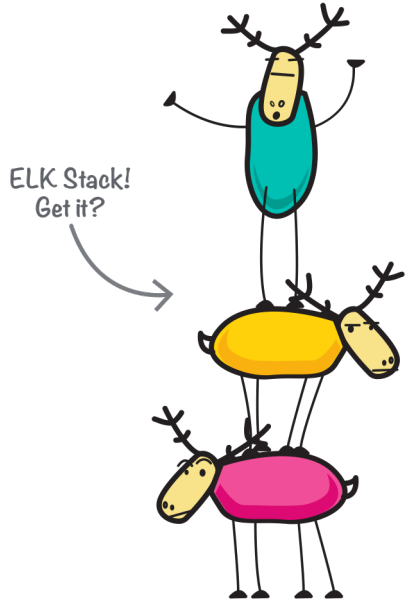
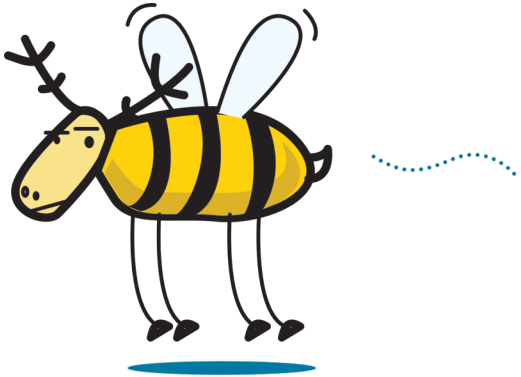
Elasticsearch workshop

Ahead of change

ELK(B) stack and elastic stack





elastic stack

Agenda

- Elastic Stack Overview
- CRUD operations
- Querying Data
- Node Types
- Understanding Shards
- Mappings and Grok

1. Elastic Stack Overview

Once upon a time…

- As any good story begins, “Once upon a time…”
  - In 1999, *Doug Cutting* created an open-source project called Lucene
- Lucene is
  - a search engine library entirely written in Java
  - a top-level Apache project, as of 2005
  - great for full-text search
- But, Lucene is also
  - a library (you have to incorporate it into your application)
  - challenging to use
  - not originally designed for scaling

The Birth of Elasticsearch

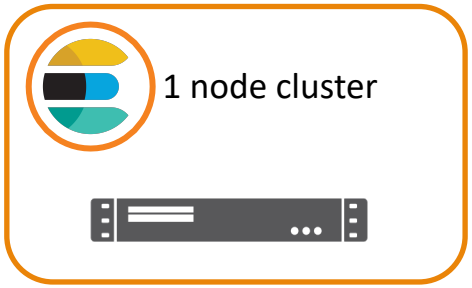
- In 2004, Shay Banon developed a product called Compass
  - Built on top of Lucene, Shay's goal was to have search integrated into Java applications as simply as possible
- The need for scalability became a top priority

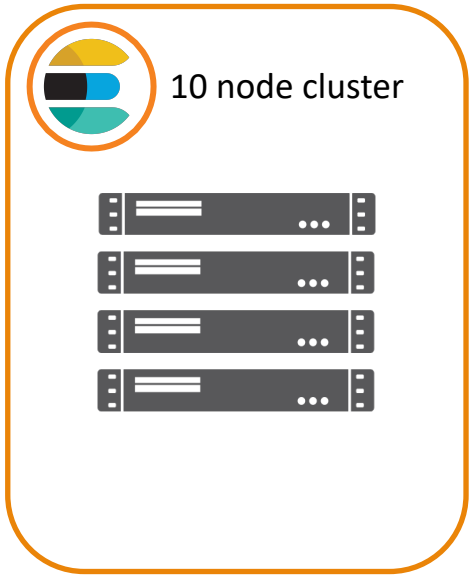
The Birth of Elasticsearch

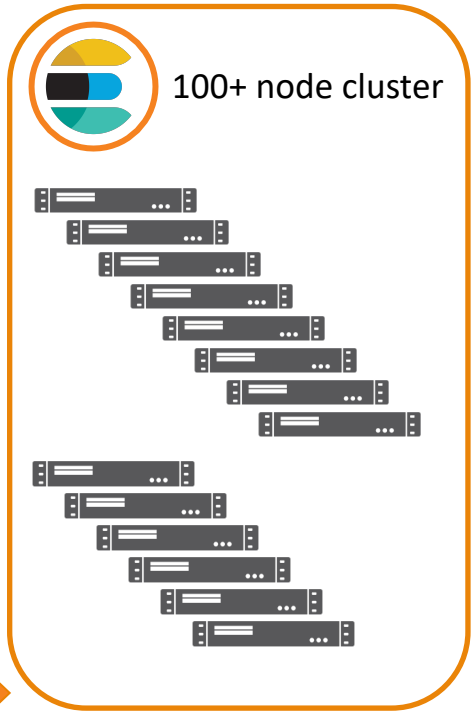
- In 2010, Banon completely rewrote Compass with two main objectives:
  - **1**. distributed from the ground up in its design
  - **2**. easily used by any other programming language
- He called it Elasticsearch
  - …and we all lived happily ever after!
- Today Elasticsearch is the most popular enterprise search engine

1. Distributed search

- Elasticsearch is distributed and scales horizontally







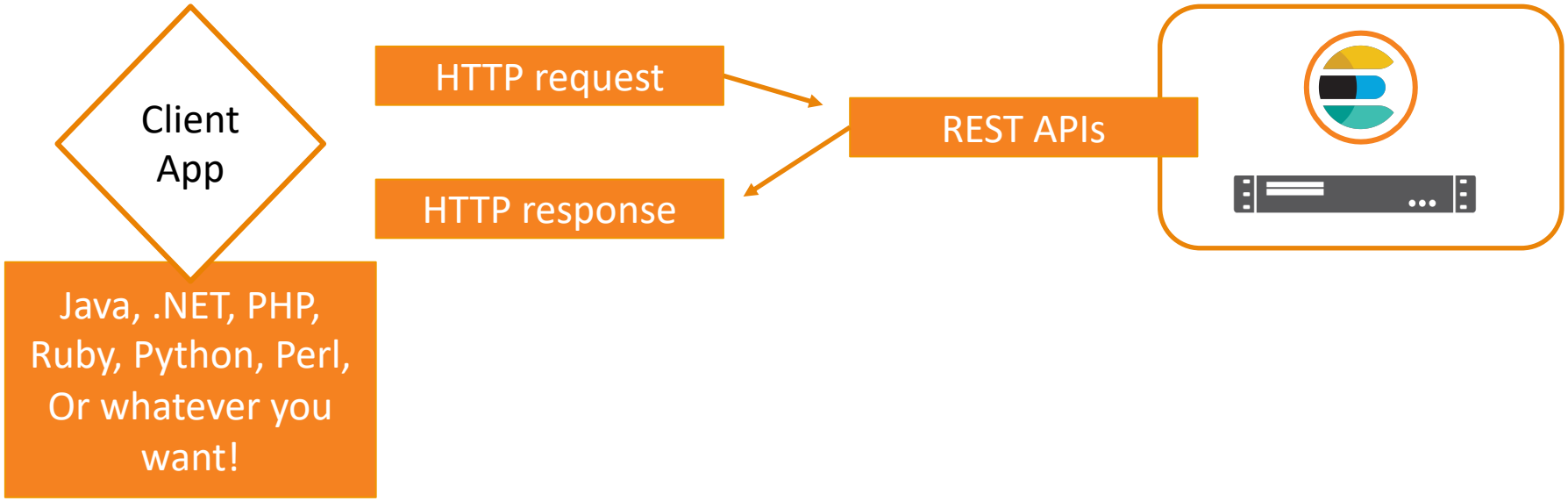
A ***node*** is an instance of Elasticsearch

A ***cluster*** is a collection of Elasticsearch nodes

Your cluster can grow as your needs grow

2. Easily used by other languages

- Elasticsearch provides REST APIs for communication with a cluster over HTTP



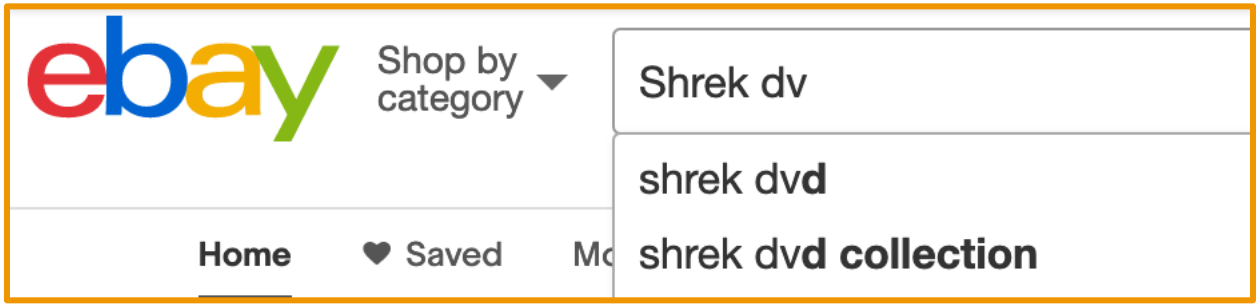
Why Elastic Stack?

You Know, for Search!

- …and ***logging***
- and ***metrics***
- and ***security analytics***
- and ***lots more!***

Search

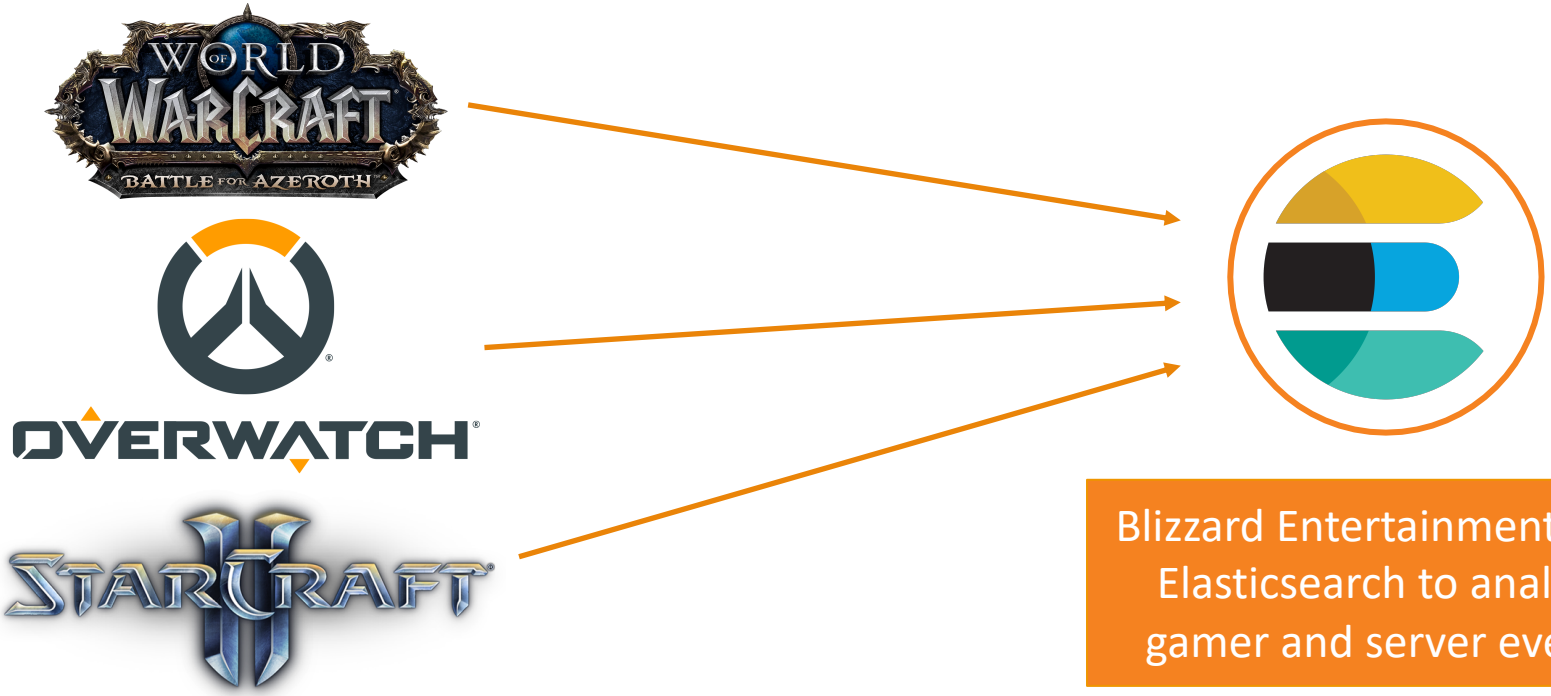
- ***Full-text search*** is where it all began



eBay uses Elasticsearch to search 800+ million listings in sub-seconds

Logging

- A popular use case for the Elastic Stack is storing and analyzing ***log data***



Blizzard Entertainment uses Elasticsearch to analyze gamer and server events

Metrics

- Using the Elastic Stack to analyze all types of ***metric data***, from CPU usage to the weather on Mars

The Mars Curiosity Rover sends telemetry, sensor, and photo data to Elasticsearch for real-time analysis

Business Analytics

- Using the Elastic Stack to ***analyze business needs*** and build custom applications
  - From studying purchasing patterns, to improving medial care, to matching people together

It's a Match!

Elasticsearch is at the core of Tinder's framework

Security Analytics

- Using the Elastic Stack to ***analyze security issues and threads***

Slack uses Elasticsearch to monitor malicious activity

Elastic Stack Components

Elastic Stack Components

Elasticsearch

Kibana

Logstash

Beats

- The ***Elastic Stack*** is a collection of products with ***Elasticsearch*** at the heart
  - Take data from any source, in any format, and search, analyze, and visualize in real time

Beats

- ***Beats*** are single-purpose data shippers
  - e.g. ***Filebeat***, ***Metricbeat***, ***Packetbeat***, ***Winlogbeat***, etc.
  - Lightweight agents that send data from your machines to Logstash or Elasticsearch

Logstash

- ***Logstash*** is a server-side data processing pipeline
  - Ingest data from a multitude of sources to simultaneously parse, transform and prepare your data for ingestion

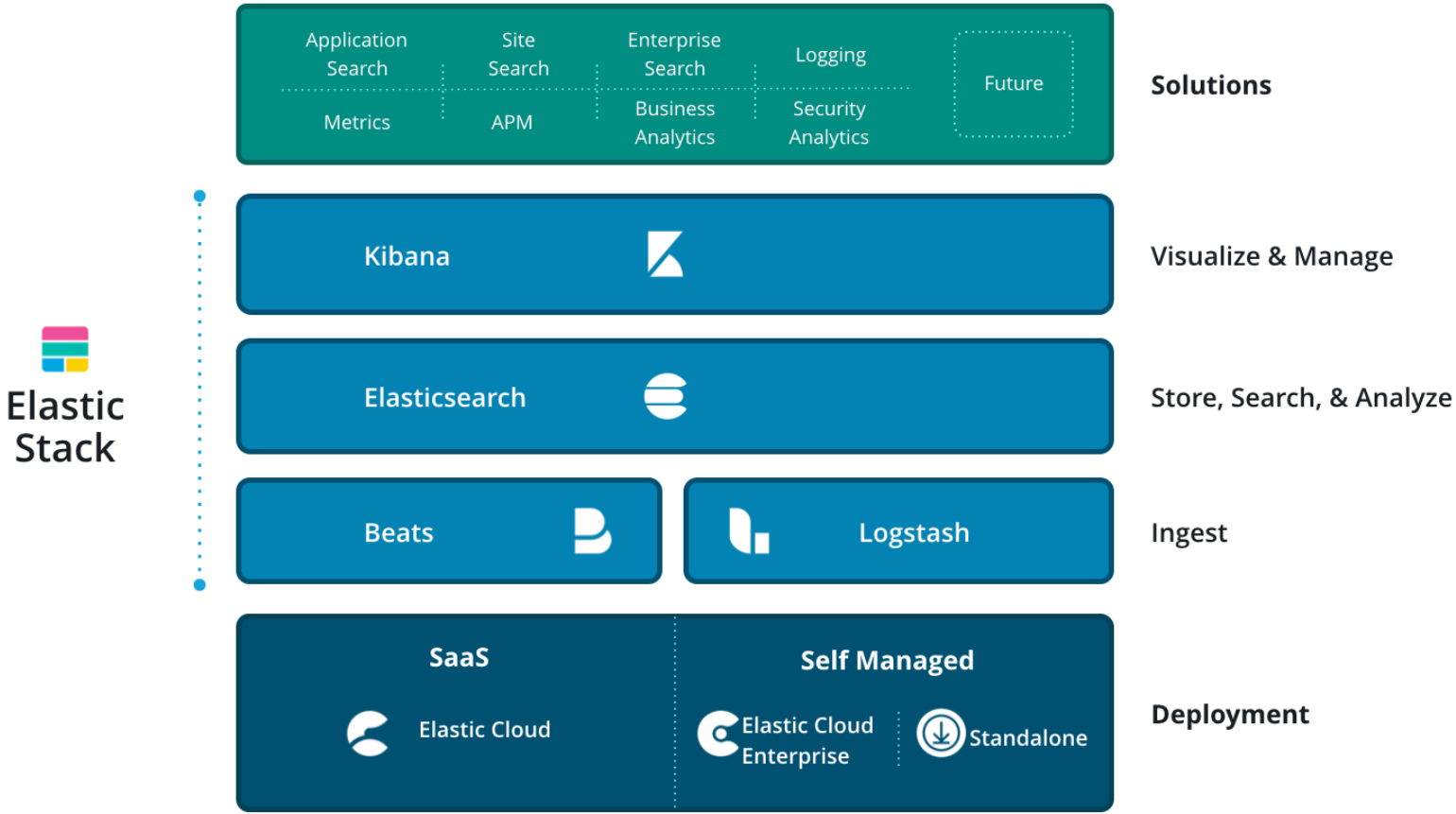
Elasticsearch

- ***Elasticsearch*** is a distributed, RESTful search and analytics engine that centrally stores your data
  - a ***node*** is a single instance of Elasticsearch
  - a ***cluster*** is a collection of one or more nodes

Kibana

- Kibana is an analytics and visualization platform
  - manage the stack
  - search, view and interact with your Elasticsearch data
  - visualize your data in a variety of charts, tables, and maps

Elastic


Application Search
Site Search
Enterprise Search
Logging
Metrics
APM
Business Analytics
Security Analytics
Future
Solutions
Kibana
Visualize & Manage
Elasticsearch
Store, Search, & Analyze
Elastic Stack
Beats
Logstash
Ingest
SaaS
Elastic Cloud
Self Managed
Elastic Cloud Enterprise
Standalone
Deployment

Chapter review

Summary

- ***Elasticsearch*** is designed to be scalable and easy-to-use by any programming language
- The ***Elastic Stack*** is a collection of products with Elasticsearch at the heart
- ***Beats*** are single-purpose data shippers
- ***Logstash*** is a server-side data processing pipeline
- ***Kibana*** is an analytics and visualization platform

Quiz

1. **True or False**: Elasticsearch uses Apache Lucene behind the scenes to index and search data.
2. What were two of Banon's main objectives when designing Elasticsearch?
3. You want to monitor the performance of your servers. How might you collect that data and ingest it into Elasticsearch?
4. You want to monitor all of your networks' traffic. How might you collect that data and ingest it into Elasticsearch?

Lab 1

2. CRUD Operations

Topics

- Getting Data in
- Cheaper in Bulk
- Getting Data Out

Getting Data in

Documents must be JSON Objects

- For example, our products could be in a DB table:

name	price	in_stock	sold
Heineken	249	70	491
Bavaria	219	61	386

- Each product needs to be converted to a JSON object:

JSON consists of ***fields*** (e.g. "**name**" and "**price**") …

```
{
  "name" : "Heineken",
  "price" : 249,
  "in_stock" : 70,
  "sold" : 491
}
```

…and ***values***

Document Store

- Elasticsearch is a distributed ***document store***
  - It can store and return complex data structures that are represented as JSON objects
- A ***document*** is a serialized JSON object that is stored in Elasticsearch under a unique ID

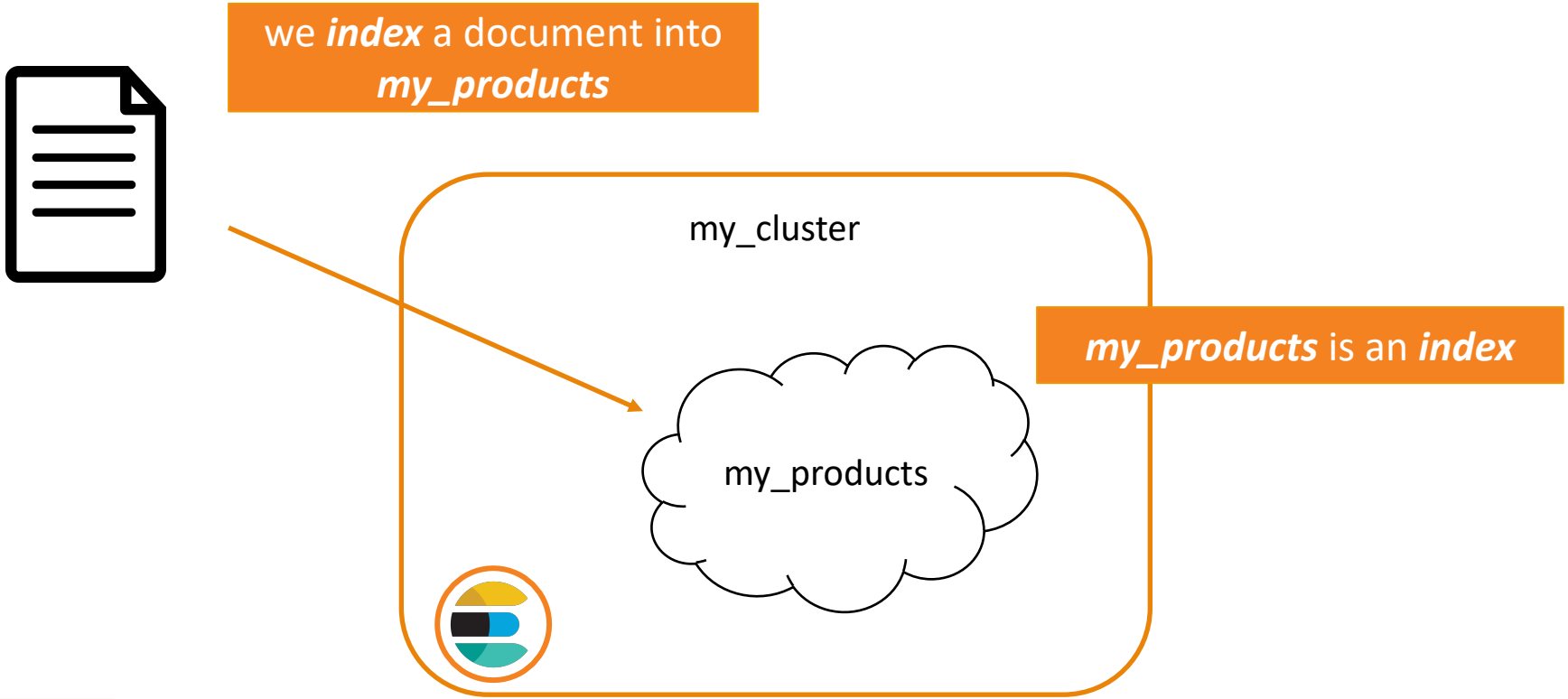
A JSON object...

```
{
  "name" : "Heineken",
  "price" : 249,
  "in_stock" : 70,
  "sold" : 491
}
```

...is stored in Elasticsearch as a document

Documents are indexed into an Index

- In Elasticsearch, a document is ***indexed*** into an ***index***

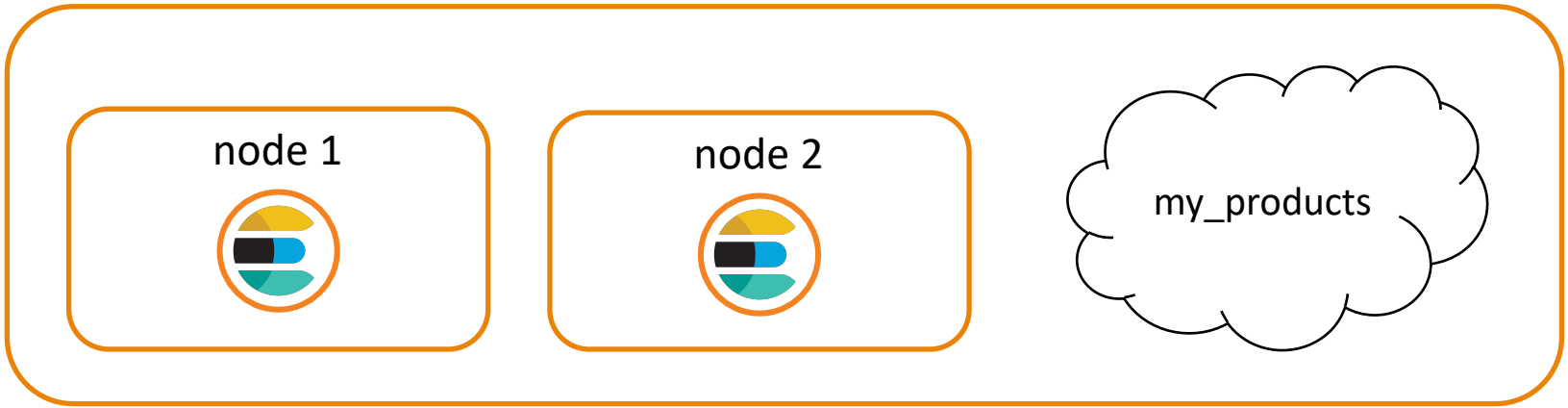


Index

- An ***index*** in Elasticsearch is a ***logical*** way of grouping data
  - an index has a ***mapping*** that defines the fields in the index
  - an index is a ***logical namespace*** that maps to where its contents are stored in the cluster

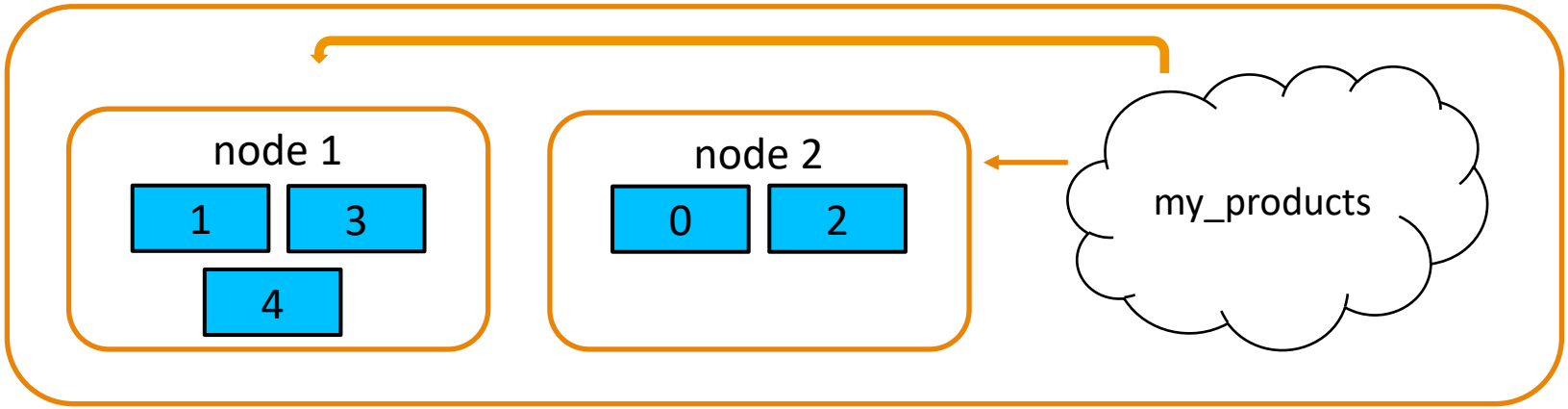
Index

- There are two different concepts in this definition
  1. an index has some type of ***data schema*** mechanism
  2. an index has some type of mechanism to ***distribute data*** across a cluster



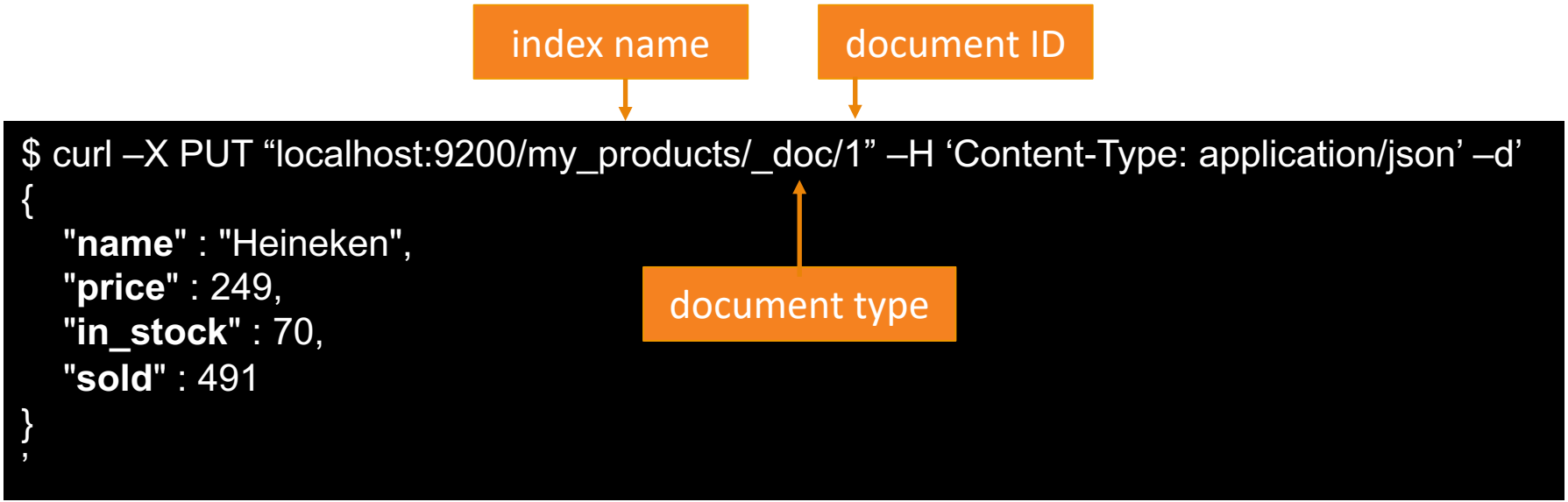
Shards

- Elasticsearch is a ***distributed*** document store
- A ***shard*** is a single piece of an Elasticsearch index
  - Indices are partitioned into shards so they can be distributed across multiple nodes



Index a Document

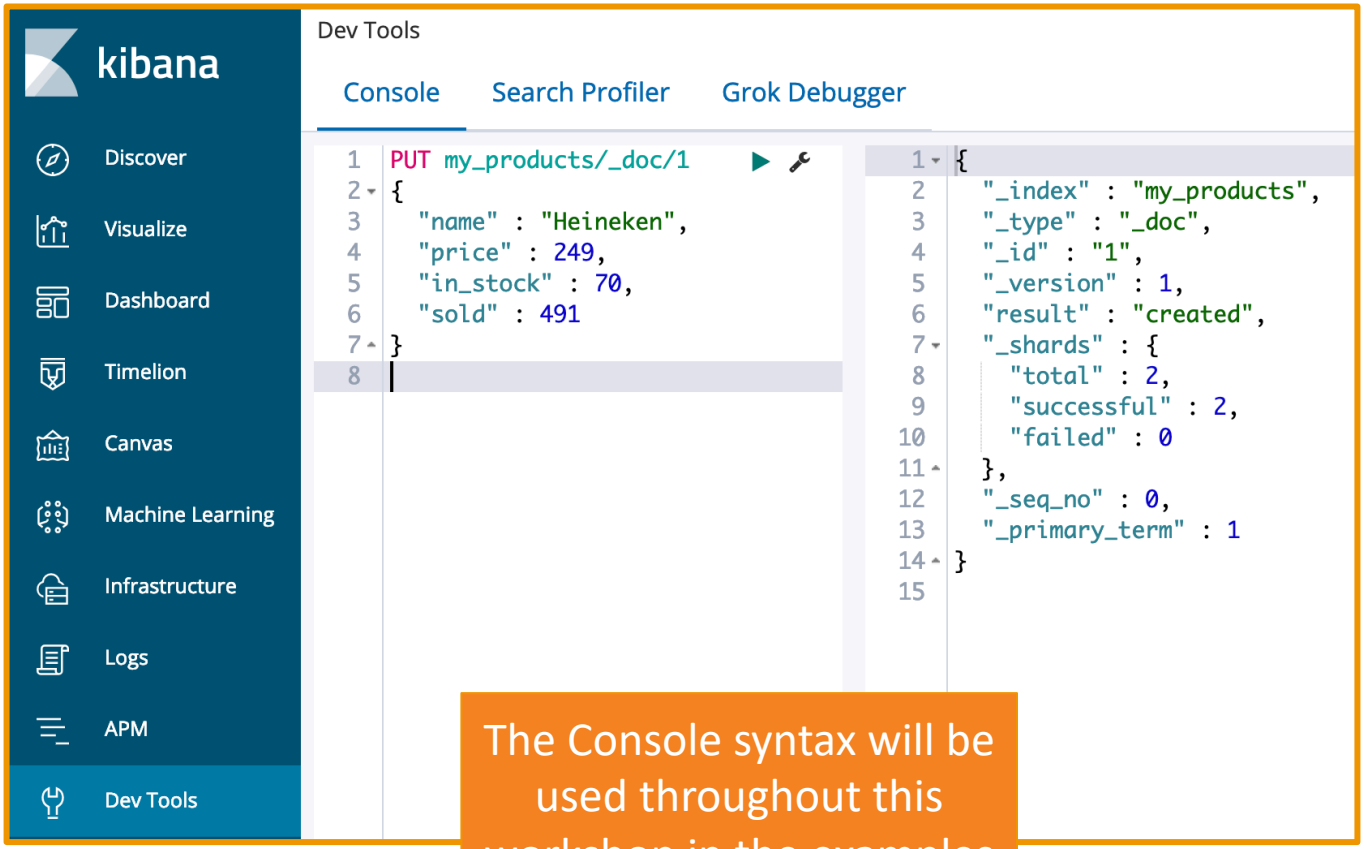
- The ***Index API*** is used to index a document
  - use **PUT** or **POST** and add the document in the body request
  - notice we specify the ***index***, the ***type*** and an ***ID***
  - if no ID is provided, Elasticsearch will generate one



Console

- Using curl all the time can be a bit tedious
- ***Kibana*** has a developer tool named ***Console*** for creating and submitting Elasticsearch requests in a simpler fashion

Console



The Console syntax will be used throughout this workshop in the examples

The Response

But we never created the index!

- As part of the easy-to-use, out-of-the-box experience of Elasticsearch, if you index a document into a non-existing index, ***the index is created automatically***
- You will likely disable this behavior on a production cluster
  - and create your own index with your settings and mappings.

```
PUT my_products
{
  "settings": {
    ...
  },
  "mappings": {
    ...
  }
}
```

We typically create the index first, then index our documents

What if the document ID already exists?

- What do you think happens if you index a document using an ID that already exists?

```
PUT my_products/_doc/1
{
  "name" : "Hertog Jan",
  "price" : 289,
  "in_stock" : 741,
  "sold" : 613
}
```

my_products already has a document with **_id = 1**

What if the document ID already exists?

- The document gets ***reindexed***
  - the new document overwrites the existing one

```
PUT my_products/_doc/1
{
  "name" : "Hertog Jan",
  "price" : 289,
  "in_stock" : 741,
  "sold" : 613
}
```

200 response (instead of **201**)

```
{
  "_index" : "my_products",
  "_type" : "_doc",
  "_id" : "1",
  "_version" : 2,
  "result" : "updated",
  …
}
```

_version is incremented

“**updated**” instead of “**created**”

The _create Endpoint

- If you do **not** want a document to be overwritten if it already exists, use the **_create** endpoint
  - no indexing occurs and returns a **409** error message

```
PUT my_products/_doc/1/_create
{
  "name" : "Hertog Jan",
  "price" : 289,
  "in_stock" : 741,
  "sold" : 613
}
```

Fails if a document with **_id=1** is already indexed

```
{
  ”status": 409,
  "error": {
    "root_cause": [
      {
        "type": "version_conflict_engine_exception",
        "reason": "[_doc][1]: version conflict,
document already exists (current version [2])",
```

The _update Endpoint

- If you want update fields in a document, use the **_update** endpoint
  - Make sure to add the **“doc”** context

update the **“price”** field

```
POST my_products/_doc/1/_update
{
  "doc": {
    "price" : 299
  }
}
```

```
{
  "_index" : "my_products",
  "_type" : "_doc",
  "_id" : "1",
  "_version" : 3,
  "result" : "updated",
  ...
```

Deleting a Document

- Use **DELETE** to delete an indexed document
  - response code is **200** if the document is found, **404** if not

```
DELETE my_products/_doc/1
```

```
{
  "_index" : "my_products",
  "_type" : "_doc",
  "_id" : "1",
  "_version" : 4,
  "result" : "deleted",
  "_shards" : {
    "total" : 2,
    "successful" : 2,
    "failed" : 0
  },
  "_seq_no" : 3,
  "_primary_term" : 1
}
```

Cheaper in Bulk

Cheaper in Bulk

- The ***Bulk API*** makes it possible to perform many write operations in a single API call, greatly increases the indexing speed
  - Useful if you need to index a data stream such as log events, which can be queued up and indexed in batches
- Four actions: **create**, **index, update** and **delete**
- The response is a large JSON structure with the individual results of each action that was performed
  - Failure of a single action does not affect remaining actions

Example of _bulk

- **_bulk** has a unique syntax based on lines of commands
  - it uses the newline delimited JSON (NDJSON) structure
  - Each command appears on a single line

```
POST comments/_doc/_bulk
{"index" : {"_id":3}}
{"title": "Tuning Go Apps with Metricbeat", "category": "Engineering"}
{"index" : {"_id":4}}
{"title": "Elastic Stack 6.1.0 Released", "category": "Releases"}
{"index" : {"_id":5}}
{"title": "Searching for needle in", "category": "User Stories"}
{"update" : {"_id":5}}
{"doc": {"title": "Searching for needle in haystack"}}
{"delete": {"_id":4}}
```

The "**index**" action is followed by the document (on a single line)

...except for "**delete**", which is not followed by a document

Getting Data Out

Retrieving a Document

- Use **GET** to retrieve an indexed document
  - Notice we specify the ***index***, the ***type*** and an ***ID***
  - response code is **200** if the document is found, **404** if not


```
GET my_products/_doc/1
```

```
{
  "_index" : "my_products",
  "_type" : "_doc",
  "_id" : "1",
  "_version" : 3,
  "_seq_no" : 6,
  "_primary_term" : 1,
  "found" : true,
  "_source" : {
    "name" : "Hertog Jan",
    "price" : 299,
    "in_stock" : 741,
    "sold" : 613
  }
}
```

The original document is returned in the **_source** field

Retrieving Multiple Documents

- The **_mget** endpoint allows you to **GET** multiple documents in a single request

```
GET _mget
{
  "docs": [
    {
      "_index": "my_products",
      "_type": "_doc",
      "_id": 3
    },
    {
      "_index": "my_products",
      "_type": "_doc",
      "_id": "F1oSa3EBOOytQ5FTHpaE"
    }
  ]
}
```

Use the **_mget** endpoint

"**docs**" is an array of documents to GET

A Simple Search

- Use a **GET** request sent to the **_search** endpoint
  - every document is a ***hit*** for this search
  - by default, Elasticsearch returns 10 hits

A simple search


```
{
  "took" : 5,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : 2,
    "max_score" : 1.0,
    "hits" : [
       ...
    ]
  }
}
```

took = the number of milliseconds it took to process the query

total = the number of documents that were hits for this query

hits = array containing the documents that hit the search criteria

Chapter Review

CRUD Operations

Index	`PUT my_products/_doc/1`
	`{`
	`  "name" : "Hertog Jan",`
	`  "price" : 289`
	`}`

Create	`PUT my_products/_doc/1/_create`
	`{`
	`  "name" : "Hertog Jan",`
	`  "price" : 289`
	`}`

Read	`GET my_products/_doc/1`

Update	`POST my_products/_doc/1/_update`
	`{`
	`  "doc": {`
	`  "price" : 299`
	`  }`
	`}`

Delete	`DELETE my_products/_doc/1`
Search	`GET my_products/_search`

Summary

- A ***document*** is a serialized JSON object that is stored in Elasticsearch under a unique ID
- An ***index*** in Elasticsearch is a ***logical*** way of grouping data
- The ***Bulk API*** makes it possible to perform many write operations in a single API call, greatly increases the indexing speed

Quiz

1. How is an index distributed across a cluster?
2. **True or False**: If an index does not exist when indexing a document, Elasticsearch will automatically create the index for you.
3. What is the ***default number of hits*** *that a query returns?*
4. What happens if you index a document and the **_id** already appears in the index?
5. **True or False**: Using the Bulk API is more efficient than sending multiple, separate requests.

Lab 2

3. Querying Data

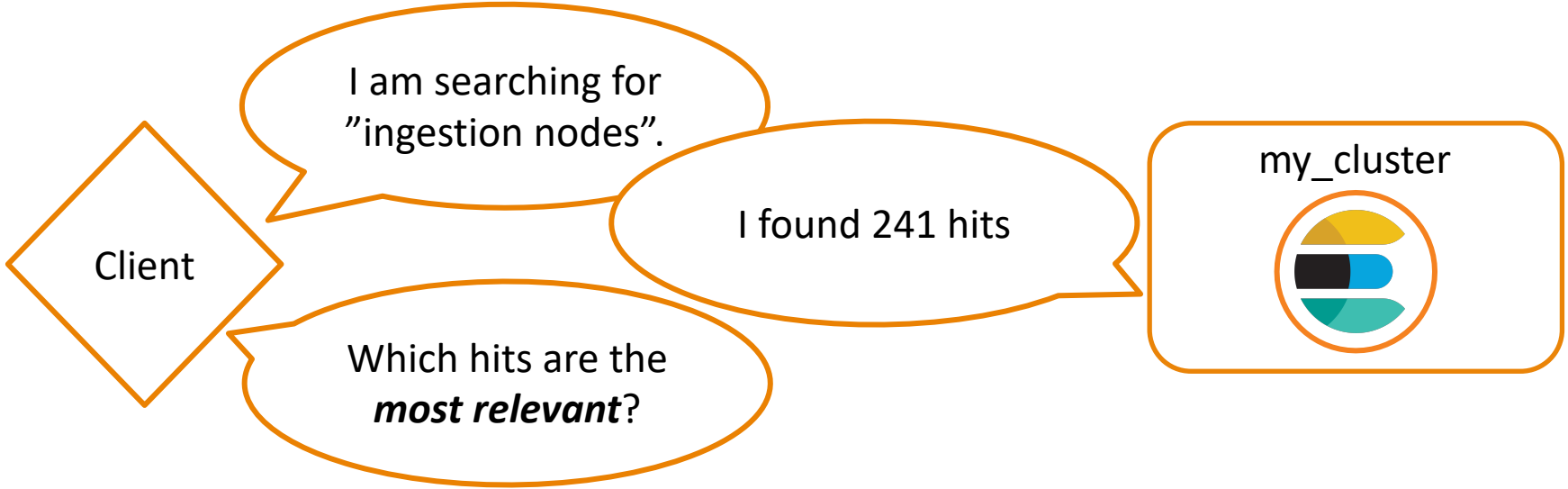
Topics

- Relevance
- Searching for Terms
- Scoring
- Searching for Phrases
- Searching within Date Ranges
- Combining Searches
- Filtering Searches

Relevance

Relevance

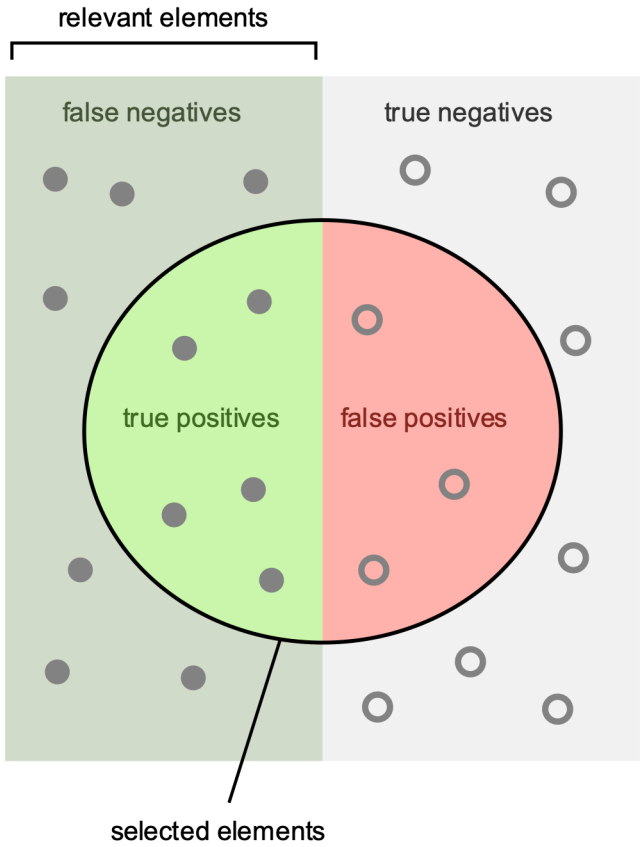
- When querying data, we are interested in the most ***relevant*** documents
  - Are you able to find all the documents you are looking for?
  - How many irrelevant documents are returned?
  - How well are the documents ranked?



Measuring Relevance

- Precision
  - Did we get irrelevant results in addition to the relevant results?
- Recall
  - Did we miss any relevant results that were not returned?
- Ranking
  - Are the results ordered from most relevant at the top, to least relevant at the bottom?
- Elasticsearch uses a ***score*** to determine the ranking of the matching documents

Precision and Recall



How many selected items are relevant?

Precision = 

How many relevant items are selected?

Recall = 

Improving Precision and Recall

- **Recall** can be improved by “widening the net”:
  - Using queries that also return ***partial*** or ***similar*** matches
  - … *but makes precision worse*
- **Precision** can be improved by making searchers more strict:
  - Using queries that only return **exact** matches
  - … *but makes recall worse*
- We will see many queries and options that either improve recall or precision

Searching for Terms

Searching Documents

- Suppose we are interested in learning about ***"ingest nodes"***:

Blog Title	Relevant?
Ingest Node: A client's Perspective	yes
A New Way to Ingest – Part 1	yes
Ingest Node to Logstash Config Converter	kind of
Monitoring Kafka with Elastic Stack: Filebeat	kind of
Elasticsearch for Apache Hadoop 5.0.0-beta1	no
Logstash 6.0.0 GA Released	no

Even though the third line looks like a close match, it is not very relevant. Using the DSL we can improve our ranking and relevancy.

Query String vs. Query DSL

- ***Query string*** is a simplified approach to search documents
  - can be used in the URL
  - simple and easy-to-use
  - but hard to write complex queries
  - unforgiving for small typos (quotes, parentheses, ...)
- ***Query DSL*** (Domain Specific Language) allows you to write queries in a JSON format
  - must send a body
  - exposes the entire collection of Elasticsearch APIs
  - very powerful

The match Query

- The ***match*** query is a simple but powerful search for fields that are text, numerical values, or dates
- This is your *"go to"* query
  - Often the starting point of your searches

The field you want to search

```
GET my_products/_search
{
  "query": {
    "match": {
      "FIELD": "TEXT"
    }
  }
}
```

The text you want to search for

Let's Search for Terms

- Let's search for "***ingest nodes***" in the "**content**" field
  - What do you think is required of a document to be a hit?

```
GET blogs/_search
{
  "query": {
    "match": {
      "content": "ingest nodes"
    }
  }
}
```

Query DSL

```
GET blogs/_search?q=content:(ingest nodes)
```

Query string

Match Uses *or* Logic

- By default, the **match** query uses ***"or"*** logic if multiple terms appear in the search query
  - Any document with the term "ingest" ***or*** "nodes" in the **"content"** field will be a hit

```
GET blogs/_search
{
  "query": {
    "match": {
      "content": "ingest nodes"
    }
  }
}
```

*Find blogs that mention "ingest" **or** "nodes"*

Query Response

- By default, the documents are returned sorted by **_score**


```
"hits": {
  "total": 241,
  "max_score": 9.3005905,
  "hits": [
    {
      "_index": "blogs",
      "_type": "_doc",
      "_id": "7ItcB2cBZWSWiUi4z_dX",
      "_score": 9.3005905,
      "_source": {
        "title": "A New Way To Ingest - Part 2"
      }
    },
    {
      "_index": "blogs",
      "_type": "_doc",
      "_id": "2otcB2cBZWSWiUi41Pkb",
      "_score": 8.616863,
      "_source": {
        "title": "Ingest Node: A Client's Perspective"
      }
    },
...
```

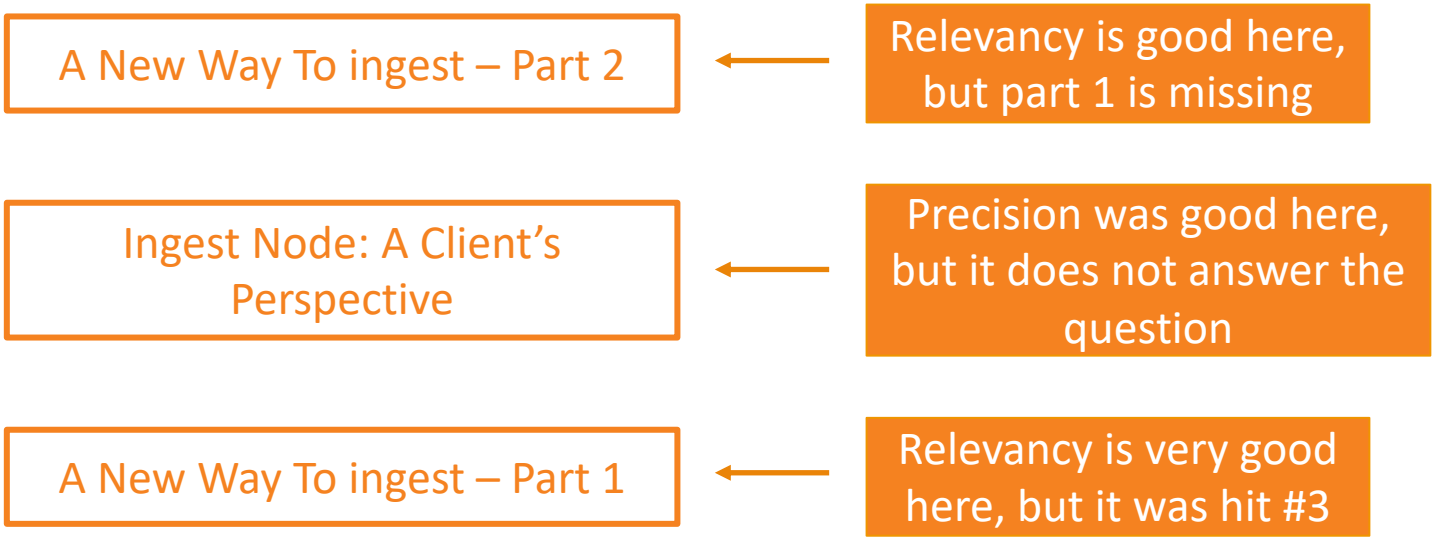

241 documents were a match

Each hit has a **_score**

Results are sorted by **_score** (by default)

Analyze the Hits

- Notice the ***"or"*** logic led to some hits that we may not have been interested in:



How can we increase precision?

- The *"or"* logic can be changed to *"and"* in a **match** query using the **operator** parameter
  - Notice the slightly different syntax for **match**

```
GET blogs/_search
{
  "query": {
    "match": {
      "content": {
        "query": "ingest nodes",
        "operator": "and"
      }
    }
  }
}
```

Precision is improved and recall is decreased (only 25 hits now)

Changed "**operator**" to "**and**"

```
GET blogs/_search?q=content:(ingest AND nodes)
```

Let's Add More Terms

- Let's try a search with three terms

The "or" logic gives us 690 hits

```
GET blogs/_search
{
  "query": {
    "match": {
      "content": "ingest nodes logstash"
    }
  }
}
```

```
GET blogs/_search?q=content:(ingest nodes logstash)
```

The minimum_should_match Parameter

- If a **match** query searches on multiple terms, the "**or**" or "**and**" options might be too wide or too strict
  - Use the **minimum_should_match** parameter to trim the long tail of less relevant results

```
GET blogs/_search
{
  "query": {
    "match": {
      "content": {
        "query": "ingest nodes logstash",
        "minimum_should_match": 2
      }
    }
  }
}
```

Only 82 hits this time

2 of the 3 terms need to match

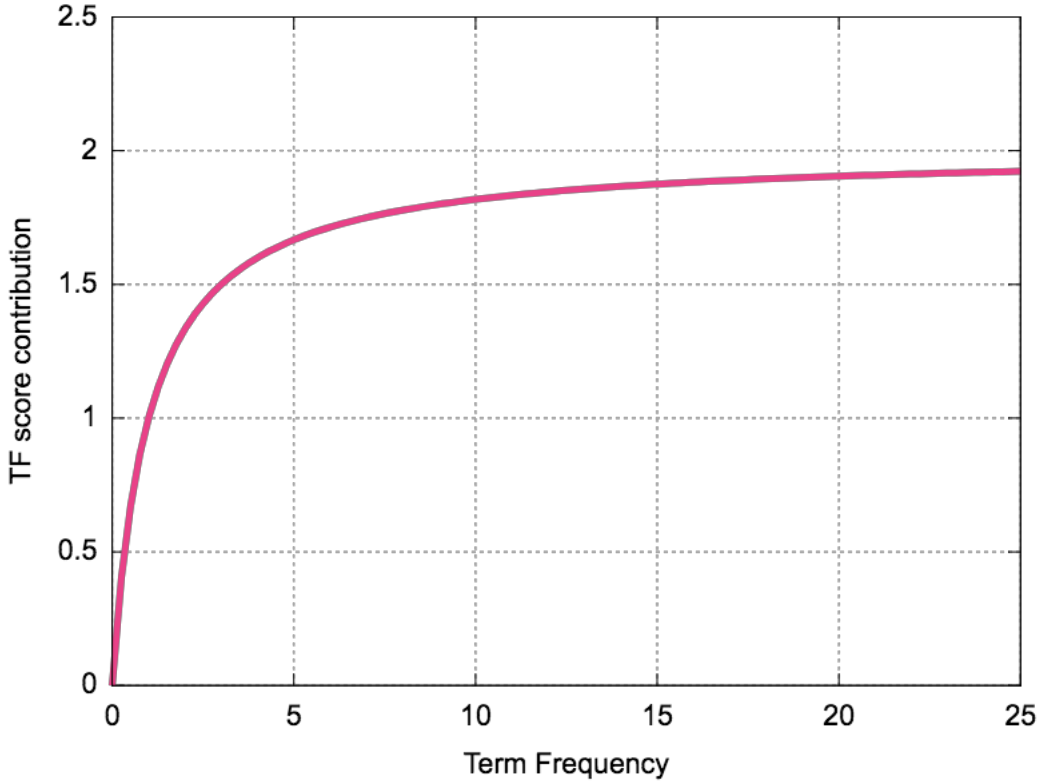
Scoring

Score!

- The **_score** is a value representing how relevant the document is in regards to that specific query
  - A **_score** is computed for each document that is a hit
- Elasticsearch's default scoring algorithm is ***BM25***
- There are three main factors of a document's score:
  - ***TF (term frequency)***: The more a term appears in a field, the more important it is
  - ***IDF (inverse document frequency)***: The more documents that contain the term, the less important the term is
  - ***Field length***: shorter fields are more likely to be relevant than longer fields

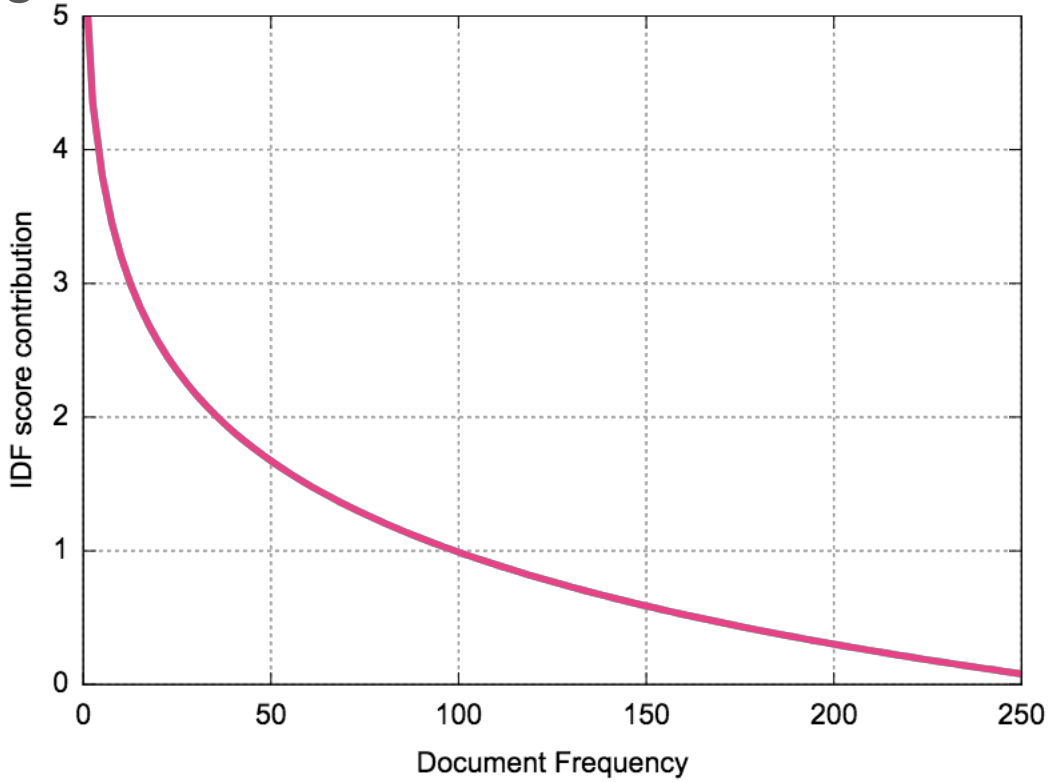
Term Frequency

- The more a term appears in a field, the more important it is



Inverse Document Frequency

- The more documents contain the term, the less important it is



Searching for Phrases

Searching for Phrases

- The match query did an OK job of searching for "***ingest nodes***"
  - but the order of terms was not taken into account:

Elastic Ingest Node: A Client's Perspective

Elasticsearch for Apache Hadoop 5.0.0-beta1

A New Way To ingest – Part 1

Monitoring Kafka with Elastic Stack: Filebeat

A New Way To ingest – Part 2

Blogs that often mention "**ingest**" scored high, even though they might not mention "**ingest nodes**"

The match_phrase Query

- The match_phrase query is for searching text when you want to find terms that are near each other
  - All the terms in the phrase must be in the document
  - The position of the terms must be in the same relative order
- A match_phrase query can be quite expensive! Use them wisely

The field you want to search

```
GET blogs/_search
{
  "query": {
    "match_phrase": {
      "FIELD": "PHRASE"
    }
  }
}
```

The phrase you want to search for

Example of match_phrase

- Let's try the ***"ingest nodes"*** search using **match_phrase** instead of **match**:

```
GET blogs/_search
{
  "query": {
    "match_phrase": {
      "content": "ingest nodes"
    }
  }
}
```

Two things must happen for **"ingest nodes"** to cause a hit:

1. "**ingest**" and "**nodes**" must appear in the **"content"** field
2. The terms must appear in that order

```
GET blogs/_search?q=content:("ingest nodes")
```

Example of match_phrase

Only 8 hits this time

A New Way To ingest – Part 1

A New Way To ingest – Part 2

Logstash 6.0.0 GA Released

Elastic Ingest Node: A Client's Perspective

Another Example

- Let's try a different search
  - Suppose we are interested in blogs about **"open data"**:

```
GET blogs/_search
{
  "query": {
    "match_phrase": {
      "content": "open data"
    }
  }
}
```

Only 5 hits

- Open data at the local government level may …
- files that can be sources from Meteo France's open data platform..
- Publicly available at the European Union Open Data Portal
- Open data at the local government level may seem "smaller"
- … improved your applications with (open) data

```
GET blogs/_search?q=content:"open data"
```

The slop Parameter

- If you want to increase the recall of a match_phrase, you can introduce some flexibility into the phrase
  - The **slop** parameter tells how far apart terms are allowed to be while still considering the document a match

```
GET blogs/_search
{
  "query": {
    "match_phrase": {
      "content": {
        "query": "open data",
        "slop": 1
      }
    }
  }
}
```

9 hits time time!

Open data at the local government level may …

…

building lightweight, open source data shippers

The terms in the phrase can be transposed now

```
GET blogs/_search?q=content:"open data"~1
```

Searching within Date Ranges

The range Query on Dates

- Range queries on **date** fields
  - You can specify the dates as a string
  - Or use a special syntax called ***date math***

```
GET blogs/_search
{
  "query": {
    "range": {
      "publish_date": {
        "gte": "2017-12-01",
        "lt": "2018-01-01"
      }
    }
  }
}
```

Date Math

- Elasticsearch has a user-friendly way to express date ranges by using ***date math***

```
GET blogs/_search
{
  "query": {
    "range": {
      "publish_date": {
        "gte": ”now-3M”
      }
    }
  }
}
```

“I am looking for blogs from the last three months.”

The date math expression for *“right now minus three months”*

Date Math Expressions

- Here are the supported time units for date math and a few examples

y	***years***
M	***months***
w	***weeks***
d	***days***
h or H	***hours***
m	***minutes***
s	***seconds***

If now = 2019-03-29T11:56:22	
now-1h	2019-03-29T10:56:22
now+1d	2019-03-30T11:56:22
now+1h+30m	2019-03-29T13:26:22
2018-01-15||+1M	Jan 15, 2018, plus 1 month

Combining Searches

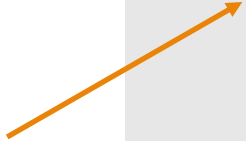
Combining Searches

- Suppose we want to write the following query:
  - Find blogs about the “Elastic Stack” published before 2017
- This search is actually a combination of two queries:
  - We need “Elastic Stack” in the content or title field,
  - and the publish_date field needs to be before 2017
- How can we combine these two queries?
  - By using Boolean logic and the bool query…

Bool Query

- Each of the following clauses is possible (but optional) in a **bool** query
  - And they can appear in any order

Notice the JSON array syntax. You can specify multiple queries in each clause if desired.



```
GET my_index/_search
{
  "query": {
    "bool": {
      "must": [
        {}
      ],
      "must_not": [
        {}
      ],
      "should": [
        {}
      ],
      "filter": [
        {}
      ]
    }
  }
}
```

The must Clause

- Let's search for "**logstash**" only in the "**engineering**" category:


```
GET blogs/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "content": "logstash"
          }
        },
        {
          "match": {
            "category": "engineering"
          }
        }
      ]
    }
  }
}
```

The must_not Clause

- Our previous query does not cast a wide net for “logstash”
  - It might be better to **not** search for blogs about “releases”

449 hits

```
GET blogs/_search
{
  "query": {
    "bool": {
      "must": {
        "match": {
          "content": "logstash"
        }
      },
      "must_not": {
        "match": {
          "category": "releases"
        }
      }
    }
  }
}
```

The should Clause

- A match in a **“should”** clause increases the **_score** of hits
  - A very useful behavior that has a lot of use cases

```
GET blogs/_search
{
  "query": {
    "bool": {
      "must": {
        "match_phrase": {
          "content": "elastic stack"
        }
      },
      "should": {
        "match_phrase": {
          "author": "shay banon"
        }
      }
    }
  }
}
```

*”I am looking for blogs about the **Elastic Stack**, preferably from Shay Banon.”*

This particular **“should”** clause does not add more hits, but blogs from **Shay Banon** score the highest

The filter Clause

- Let's answer the question we posed earlier:
  - ***Find blogs about the "Elastic Stack" published before 2017***

98 hits

The filter clause does not affect the **_score**

```
GET blogs/_search
{
  "query": {
    "bool": {
      "must": {
        "match_phrase": {
          "content": "elastic stack"
        }
      },
      "filter": {
        "range": {
          "publish_date": {
            "lt": "2017-01-01"
          }
        }
      }
    }
  }
}
```

Query vs. Filter

- We have seen query contexts, but there is another clause known as a ***filter context:***
  - **Query context**: results in a **_score**
  - **Filter context**: results in a **“yes”** or **“no”**

The bool Query

- A ***bool*** query is a combination of one or more of the following Boolean clauses:

Clause	Exclude docs	Scoring
must	Y	Y
must_not	Y	Y
should	N*	Y
filter	Y	N

Chapter review

Summary

- ***Precision*** refers to how many irrelevant results are returned in addition to the relevant results
- ***Recall*** refers to how many relevant results are not returned
- ***Query DSL*** (Domain Specific Language) allows you to write queries in a JSON format
- The **match** query is a simple but powerful search for fields that are text, numerical values, or dates

Summary

- The **match_phrase** query is for searching text when you want to find terms that are near each other
- The **range** query is for finding documents with fields that fall in a given range
- The **bool** query allows you to combine queries using Boolean logic

Quiz

1. Explain the difference between **match** and **match_phrase**.
2. If you want to add some flexibility into **match_phrase**, you can configure a _____ property.
3. How could you improve the precision of a match query that consists of 5 terms?
4. A user searches our blogs for "***scripting***" and checks the box that only displays blogs from the "**Engineering**" category. How would you implement this query?
5. **True or False**: If a document matches a filter clause, the score is increased by a factor of 2.

Lab 3

4. Node Types

Topics

- Node Roles
- Cluster State and Master Nodes
- Data Nodes

Node Roles

Node Roles

- There are several roles a node can have
  - Master eligible
  - Data
  - Ingest
  - Coordinating
  - Machine Learning
- Nodes can take on multiple roles at the same time
  - or they 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:
  - defined on the command line or config file (elasticsearch.yml)

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

Cluster State and Master Nodes

Cluster State

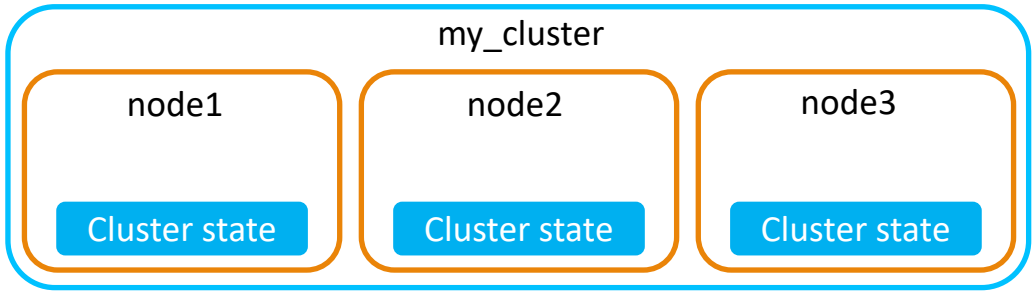
- The details of a cluster are maintained in the cluster state
  - includes the nodes in the cluster, indices, mappings, settings, shard allocation, and so on
- Use the **_cluster** endpoint to view a cluster's state:

```
GET _cluster/state
```


```
{
"cluster_name": " docker-cluster ",
"compressed_size_in_bytes": 13103,
"version": 10,
"state_uuid": "rPiUZXbURICvkPl8GxQXUA",
"master_node": " C0hdYV2XS5iqbe_9LKSrsQ ",
"blocks": {},
"nodes": {...},
"metadata": {...},
"routing_table": {...},
"routing_nodes": {...},
"snapshots": {...},
"restore": {...},
"snapshot_deletions": {...}
}
```

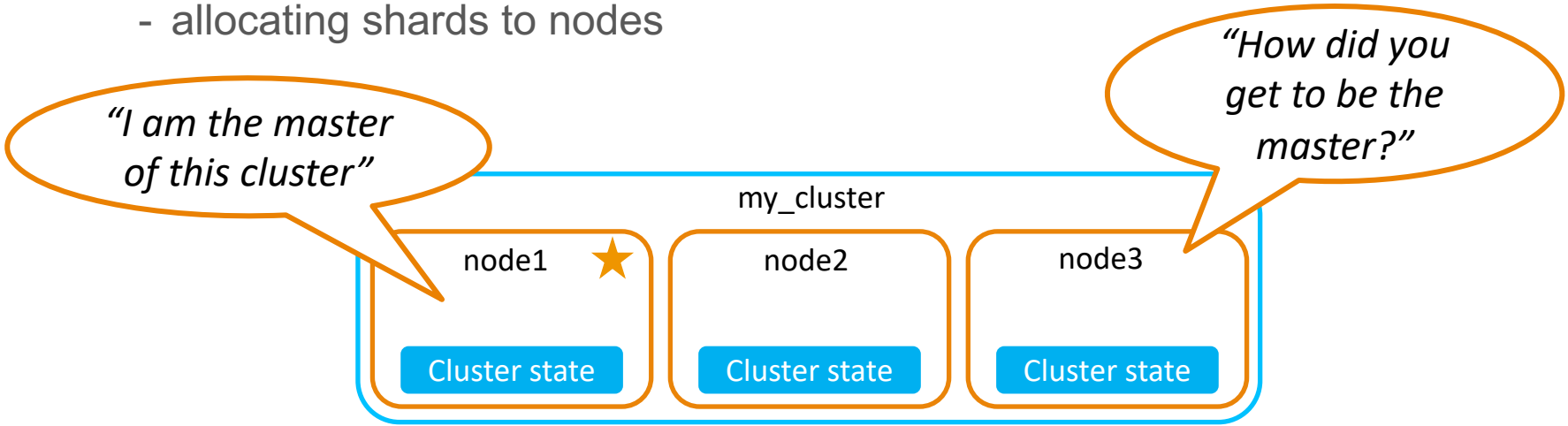
Cluster State

- The cluster state is stored on each node
  - but it ***can only be changed by the master node***



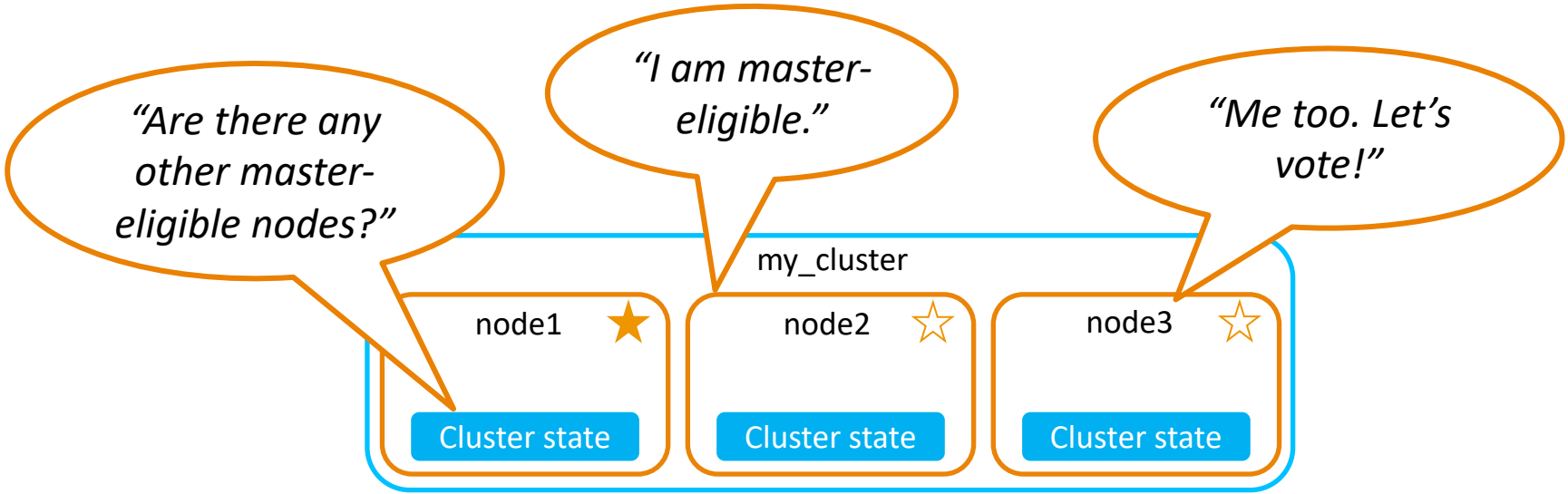
The Master Node

- Every cluster has ***one node*** designated as the ***master***
- The master node is in charge of cluster-wide settings and changes, like:
  - creating or deleting indices
  - adding or removing nodes
  - allocating shards to nodes



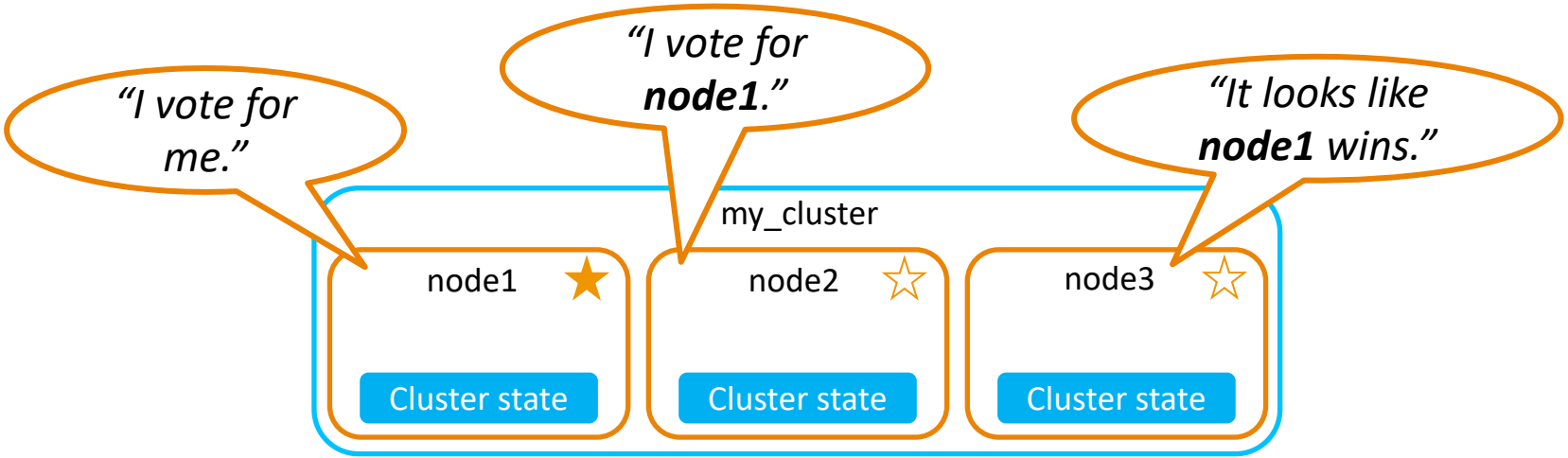
Master-eligible Nodes

- The master node is ***elected*** from the ***master-eligible*** nodes in the cluster
  - a node is master-eligible if **node.master** is set to **true** (the default value)
  - each master-eligible node votes



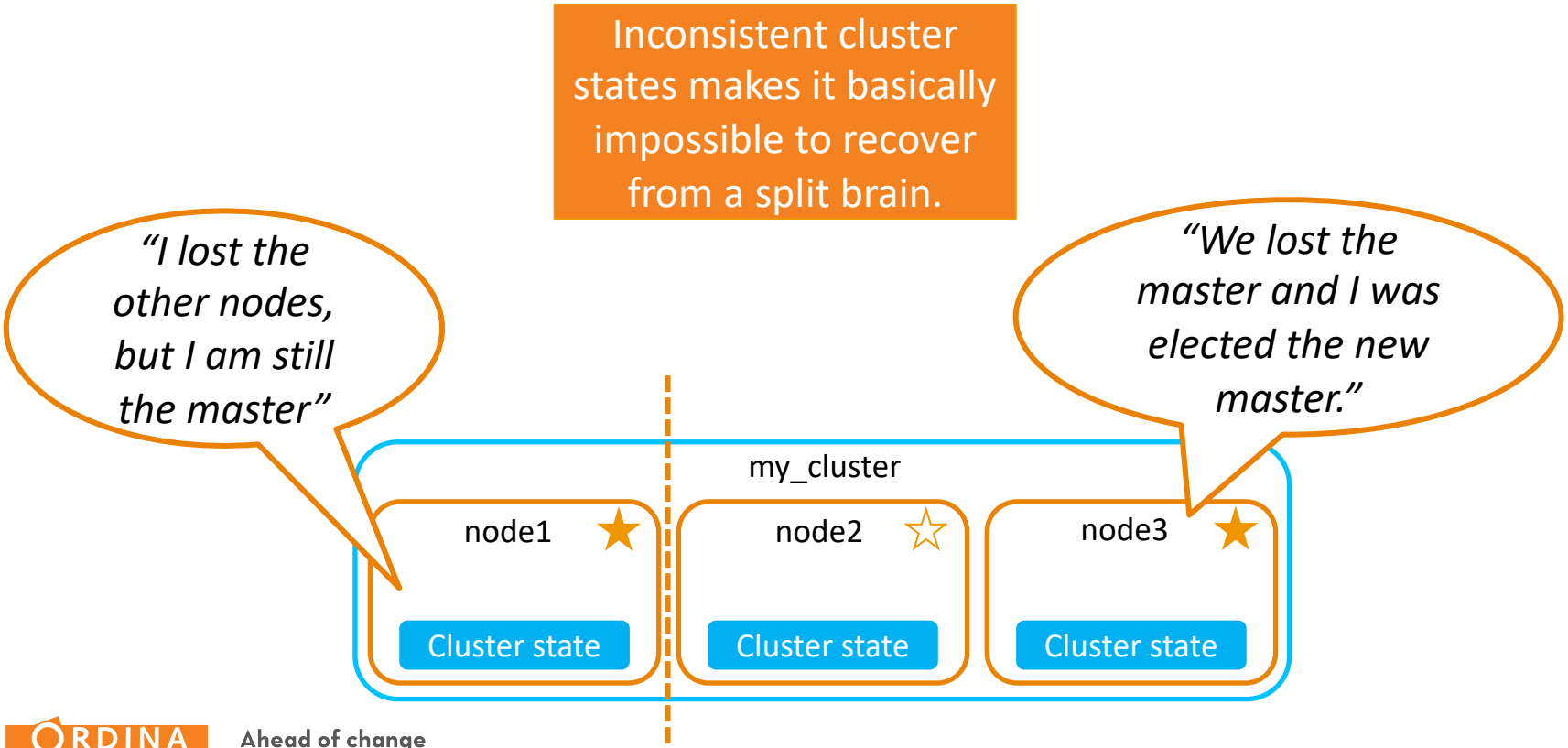
Master-eligible Nodes Election

- The number of votes to win the election is configured with the **discovery.zen.minimum_master_nodes** setting
  - set it to **N/2+1**, where **N** is the number of master-eligible nodes
- Why is it important to have a **quorum**?



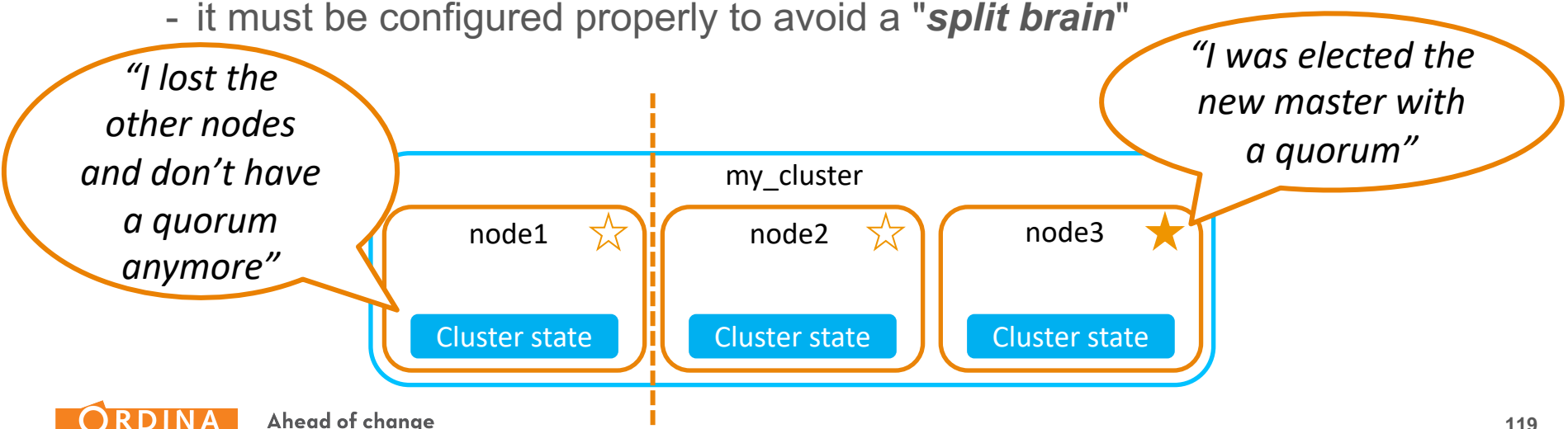
Split Brain

- Two masters can lead to inconsistency:



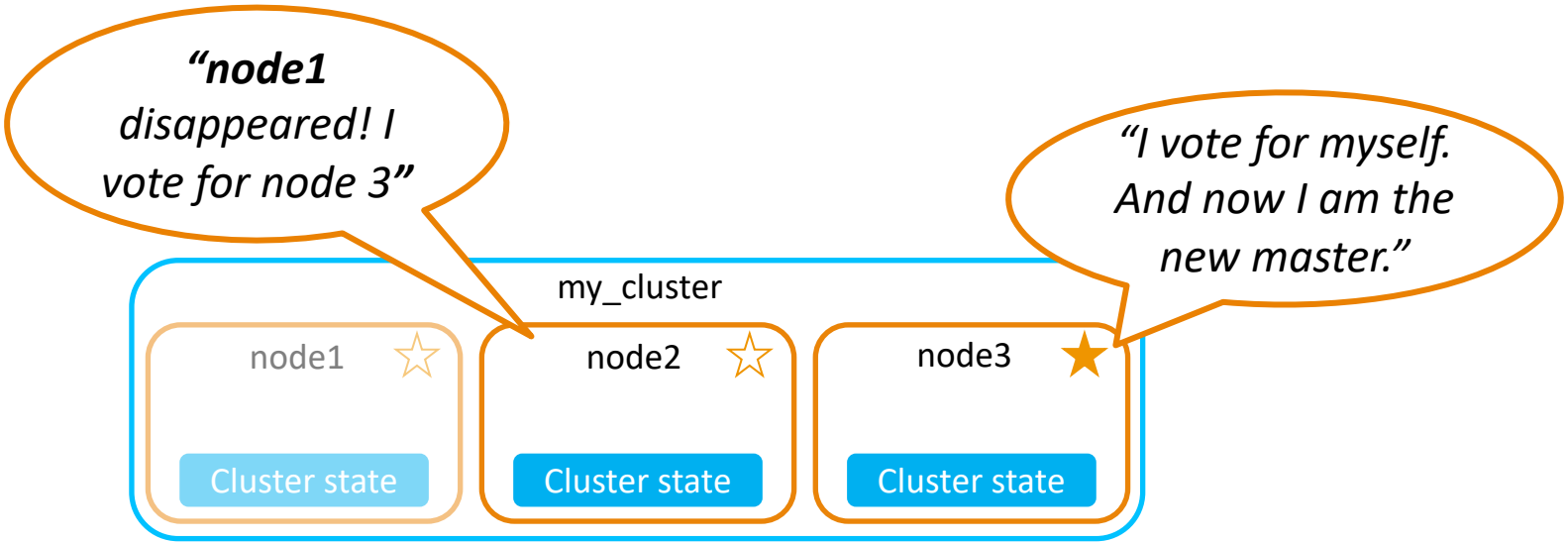
Configuring minimum_master_nodes

- We recommend your production clusters have **3 master-eligible** nodes with **minimum_master_nodes** set to **2**
- Should be defined in the config file (elasticsearch.yml):
  - an ***extremely important setting*** for the stability of your cluster
  - it must be configured properly to avoid a "***split brain***"



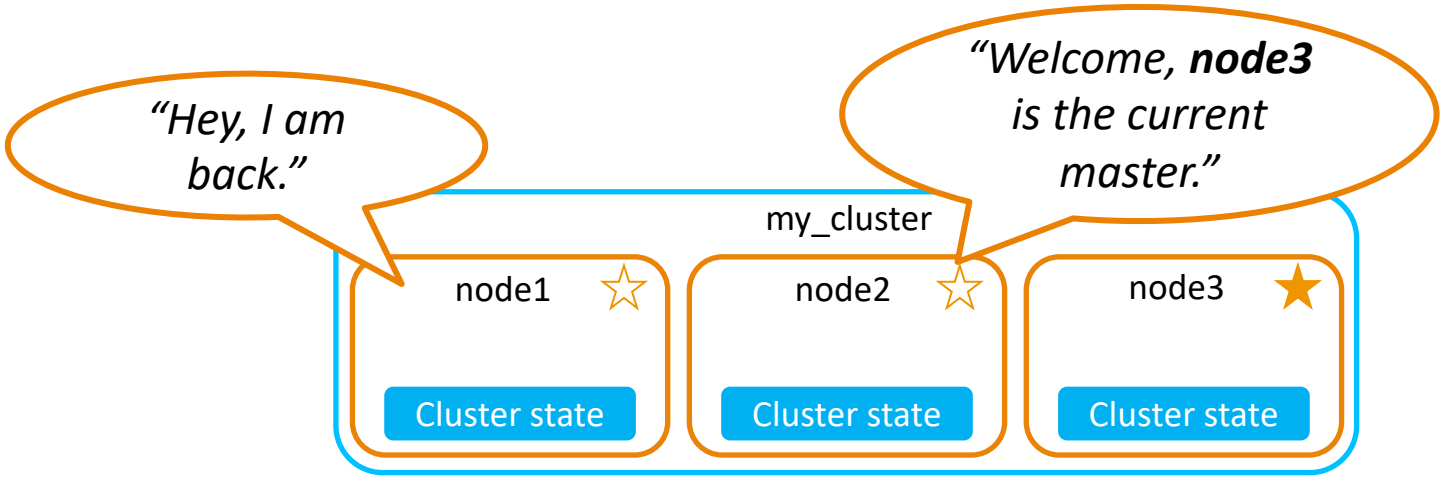
High Availability of the Master Node

- The purpose of having 3 master-eligible nodes is so that if the master node fails, there are 2 backups
  - just enough master-eligible nodes so that a new master can be elected



High Availability of the Master Node

- When node1 comes back online, one of the other master-eligible nodes will be the master
  - so it will simply re-join the cluster as a master-eligible node



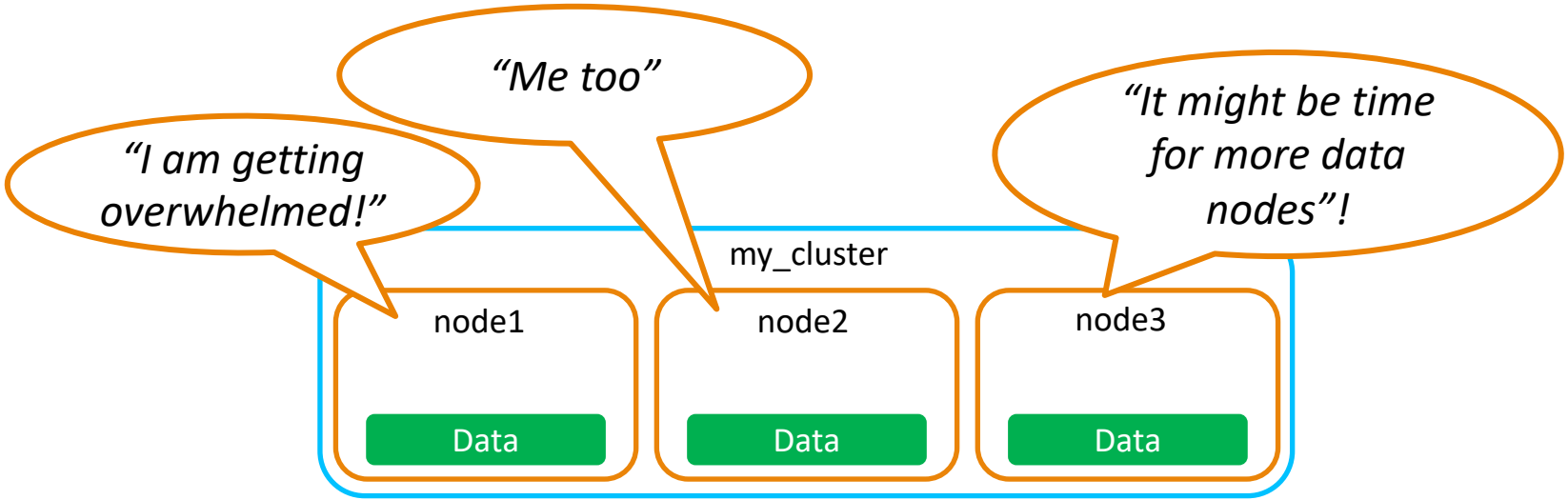
Data Nodes

Data Nodes

- ***Data nodes*** have two main features:
  - They hold the shards that contain the documents you have indexed
  - They execute data related operations like CRUD, search, and aggregations
- All nodes are data nodes by default
  - configured using the **node.data** property

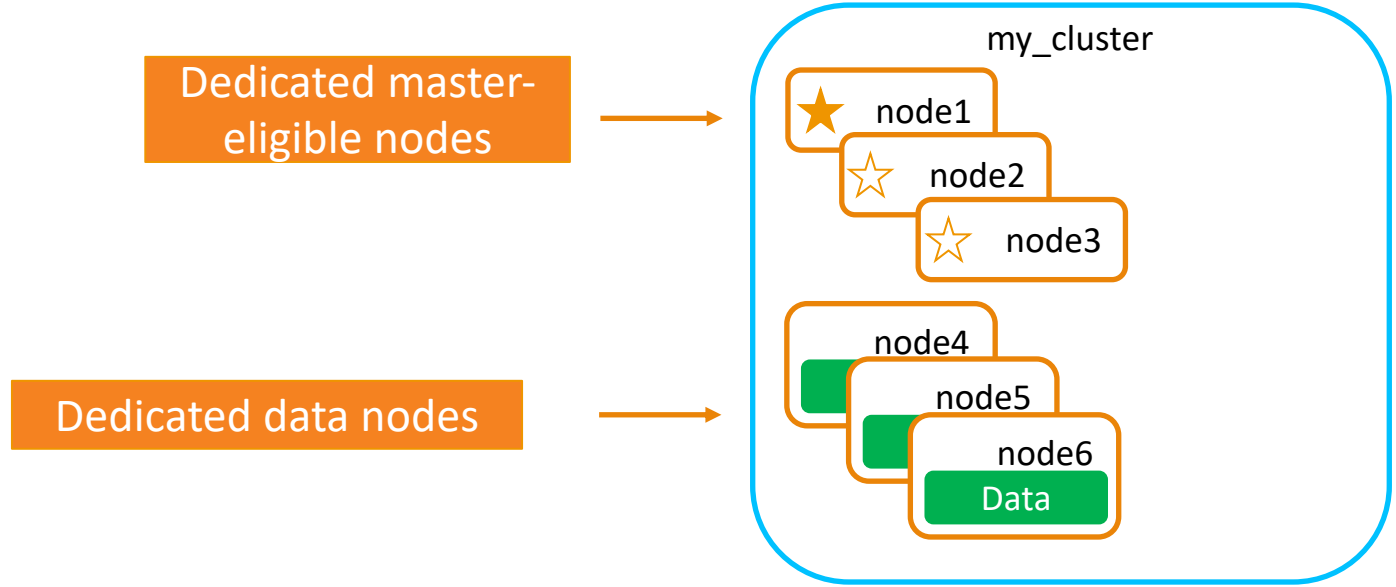
Data Nodes Scale

- Data nodes are I/O, CPU, and memory-intensive
  - It is important to monitor these resources and add more data nodes if they are overloaded



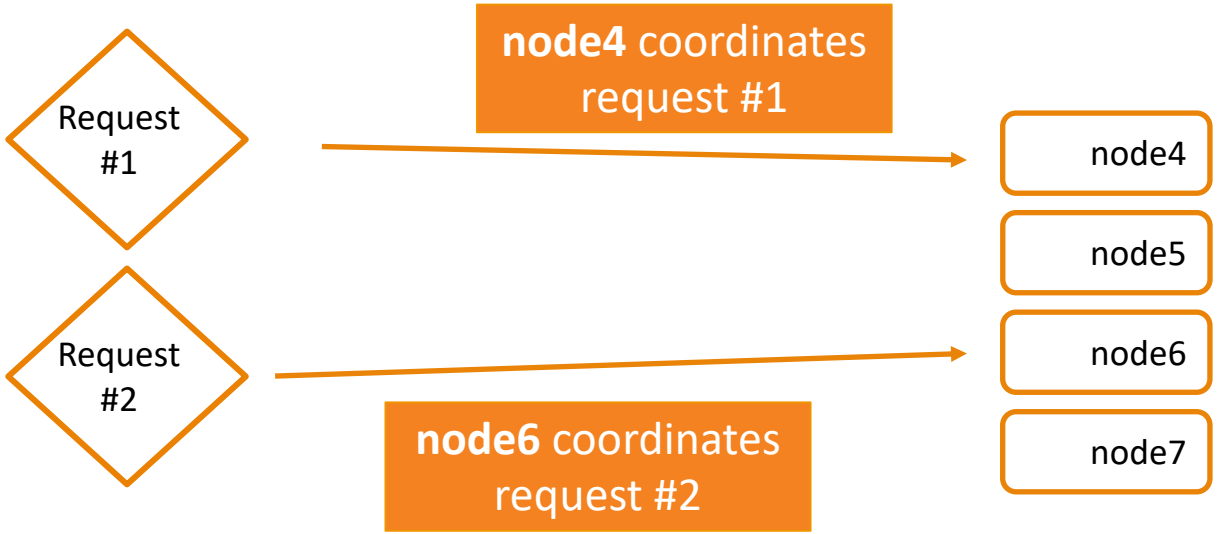
Dedicated Master and Data Nodes

- You can configure a node to be ***just*** a data or master-eligible node
  - by setting **node.master** or **node.data** to **false**
  - provides a nice separation of the master and data roles



Coordinating Node

- A coordinating node is the node that receives and handles a specific client request
  - every node is implicitly a coordinating node
  - forwards the request to other relevant nodes, then combines those results into a single result



Chapter review

Summary

- The details of a cluster are maintained in the ***cluster state***
- Every cluster has ***one node*** designated as the ***master***
- A master-eligible node needs at least **minimum_master_nodes** votes to win an election
- ***Data nodes*** hold shards and execute data-related operations like CRUD, search, and aggregations

Quiz

1. If you have ***three*** master-eligible nodes in your cluster, what should you set ***minimum_master_nodes*** to?
2. If you have **two** master-eligible nodes in your cluster, what should you set ***minimum_master_nodes to***?
3. How would you configure a node to be a ***dedicated master-eligible*** node?

Lab 4

5. Understanding Shards

Topics

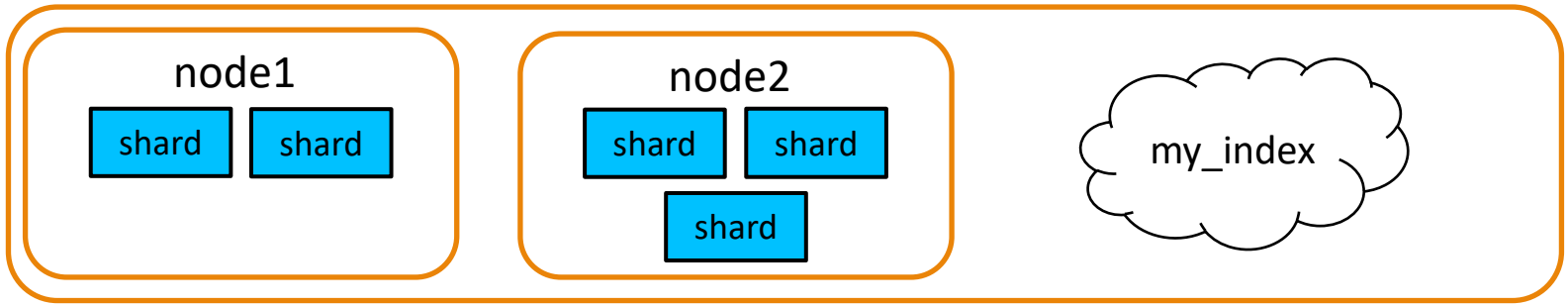
- Understanding Shards
- Anatomy of a Write Operation
- Anatomy of a Search

Understanding Shards

Remember Shards?

- A ***shard*** is a worker unit that holds data and can be assigned to nodes
  - An index is a virtual namespace which points to a number of shards

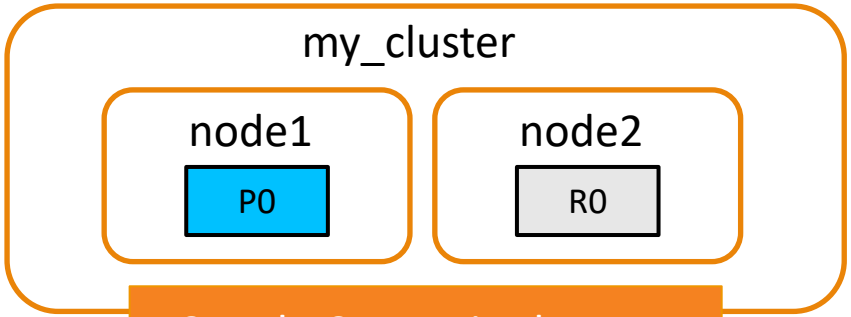
an index is “split” into shards ***before*** any documents are indexed



Primary vs. Replica

- There are two types of shards
  - primary: the original shards of an index
  - replicas: copies of the primary
- Documents are replicated between a primary and its replicas
  - a primary and all replicas are guaranteed to be on different nodes

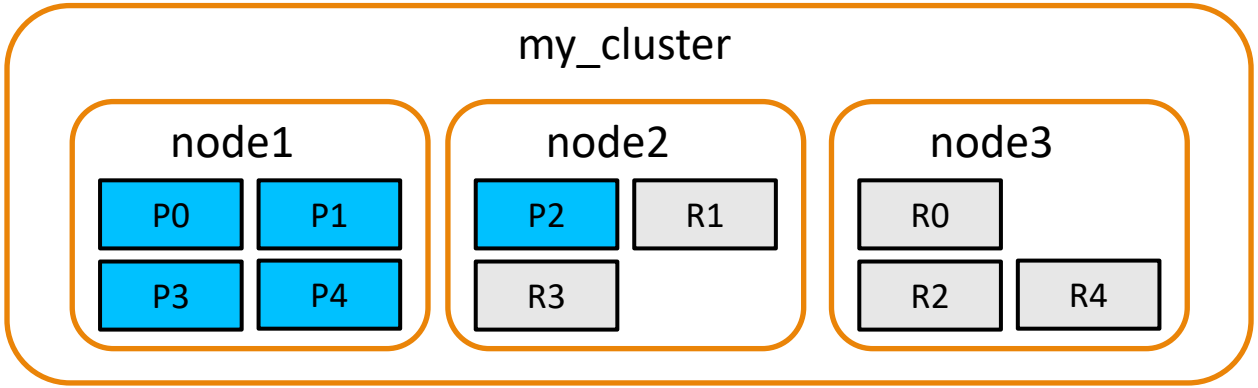
When an index is created, shards are numbered: 0, 1, 2, 3 ...



P0 and **R0** contain the same documents. They are called copies of shard 0

Scaling Elasticsearch

- Adding nodes to our cluster will cause a ***redistribution of shards***
  - and the creation of replicas (if enough nodes exist)
  - the ***master node*** determines the distribution of shards

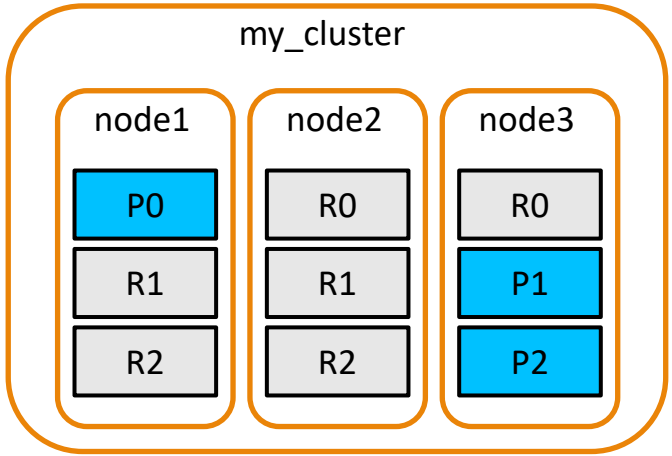


Configuring the number of Shards

- The default number of primary shards for an index is 5
  - You specify the number of primary shards when you create the index
  - It is costly to change the number of shards, so choose wisely!
  - The number of replicas can be changed at any time

```
PUT my_new_index
{
  "settings": {
    "number_of_shards": 3,
    "number_of_replicas": 2
  }
}
```

3 primaries + 2 replicas
=
9 total shards

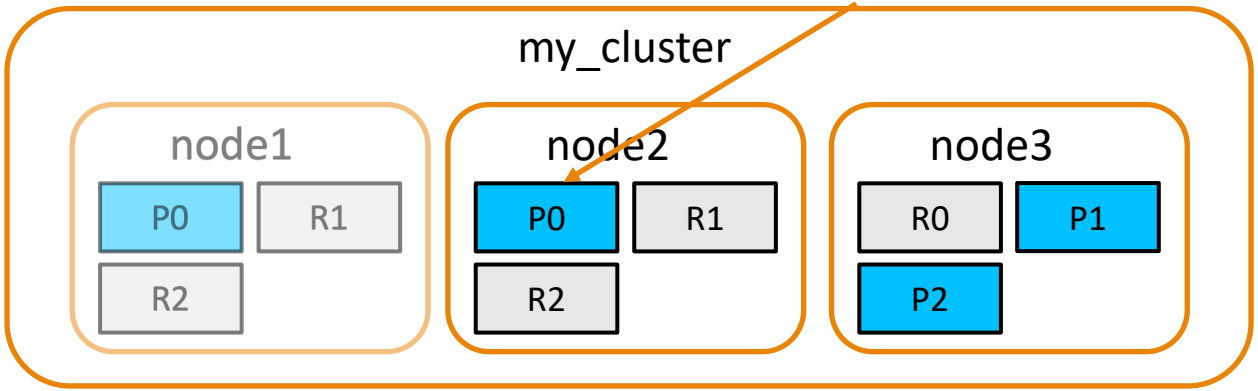


Why create Replicas?

- ***High availability***
  - we can lose a node and still have all the data available
  - replicas are promoted to primaries as needed

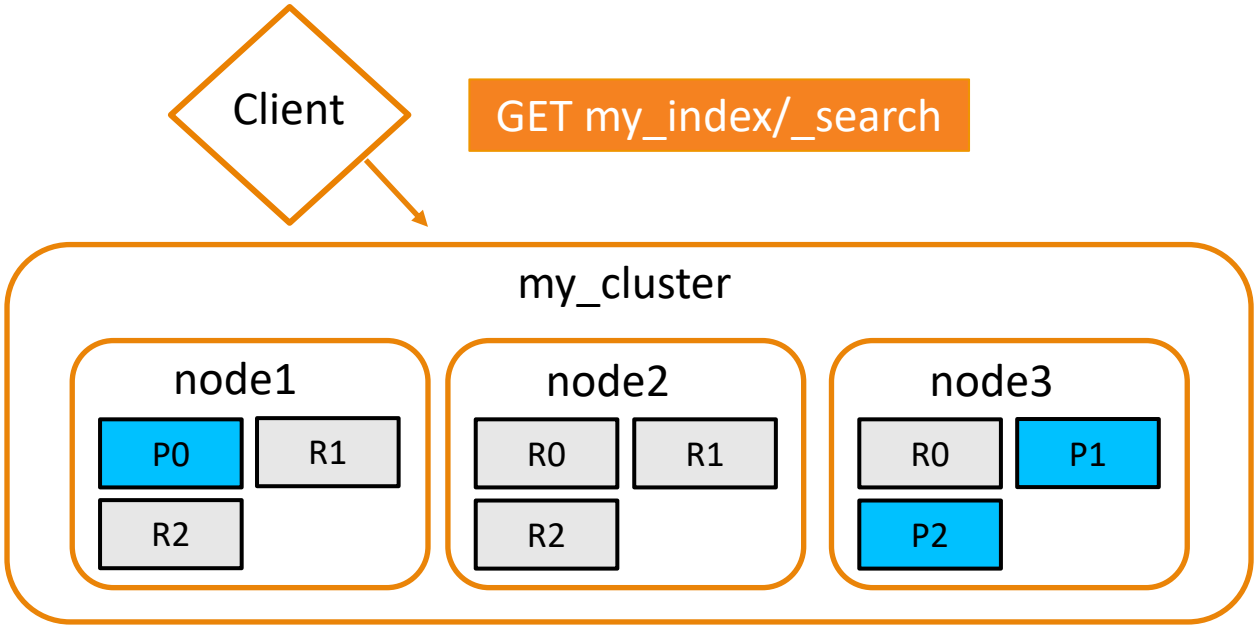
If **node1** fails, its data is still available on other nodes

We need a new primary for **P0**, so **R0** will be ***promoted***



Why create Replicas?

- ***Read throughput***
  - A query can be performed on a primary or replica shard
  - allows you to scale your data and better utilize cluster resources

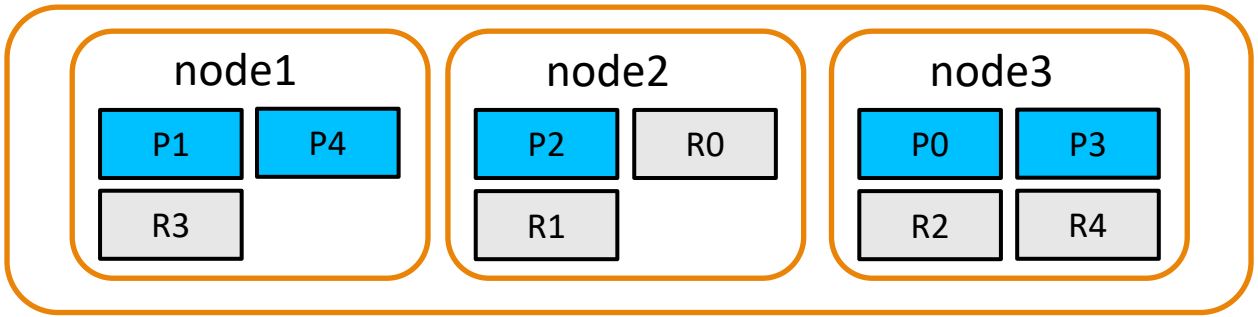


Anatomy of a Write Operation

How Data Gets In

- Let's take a look at the details of how a document is indexed into a cluster
  - Suppose we index the following document into our blogs index, which currently has 5 primary shards with 1 replica

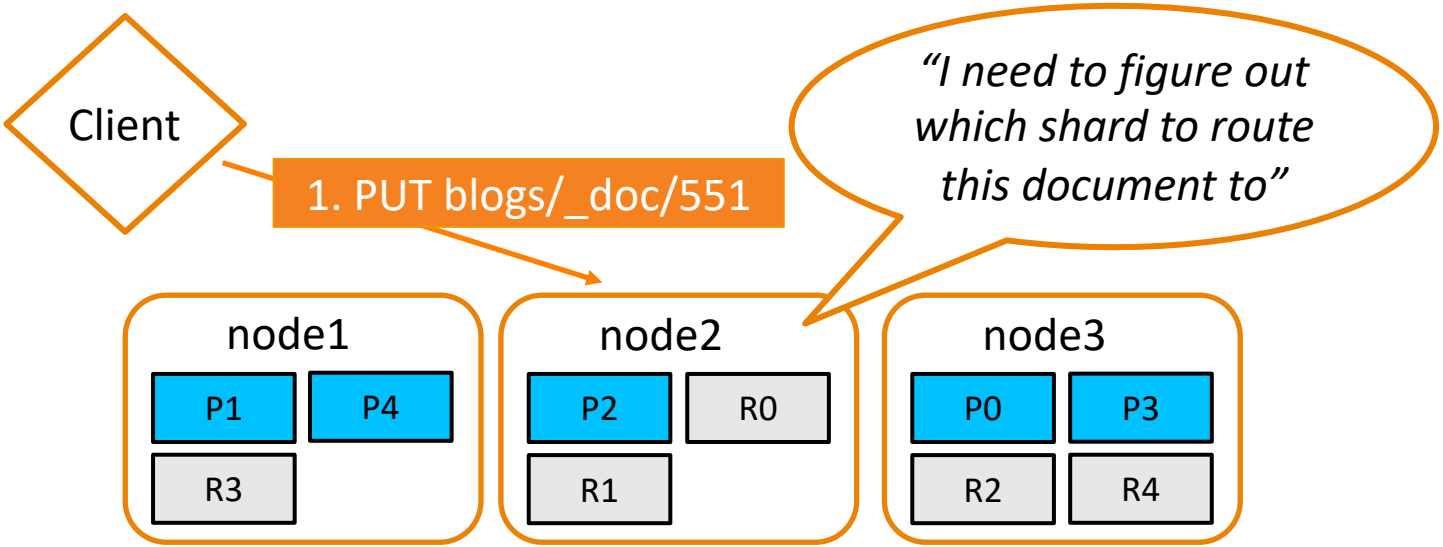
```
PUT blogs/_doc/551
{
  "title": "A History of Logstash Output Workers",
  "category": "Engineering",
  ...
}
```



Document Routing

- When a document is indexed, it needs to be determined which shard to index the document to
  - The shard is chosen based on a simple formula

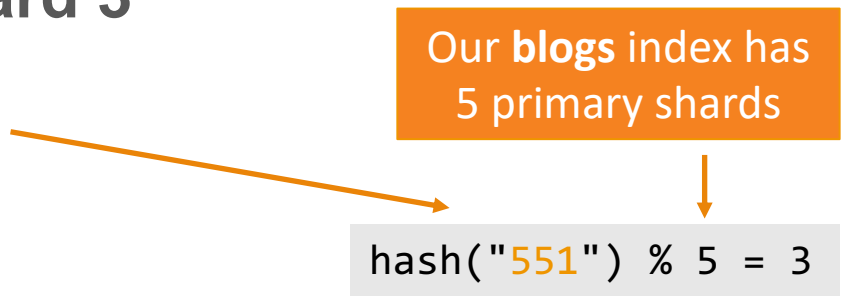
```
shard = hash(_routing) % number_of_primary_shards
```



Document Routing

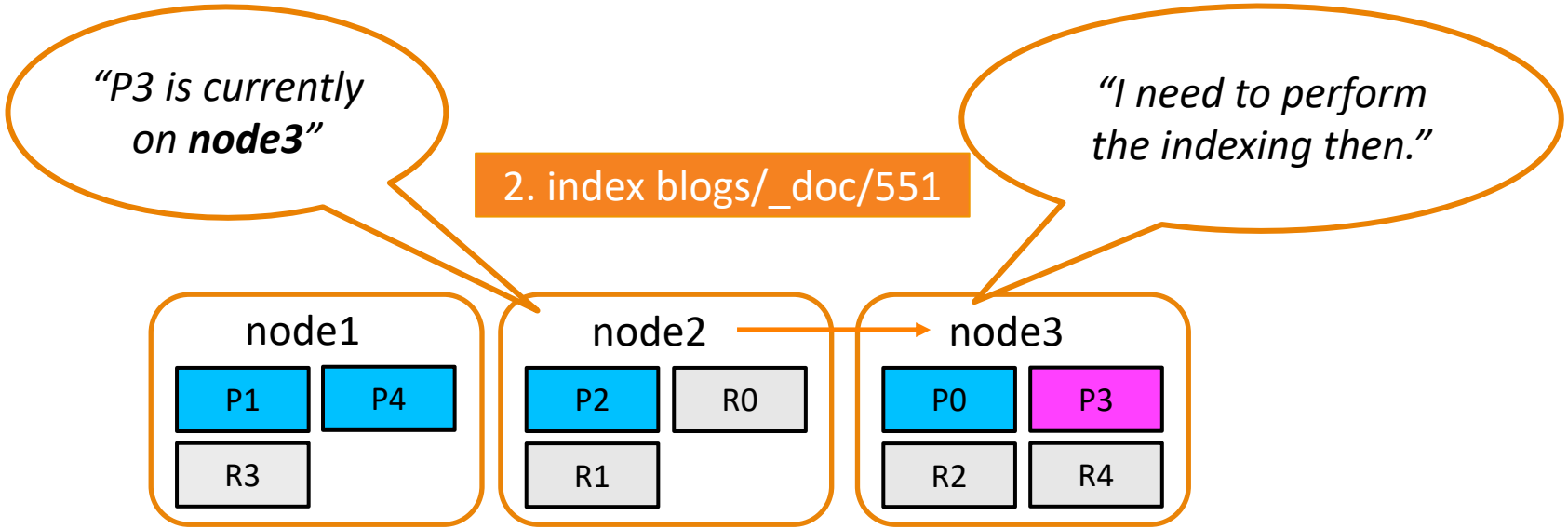
- **Why does it matter which shard a document is routed to?**
  - Because Elasticsearch needs to be able to retrieve the document later!
- **What is the value of _routing?**
  - The **_id** is used by default, but you can specify a different value ifdesired
- In our **blogs** index, the document with **_id** equal to "**551**" gets routed to **shard 3**

```
PUT blogs/_docs/551
{
   ...
}
```



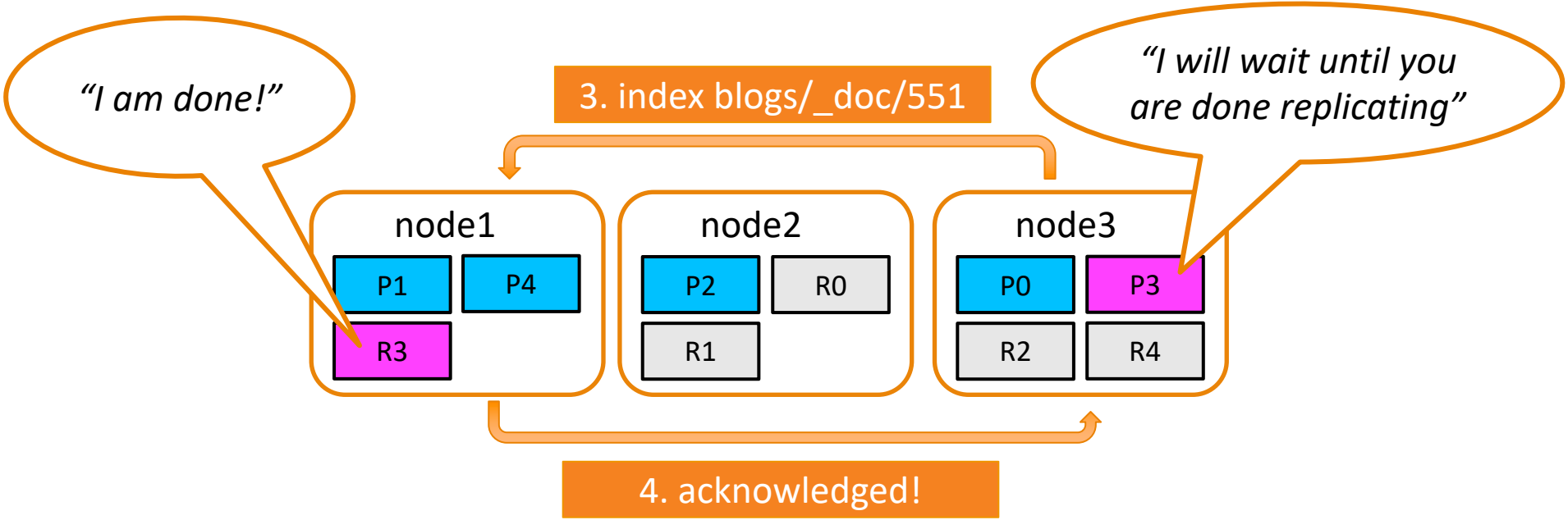
Write Operations on the Primary Shard

- When you index, delete, or update a document, the **primary shard** has to perform the operation first
  - so **node2** is going to forward the indexing request to **node3**



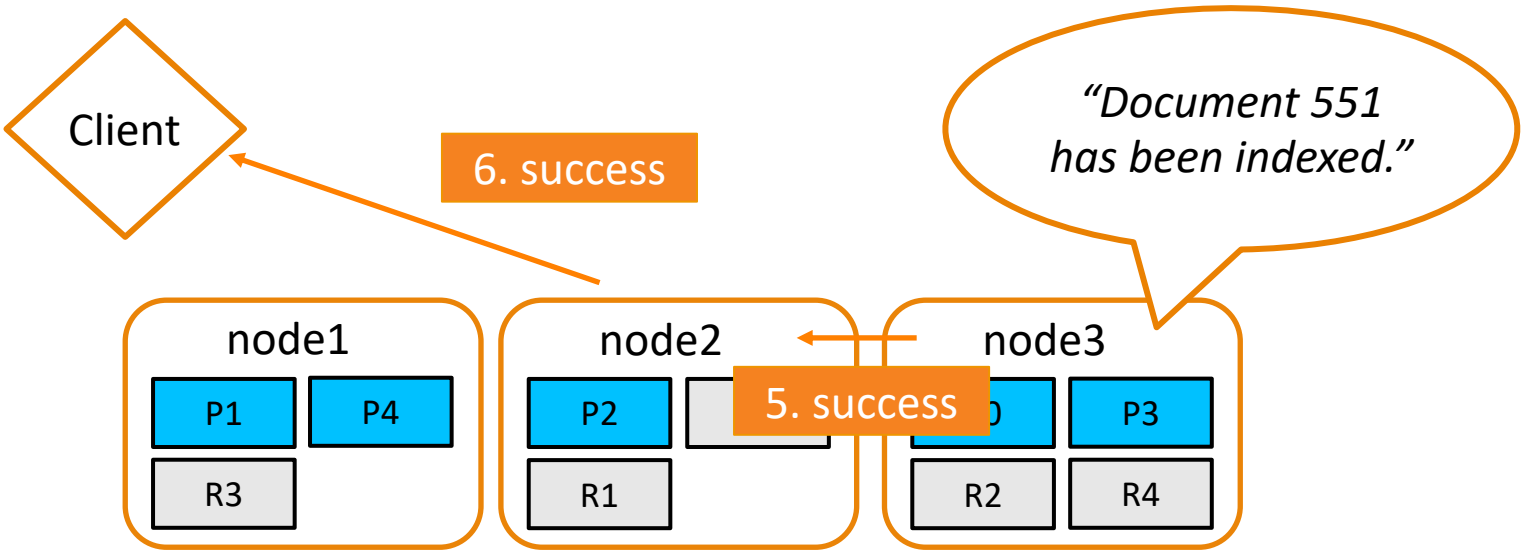
Replicas are Synced

- **node3** indexes the new document, then forwards the request (in parallel) to all replica shards
  - In our example, **P3** has one replica (**R3**) that is currently on **node1**



Client Response

- **node3** lets the coordinating node (**node2** in our example) know that the write operation is successful on every shard
  - and **node2** sends the details back to the client application



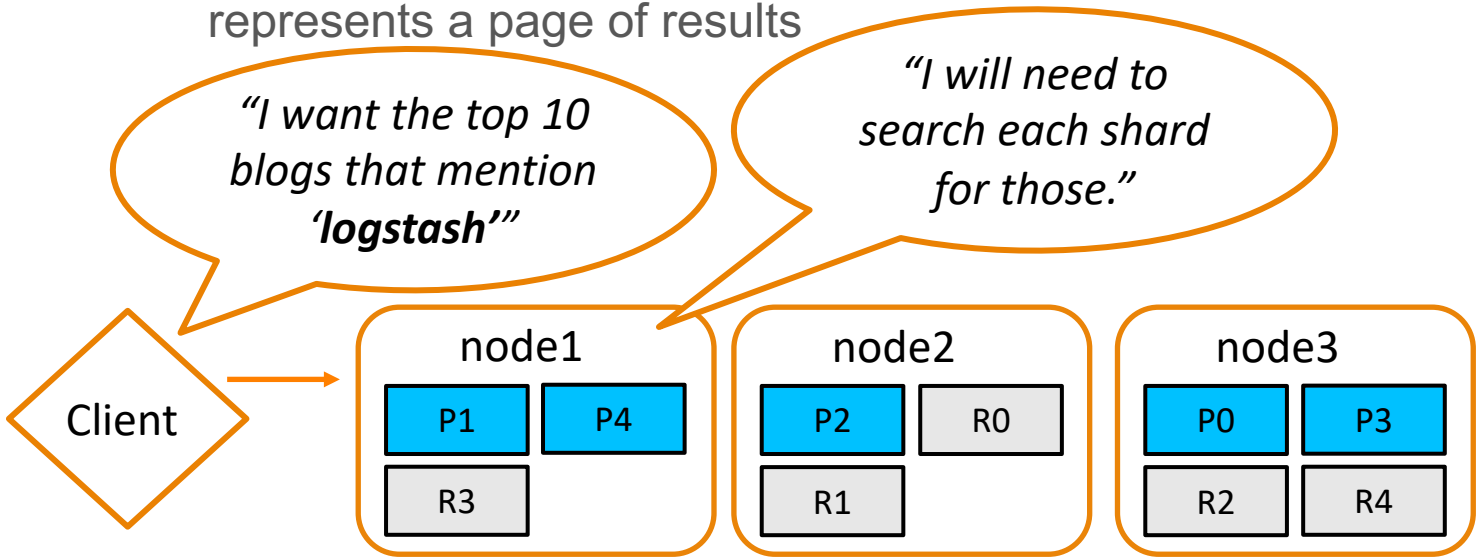
Updates and Deletes

- The process of an **_update** or **DELETE** is similar to the indexing of a document
- An **_update** to a document is actually three steps:
  - the source of the current document is retrieved
  - the current version of the document is deleted
  - then a merged new version of the entire document is indexed

Anatomy of a Search

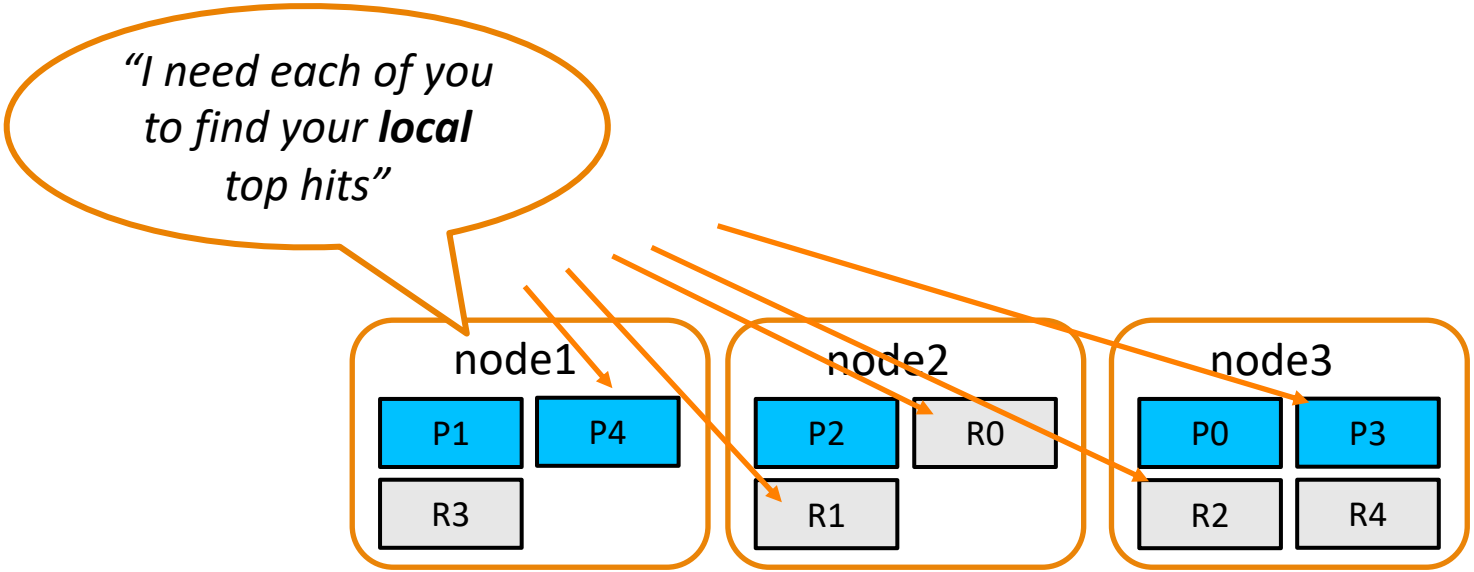
Anatomy of a Search

- Distributed search is a challenging task
  - We have to search for hits in a **copy of every shard** in the index
- And finding the documents is only half the story!
  - The hits must be combined into a single, sorted list of documents that represents a page of results



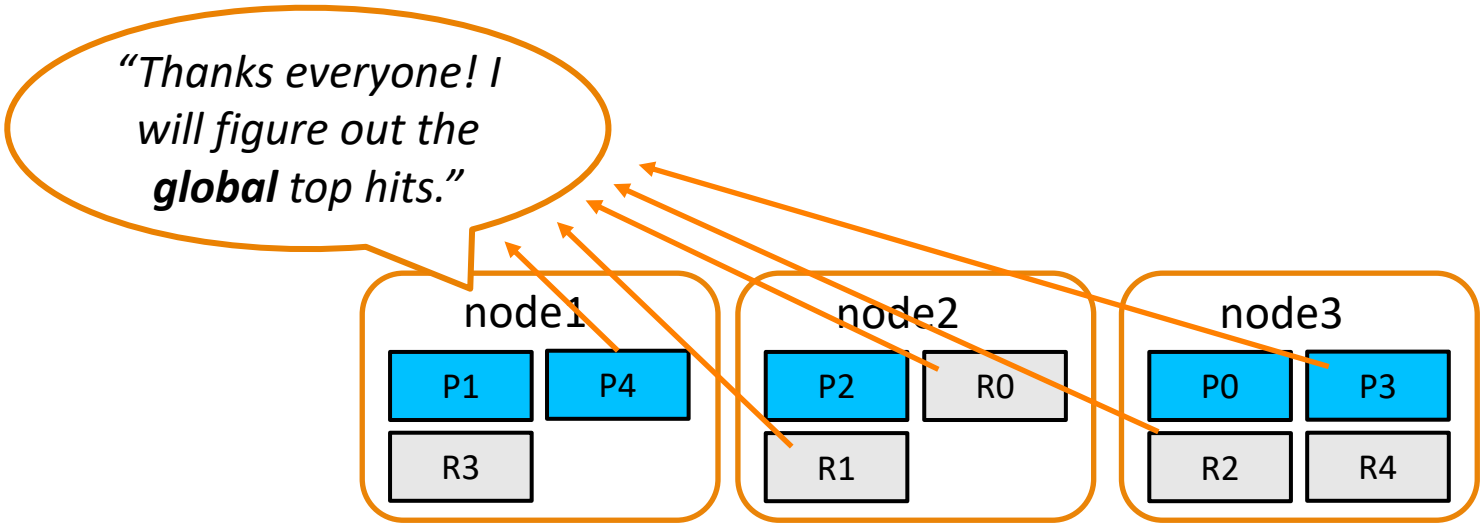
The Query Phase

- The initial part of a search is referred to as the **query phase**
  - The query is broadcast to a ***shard copy*** of every shard in the index
  - and each shard executes the query ***locally***



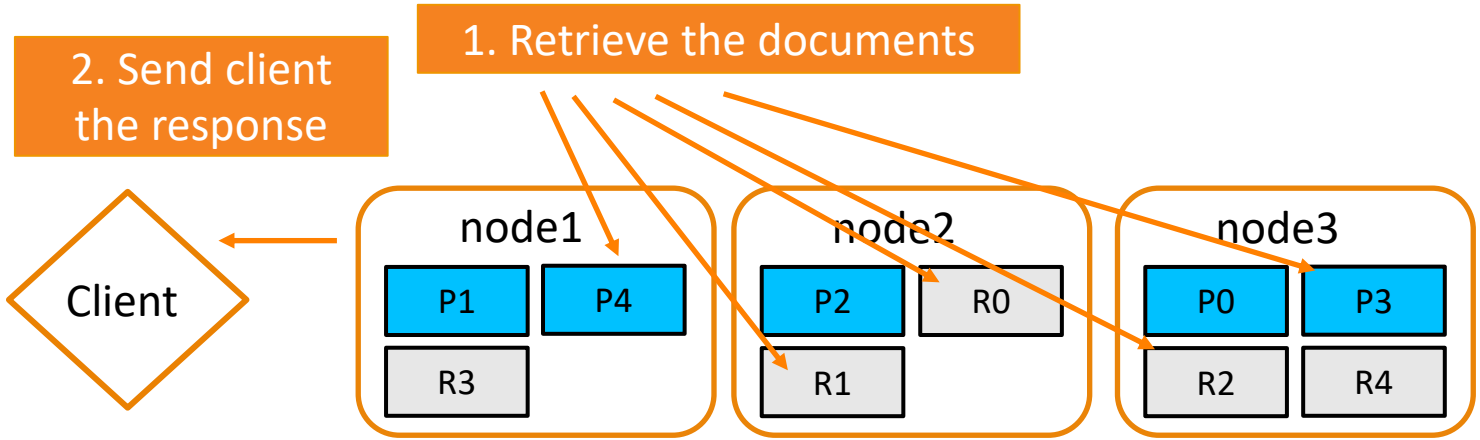
The Query Phase

- Each shard ***returns the doc IDs and sort values*** of its top hits to the coordinating node
- The coordinating node ***merges these values*** to create a ***globally*** sorted list of results



The Fetch Phase

- Now that the coordinating node knows the doc ID's of the top 10 hits, it can now ***fetch the documents***
  - and then returns the top documents to the client



Chapter Review

Summary

- Elasticsearch subdivides the data of your index into multiple pieces called ***shards***
- Each shard has one (and only one) ***primary*** and zero or more ***replicas***
- A search consists of a ***query phase*** and a ***fetch phase***

Quiz

1. If **number_of_shards** for an index is 4, and **number_of_replicas** is 2, how many total shards will exist for this index?
2. **True or False**: The nodes where shards are allocated are decided by the coordinating node.
3. **True or False**: By default, the size of a document is considered when determining which shard of the index to route the document to.
4. **True or False**: An index operation has to be executed on the primary shard first before being synced to replicas.

6. Mapping and Grok

Topics

- What is a Mapping?
- Grok

What is a Mapping?

What is a Mapping?

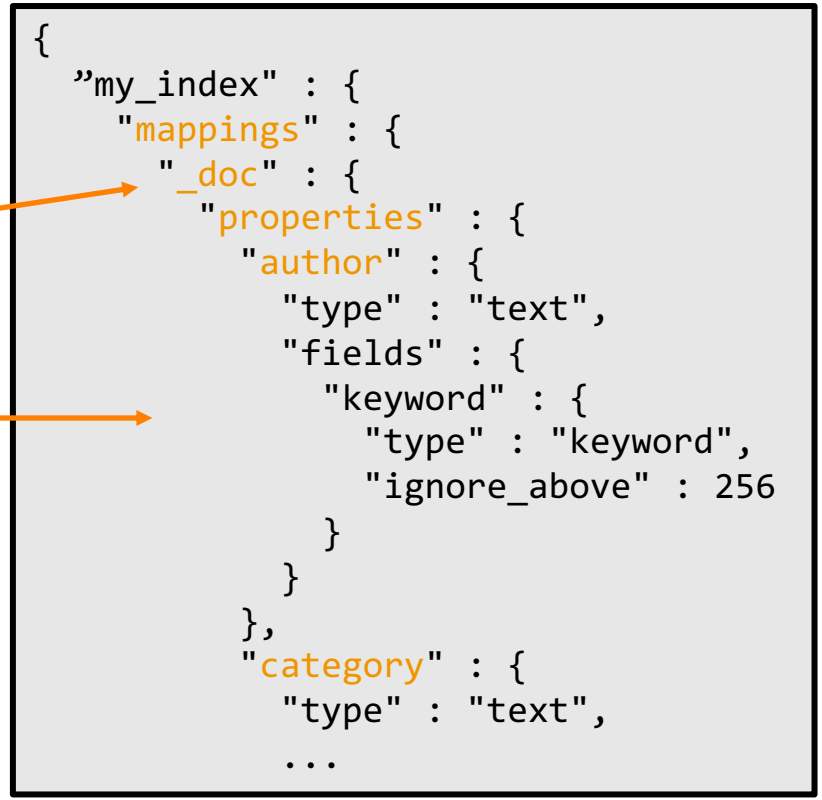
- Elasticsearch will happily index any document without knowing its details (number of fields, their data types, etc.)
  - However, behind-the-scenes Elasticsearch assigns data types to your fields in a ***mapping***
- A mapping is a schema definition that contains:
  - names of fields
  - data types of fields
  - how the field should be indexed and stored by Lucene
- Mappings map your complex JSON documents into the simple flat documents that Lucene expects

Let's view a mapping:

```
GET my_index/_mapping
```

This particular mapping has a ***type*** named **"_doc"**

The **"properties"** section contains the fields and data types in your documents



What is a Mapping?

- Each document has a ***type***
  - **Prior to 6.0**: an index could have multiple types
  - **Since 6.0**: an index can only have a single type
- Types were a design mistake that have caused issues with some users
  - so Elasticsearch is moving away from types
  - moving forward, you will eventually need to redesign your old indices to only have a single document type
- https://www.elastic.co/guide/en/elasticsearch/reference/current/removal-of-types.html

Elasticsearch Data Types for Fields

- **Simple Types, including**:
  - **text**: for full text (analyzed) strings
  - **keyword**: for exact value strings
  - **date**: string formatted as dates, or numeric dates
  - integer types: like **byte**, **short**, **integer**, **long**
  - floating-point numbers: **float**, **double**, **half_float**, **scaled_float**
  - **boolean**
  - **ip**: for IPv4 or IPv6 addresses
- **Hierarchical Types**: like **object** and **nested**
- **Specialized Types**: **geo_point** and **geo_shape**
- Plus range types and more

Define a Mapping

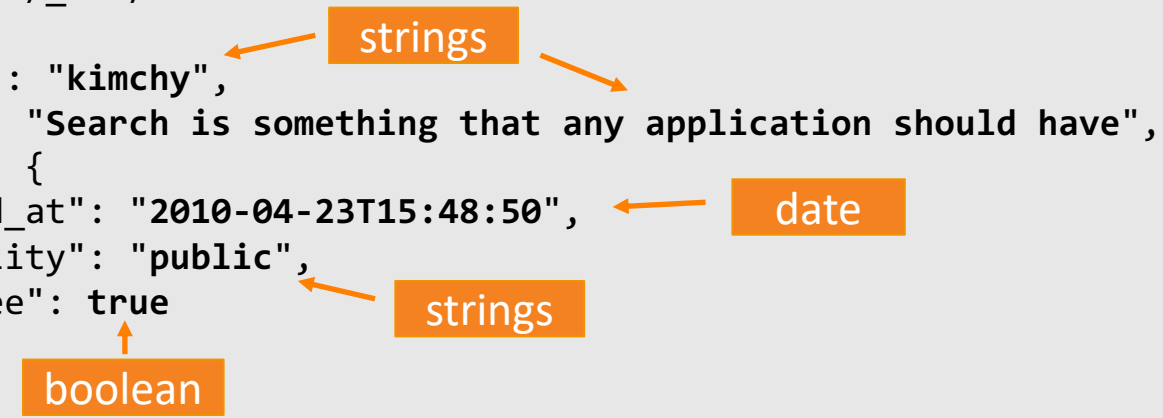
- In most use cases, you will need to define your own mappings
- Mappings are defined in the “**mappings**” section of an index. You can:
  - ***define*** mappings at index creation, or
  - ***add to*** a mapping of an existing index

```
PUT comments
{
  "mappings": {
    my mappings here
  }
}
```

Why have we not defined any mappings yet?

- You are not required to manually define mappings
  - When you index a document, Elasticsearch ***dynamically creates or updates the mapping***
- By default:
  - a new mapping is defined if one does not exist
  - fields not already defined in a mapping are added

```
PUT /my_index/_doc/1
{
  "username": "kimchy",
  "comment": "Search is something that any application should have",
  "details": {
    "created_at": "2010-04-23T15:48:50",
    "visibility": "public",
    "employee": true
  }
}
```



Elasticsearch “guesses” data types:

Sometimes it conveniently guesses what you want

```
PUT comments/_doc/2
{
  "num_of_views": 430
}
```

```
"num_of_views" : {
  "type" : "long"
}
```

Elasticsearch “guesses” data types:

Dates in the default format are also recognized

```
PUT comments/_doc/3
{
 "comment_time": "2016-11-06"

}
```

```
"comment_time" : {
  "type" : "date"
}
```

Elasticsearch “guesses” data types:

Sometimes it may choose a type you do **not** want...

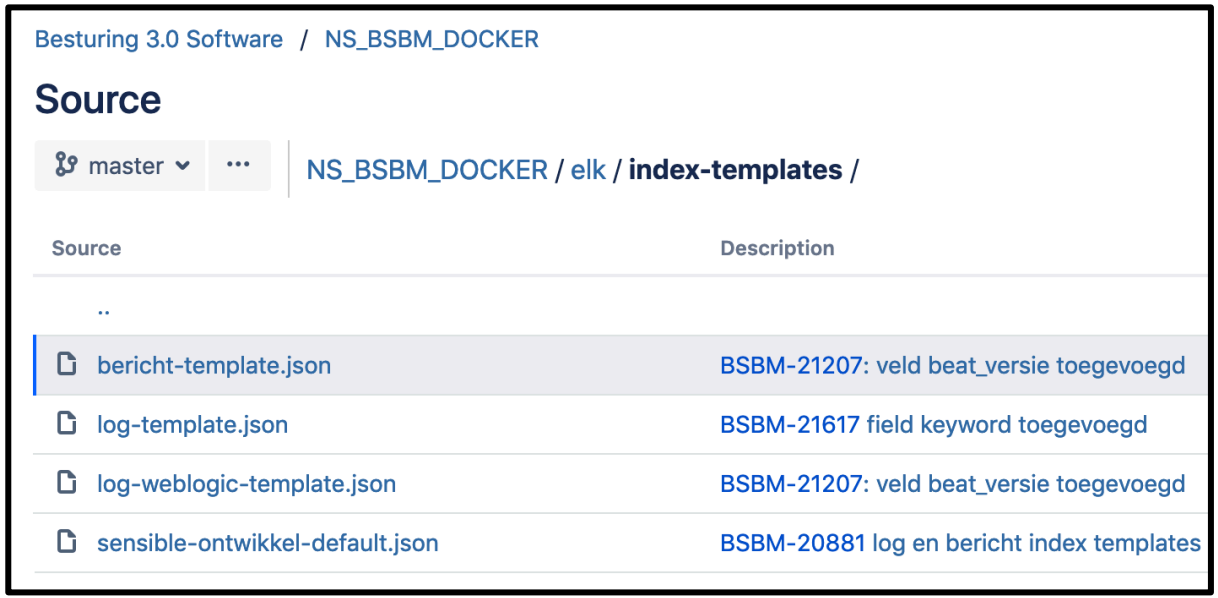
```
PUT comments/_doc/3
{
 "average_length": "24.8"

}
```

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

- Strings are mapped twice: once as type text and once as type keyword

So where are our mappings?

- We use index templates to define our mappings
- Template JSON is found in NS_BSBM_DOCKER repo



Grok

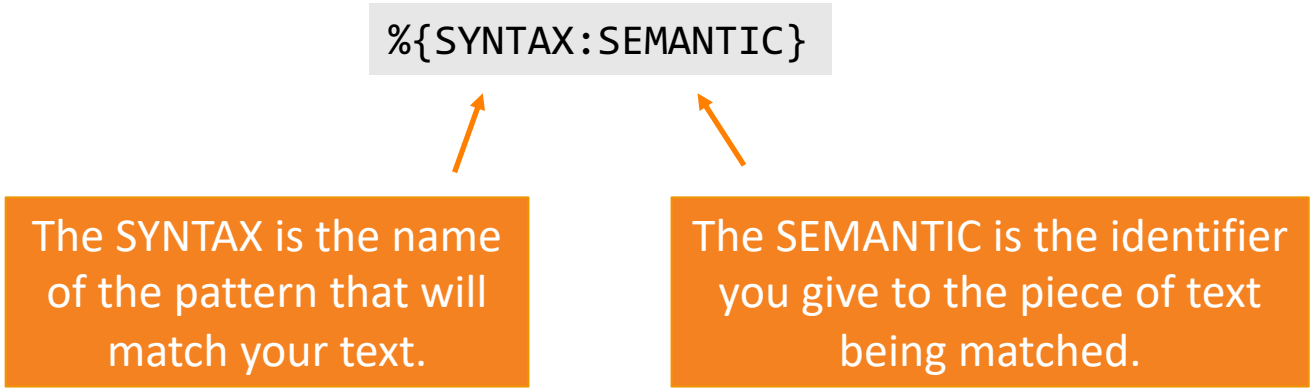
Grok

- Our logfiles are formatted in a specific way
- Elasticsearch has no idea what each field should have as a tag/name
- Enter Grok!

Grok

- Parse arbitrary text and structure it.
- Grok is a great way to parse unstructured log data into something structured and queryable.
- https://www.elastic.co/guide/en/logstash/current/plugins-filters-grok.html

Syntax for a grok pattern



Built-in patterns

https://github.com/elastic/logstash/blob/v1.4.2/patterns/grok-patterns

Executable File | 95 lines (84 sloc) | 5.18 KB — Raw | Blame | History

```
1  USERNAME [a-zA-Z0-9._-]+
2  USER %{USERNAME}
3  INT (?:[+-]?(?:[0-9]+))
4  BASE10NUM (?<![0-9.+-])(?>[+-]?(?:(?:[0-9]+(?:\.[0-9]+)?)|(?:\.[0-9]+)))
5  NUMBER (?:%{BASE10NUM})
6  BASE16NUM (?<![0-9A-Fa-f])(?:[+-]?(?:0x)?(?:[0-9A-Fa-f]+))
7  BASE16FLOAT \b(?<![0-9A-Fa-f.])(?:[+-]?(?:0x)?(?:(?:[0-9A-Fa-f]+(?:\.[0-9A-Fa-f]*)?)|(?:\.[0-9A-Fa-f]+)))\b
8
9  POSINT \b(?:[1-9][0-9]*)\b
10 NONNEGINT \b(?:[0-9]+)\b
11 WORD \b\w+\b
12 NOTSPACE \S+
13 SPACE \s*
14 DATA .*?
15 GREEDYDATA .*
16 QUOTEDSTRING (?>(?<!\\)(?>"(?>\\.|[^\\"]+)+"|""|(?>'(?>\\.|[^\\']+)+')|''|(?>`(?>\\.|[^\\`]+)+`)|``))
17 UUID [A-Fa-f0-9]{8}-(?:[A-Fa-f0-9]{4}-){3}[A-Fa-f0-9]{12}
18
```

A simple example

```
Volkswagen -- Germany
```

```
%{DATA:brand} -- %{GREEDYDATA:brand_country}
```

DATA syntax is .*? regex

GREEDYDATA syntax is .* regex

```
{
  "brand_country": "Germany",
  "brand": "Volkswagen"
}
```

Custom Patterns

- Grok supports custom patterns
  - Nothing more than named regex patterns
- Defined in a file
- Pointed to in Logstash config

Custom Patterns

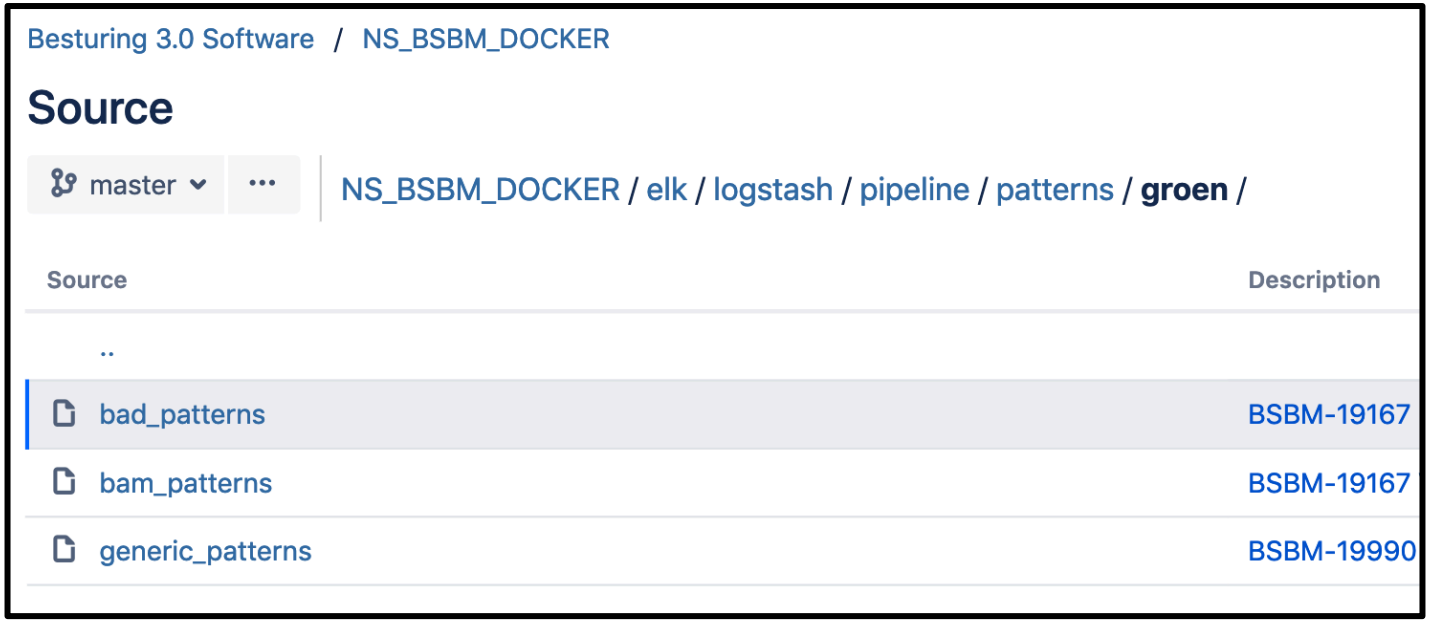
```
%{BAM_LOGREGEL}
```

```
BAM_LOGREGEL %{DATA:log_timestamp}%{DELIM}%{DATA:loglevel}...
```

```
DELIM \s-\s
```

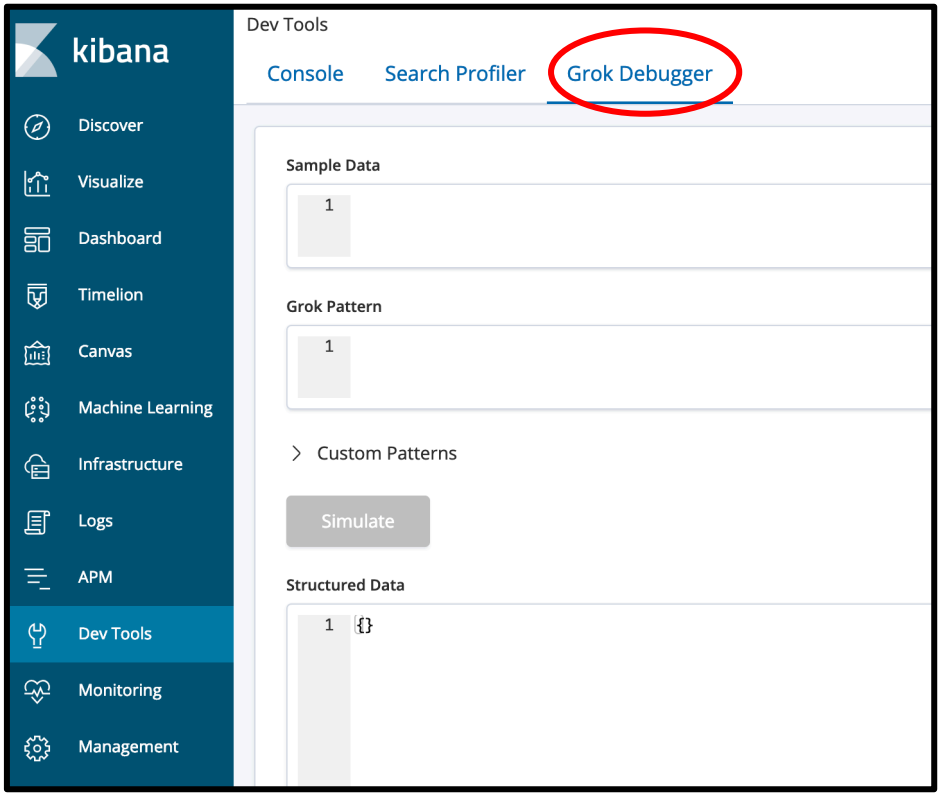
Grok patterns used by us

- Grok patterns are found in NS_BSBM_DOCKER repo



Debugging Grok

- Dev Tools in Kibana has a Grok Debugger



Chapter Review

Summary

- Every type has a ***mapping*** that defines a document's fields and their data types
- Mappings are created ***dynamically***, or you can define your own
- Grok is a great way to parse ***unstructured*** log data into something ***structured*** and ***queryable***

Quiz

1. What data type would **“value” : 300** get mapped to dynamically?
2. What data type would **“value”: “January”** get mapped to dynamically?
3. **True or False**: You can change a field’s data type from “**integer**” to “**long**” because those two types are compatible.

Lab 6