6 to 6.x
- A ***rolling upgrade*** allows the nodes in the cluster to be restarted one at a time



Quiz

1. **True or False:** In production mode, if a bootstrap check fails then the node will not start.
2. **True or False:** SAN storage is preferred over local disks to provide high availability of data.
3. **True or False:** It is a good idea to separate the **transport** and **HTTP** traffic over different network interfaces.
4. Why would you use cross cluster search?
5. **True or False:** You can search and index documents during a rolling restart.
6. What is the benefit of performing a synced flush right before a node restart?



Lab 9

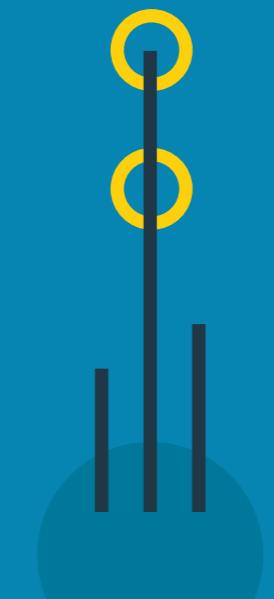
From Dev to Production

05-Mar-2019



Conclusions

,05-Mar-2019 - 0



Resources

- <https://www.elastic.co/learn>
 - <https://www.elastic.co/training>
 - <https://www.elastic.co/community>
 - <https://www.elastic.co/docs>
- <https://discuss.elastic.co>

~05-Mar-2019 - 0



Elastic Training

Empowering Your People

Immersive Learning

Lab-based exercises and knowledge checks to help master new skills

Solution-based Curriculum

Real-world examples and common use cases

Experienced Instructors

Expertly trained and deeply rooted in everything Elastic

Performance-based Certification

Apply practical knowledge to real-world use cases, in real-time

FOUNDATION



SPECIALIZATIONS



05-Mar-2019 - 0

Elastic Consulting Services

ACCELERATING YOUR PROJECT SUCCESS

FLEXIBLE SCOPING

Shifts resource as your requirements change

PHASE-BASED PACKAGES

Align to project milestones at any stage in your journey

GLOBAL CAPABILITY

Provide expert, trusted services worldwide

EXPERT ADVISORS

Understand your specific use cases

PROJECT GUIDANCE

Ensures your goals and accelerate timelines

.05-Mar-2019 - 0

Thank you!

Please complete the online survey

-05-MAY-2019-

Quiz Answers

-05-Mar-2019-0

Chapter 1 Quiz Answers

1. True
2. Field names, term dictionary, term frequency, term proximity, deleted documents, stored fields, normalization factors
3. False! Only call `_forcemerge` on read-only indices
4. True
5. The response is only returned after the document is searchable (in a segment)
6. True. The write operation is written in the translog, which is fsynced to disk. So, when a client gets an OK for the write operation, it means that it is fsynced in the translog.
7. Any time that you will not have more writes and would like to speedup recoveries, e.g. cluster restart.



Chapter 2 Quiz Answers

1. True
2. False. Only documents indexed after the mapping change will pick up the new field
3. Documents in the source index that have changed (have a higher `_version` number) will overwrite older documents in the destination
4. All of the the documents in the index! It is a `match_all` query, so all documents are hits
5. The document will be indexed into the index named `clientip.country_iso_code`



Chapter 3 Quiz Answers

1. False. They are powerful, but expensive and should **NOT** be used often
2. Using the `exist` query inside a **must_not**
3. True
4. Lots of reasons, including to avoid repeating code in multiple places, to minimize mistakes, make it easier to test and execute your queries, share queries between applications, and allow users to only execute a few predefined queries
5. In **buckets_path**, when a pipeline agg is defined at a higher level then the sub-aggregation it needs to refer to
6. True
7. Use a **percentile** aggregation

Chapter 4 Quiz Answers

1. The current day's index could be a **hot** node handling indexing and queries, while all previous indices could be on **warm** nodes (assuming they do not get queried as often)
2. The index will be defined, but all of its shards will be unallocated and the cluster will go into a **red** status
3. Filtering is not “aware” of the physical configuration of your hardware. Awareness ensures that shards are distributed across “zones” that you define
4. Forced awareness never allows copies of the same shard to be in the same zone - shard allocation awareness does



Chapter 5 Quiz Answers

1. True
2. In terms of searching, it would essentially be equivalent
3. Overallocate! It allows for future scaling of the cluster
4. False. There are many scenarios where 1 shard might actually be optimal, especially if the data all fits in a single shard
5. 9 or 10, depending upon if you want some extra buffer room. And that depending up on other requirements it may be a single index or multiple indices that total the 9 or 10 shards

Chapter 6 Quiz Answers

1. The field would produce better search results if the individual developer environments were split apart and stored in an array, instead of as a delimited string
2. It is a numeric value but it is stored as a string, so any situation that requires its value would require parsing and be tedious. This would be much better modeled as a **float**
3. Disable the field by setting “**enabled**” to **false**
4. Set “**dynamic**” to “**strict**”
5. “**intersect**”

Chapter 7 Quiz Answers

1. True
2. False
3. There is an overhead to parent/child - they are separate documents that must be joined. The join is fast, but nested objects are all a single document which will always be faster for searches (but more expensive for updates)
4. True
5. False
6. True

,05-Mar-2019-0



Chapter 8 Quiz Answers

1. True, but only the commercial version does. The free version can only monitor one
2. If a cluster fails, you will be able to view its history and perhaps diagnose the issue. There are also performance and security benefits.
3. 10 seconds
4. Look at the thread pool queues either with `_nodes/thread_pool` or `_cat/thread_pool`
5. Trigger, Input, Condition, Transform, Action
6. False. Only users with `watcher_admin` roles
7. True, using Monitoring data. create a watch on `monitoring-es*` in the ``node_stats.fs.total.free_in_bytes`` field



Chapter 9 Quiz Answers

1. True
2. False: SAN is not needed
3. True
4. To have a single view of the data spread across multiple clusters.
5. True. You can, but it slows down the recovery time
6. It greatly speeds up the recovery time of indices that have not changed while the node (or nodes) was down

Elasticsearch Engineer II

Course: Elasticsearch Engineer II

Version 6.5.1

© 2015-2019 Elasticsearch BV. All rights reserved. Decompiling, copying, publishing and/or distribution without written consent of Elasticsearch BV is strictly prohibited.

~05-Mar-2019~0