



elastic

# Core Elasticsearch for Developers

An Elastic Training Course

Yasaswy Ravalay  
24-Apr-2017 Capital One

[training.elastic.co](http://training.elastic.co)

Course: Core Elasticsearch for Developers

Version 5.3.1

© 2015-2017 Elasticsearch BV. All rights reserved. Decompiling, copying, publishing and/or distribution without written consent of Elasticsearch BV is strictly prohibited.

Yasaswy Raval - 24-Apr-2017 - Capital One

# Lab Prep Checklist

- Download the PDF and zip file from links provided in email
- Minimum requirements:
  - 20% or more free disk space
  - Install Java
    - version 1.8.0\_20 minimum/1.8.0\_73 or newer recommended
    - 64-bit JDK preferred
    - set **JAVA\_HOME** environment variable
- Unzip the downloaded lab zip file on your hard drive
  - this creates a directory called **CoreDeveloper**
  - do **not** unzip the lab file into a folder with spaces in the name

# Agenda and Introductions

Yasaswy Raval - 24-Apr-2017 - Capital One

# Course Agenda

1 Introduction to Elasticsearch

2 The Search API

3 Text Analysis

4 Mappings

5 More Search Features

6 The Distributed Model

7 Working with Search Results

8 Aggregations

9 More Aggregations

10 Handling Relationships

# Introductions

- Name
- Company
- What do you do?
- What are you using Elasticsearch for?
- What do you hope to get out of this training?

Yasaswy Raval - 24-Apr-2017 - Capital One

# Logistics

- Facilities
- Emergency Exits
- Restrooms
- Breaks/Lunch

Yasaswy Raval - 24-Apr-2017 - Capital One

# Chapter 1

# Introduction to Elasticsearch

Yasaswy Raval - 24-Apr-2017 - Capital One

- 1 Introduction to Elasticsearch
- 2 The Search API
- 3 Text Analysis
- 4 Mappings
- 5 More Search Features
- 6 The Distributed Model
- 7 Working with Search Results
- 8 Aggregations
- 9 More Aggregations
- 10 Handling Relationships

# Topics covered:

- The Story of Elasticsearch
- The Components of Elasticsearch
- Documents
- Indexes
- Indexing Data
- Searching Data
- The Bulk API

Yasaswy Raval - 24-Apr-2017 - Capital One

# The Story of Elasticsearch

Yasaswy Raval - 24-Apr-2017, Capital One



# Once upon a time...

- As any good story begins, “Once upon a time...”
  - More precisely: 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

“

**Search is something  
that any application  
should have”**

**Shay Banon**  
*Creator of Elasticsearch*

Yasaswini Pavalal - 24-Apr-2017 - Capital One

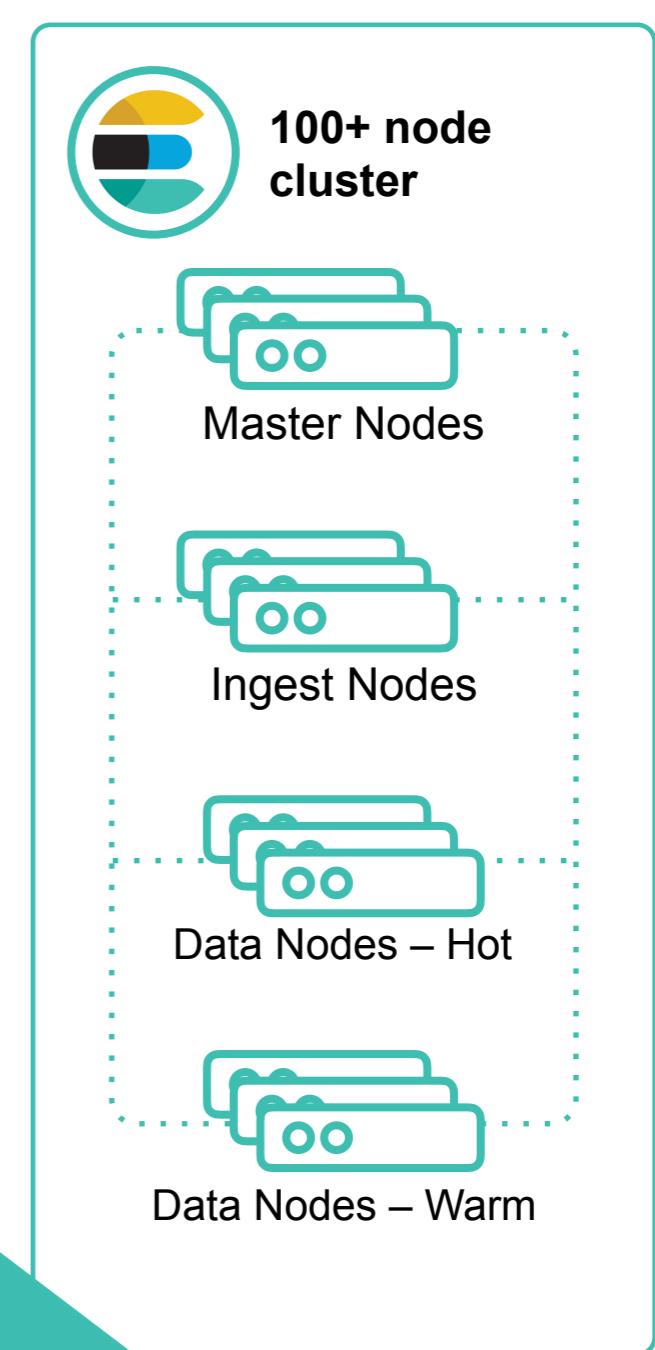
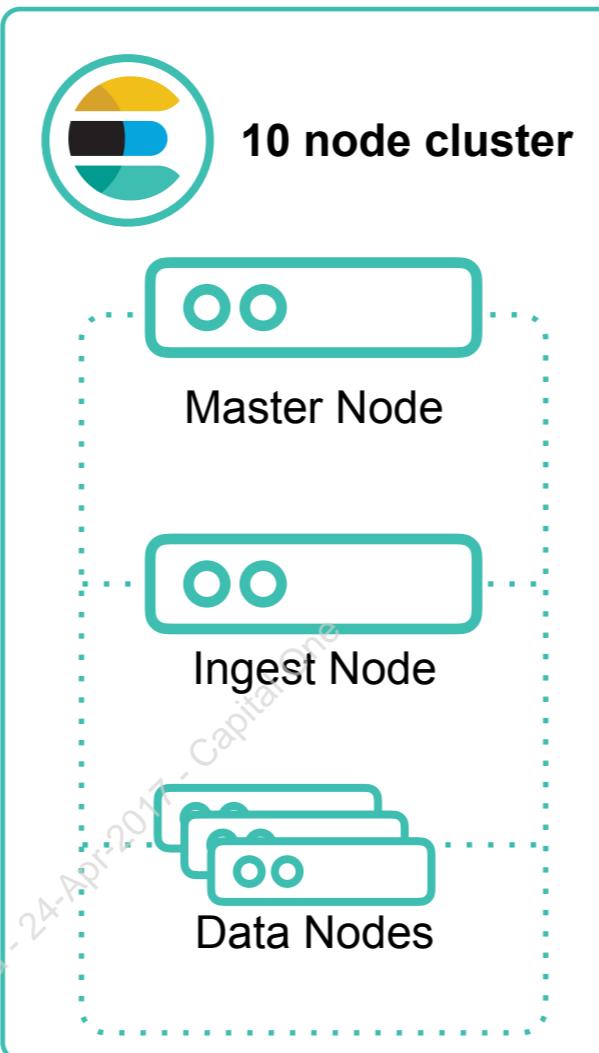
# The Birth of Elasticsearch

- 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
- In 2010, Shay 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:



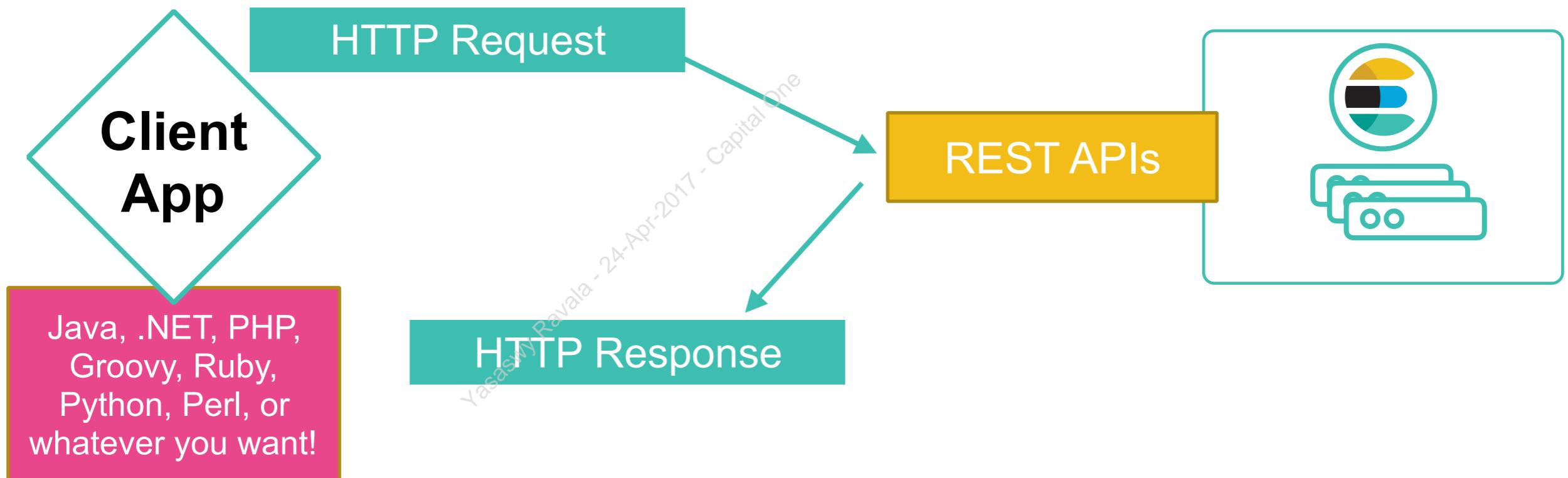
A **node** is an instance of Elasticsearch

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 communicating with a cluster over HTTP
  - Allows client applications to be written in any language



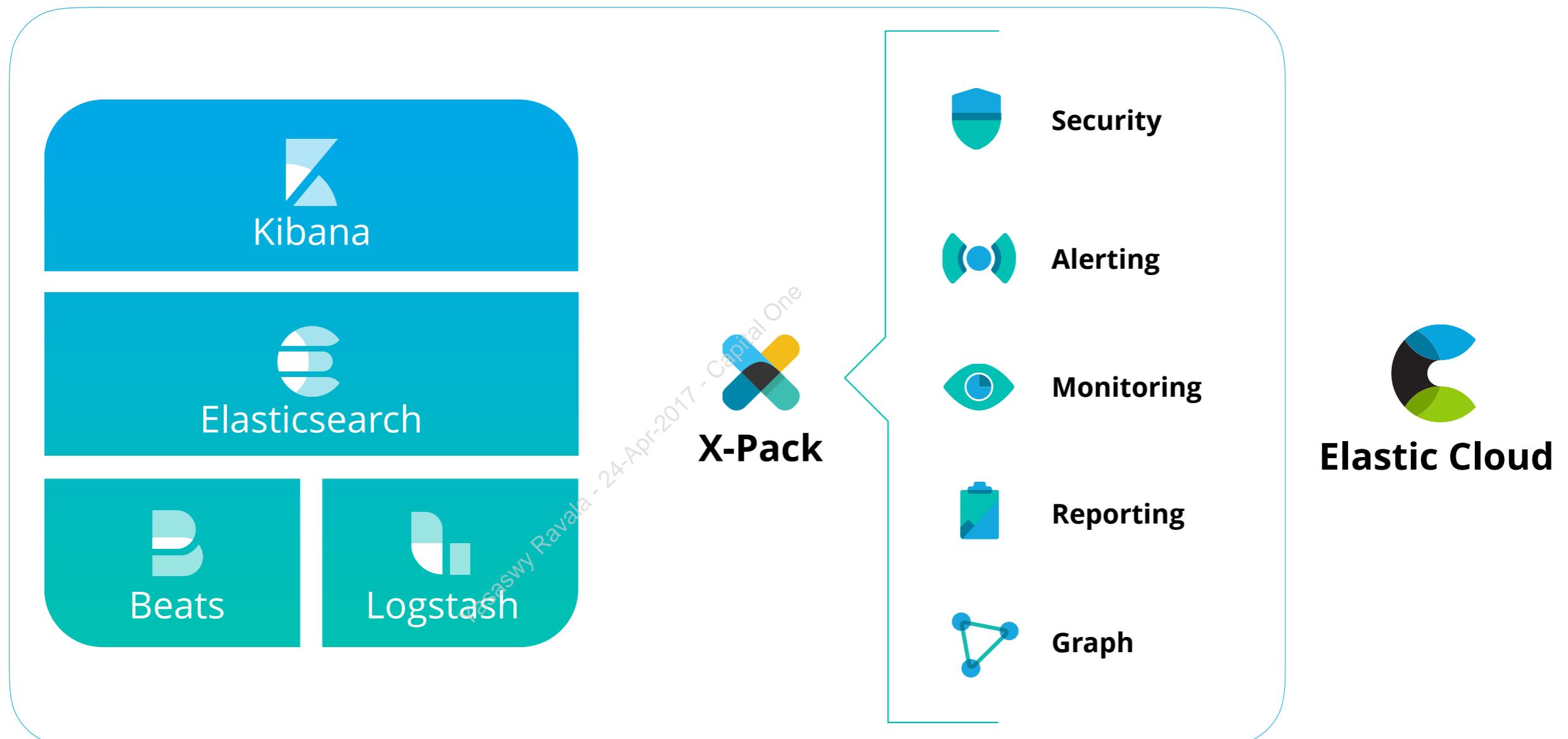
# Elasticsearch History

- **0.4:** first version was released in February, 2010
- **1.0:** released in January, 2014
- **2.0:** released in October, 2015
- **5.0:** released in October, 2016
- Why the big jump in version number?
  - The Elastic Stack...

Yasaswy Raval - 24-Apr-2017 - Capital One

# The Elastic Stack

- Elasticsearch is a component of the *Elastic Stack*



# The Components of Elasticsearch

Yasaswy Raval - 24-Apr-2017 Capital One

# The Components of Elasticsearch

- The main components of Elasticsearch are:
  - ***node***: an instance of Elasticsearch
  - ***cluster***: one or more nodes that work together to serve the same data and API requests
  - ***document***: a piece of data that you want to search
  - ***index***: a collection of documents that have somewhat similar characteristics
- Let's take a look at documents and indexes (indices)...



# Documents

Yasaswy Raval - 24-Apr-2017 - Capital One

# Documents

- A **document** can be any text or numeric data you want to search and/or analyze
- For example:
  - an entry in a log file,
  - a comment made by a customer on your website,
  - the weather details from a weather station at a specific moment
- Specifically, a **document** is a top-level object that is serialized into JSON and stored in Elasticsearch
- Every document has a **unique ID**
  - which either you provide
  - or Elasticsearch generates for you



# Documents must be JSON objects

- Suppose you have some stock prices in a CSV format
  - The data needs to be converted to JSON
  - Each row of data represents a single document



stocks.csv

each row has stock price data



20100526,PPG,62.83,62.83,61.46,61.86,32203

convert the CSV to JSON

```
{  
    "volume" : 32203,  
    "high" : 62.83,  
    "stock_symbol" : "PPG",  
    "low" : 61.46,  
    "close" : 61.86,  
    "trade_date" : "2010-05-26T06:00:00.000Z",  
    "open" : 62.83  
}
```

JSON consists of **fields** and **values**



# Indexes

Yasaswy Raval - 24-Apr-2017 - Capital One

# Definition of an Index

- An **index** in Elasticsearch is a ***logical*** way of grouping data:
  - An index contains ***mappings*** that define the documents' field names and data types of the index
  - an index is a ***logical namespace*** that maps to where its contents are stored in the cluster
- There are two different concepts in this definition:
  - an index has some type of data schema mechanism
  - an index has some type of mechanism to distribute data across a cluster



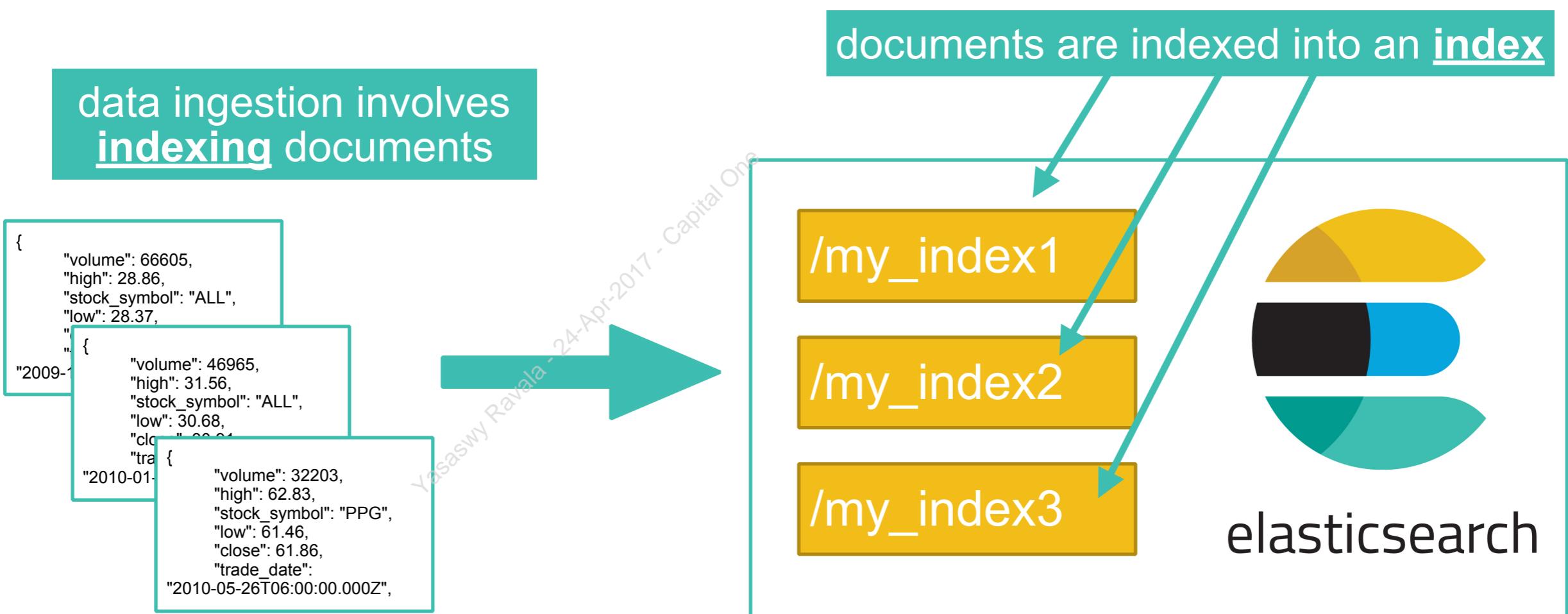
# You can have lots of indexes:

- Indexes are fairly lightweight in Elasticsearch, so it is not unusual to create lots of them
- For example, you might define a new index everyday for searching logs
  - Using multiple indices allows you to organize your data *logically*:
  - **logs-2017-01-01**
  - **logs-2017-01-02**
  - **logs-2017-01-03**
  - and so on



# When we say “index”, we mean...

- We use the word “index” in multiple ways:
  - **Noun:** a document is put into an *index* in Elasticsearch
  - **Verb:** to *index* a document is to put the document into an index in Elasticsearch



# Indexing Data

Yasaswy Raval - 24-Apr-2017 - Capital One

# Let's index some simple JSON documents:

“username” and  
“comment” are fields

```
{  
  "username" : "harrison",  
  "comment" : "My favorite movie is Star Wars!"  
}
```

These are the values

```
{  
  "username" : "maria",  
  "comment" : "The North Star is right above my house."  
}
```

```
{  
  "username" : "eniko",  
  "comment" : "My favorite movie star in Hollywood is Harrison Ford."  
}
```



# Define an Index

- Clients communicate with a cluster using Elasticsearch's REST APIs
  - There is a collection of ***Document APIs*** for working with indexes and documents
- An index is defined using the **Create Index API**, which can be accomplished with a simple **PUT** command:
  - “200 OK” response if successful

```
curl -XPUT 'http://localhost:9200/my_index' -H "Content-Type: application/json"
```

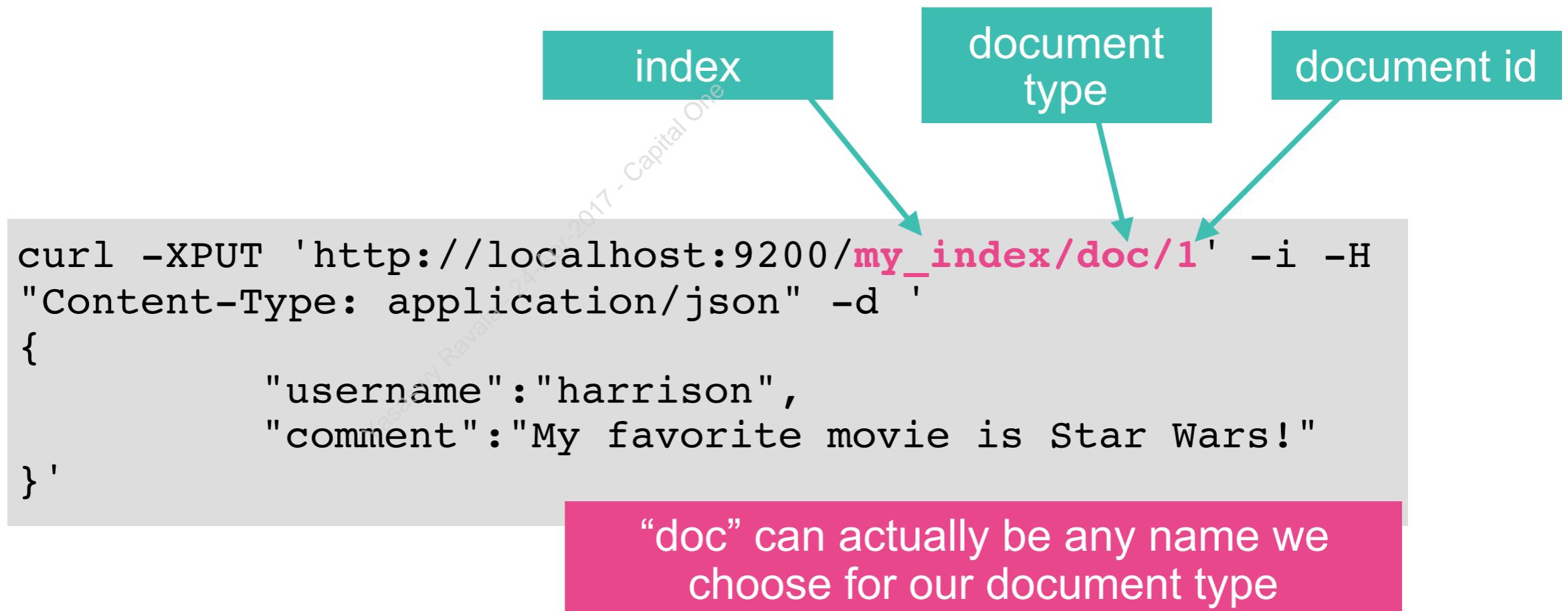
PUT command

The name of  
the new index



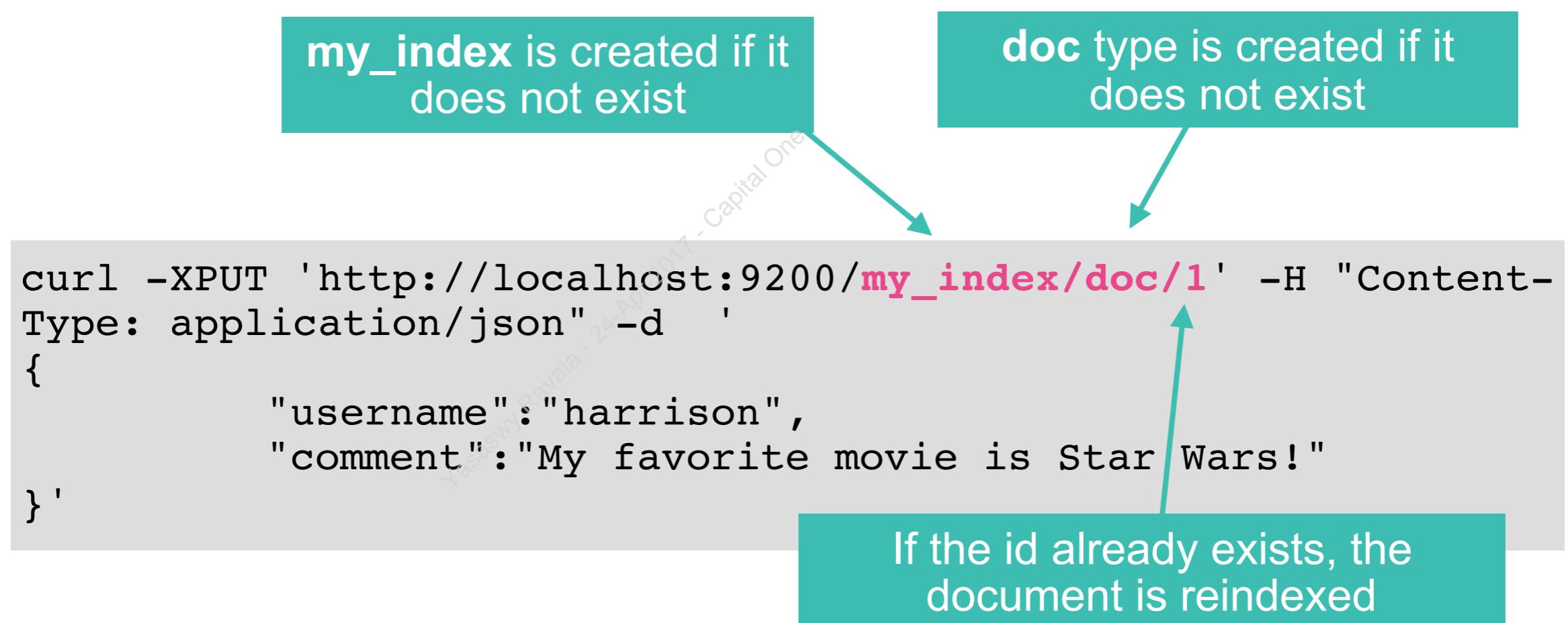
# Index a Document

- The **Index API** is used to index a document
- Use a **PUT** again, but this time send the document in the body of the PUT request:
  - notice you specify a document **type** and a unique **ID**
  - “201 Created” response if successful



# Dynamic Index Creation

- Elasticsearch will create the index for you during the indexing of a document, if the index is not already created:
  - we could have simply executed the command on the previous slide (and skipped that first “PUT `my_index`” command)



# Index Without Specifying an ID

- You can leave off the id and let Elasticsearch generate one for you:
  - But notice that only works with **POST**, not **PUT**

POST instead of PUT

```
curl -XPOST 'http://localhost:9200/my_index/doc/' -i -H  
"Content-Type: application/json" -d '  
{  
  "username" : "maria",  
  "comment" : "The North Star is right above my house."  
}'
```

No id specified

# Index Without Specifying an ID

- If the **POST** is successful, the auto-generated ID is returned in the response:

```
HTTP/1.1 201 Created
Location: /my_index/doc/
AVnWIzYrUxRN673zt1An
Content-Type: application/json;
charset=UTF-8
Content-Length: 220

{
  "_index" : "my_index",
  "_type" : "doc",
  "_id" : "AVnWIzYrUxRN673zt1An",
  "_version" : 1,
  "result" : "created",
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 0
  },
  "created" : true
}
```

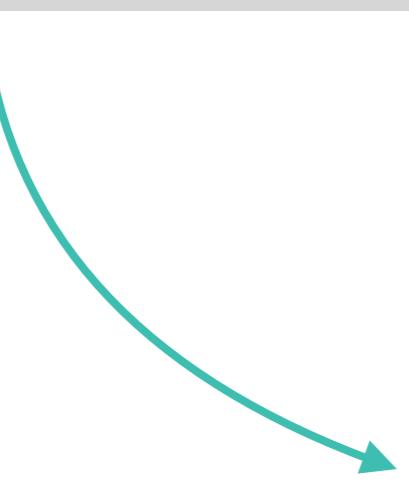
201 response if successful



# Retrieving a Document

- Use **GET** to retrieve an indexed document
  - Returns 200 if the document is found
  - Returns a 404 error if the document is not found

```
curl -XGET 'http://localhost:9200/my_index/doc/1' -H  
"Content-Type: application/json"
```



```
{  
  "_index": "my_index",  
  "_type": "doc",  
  "_id": "1",  
  "_version": 1,  
  "found": true,  
  "_source": {  
    "username": "harrison",  
    "comment": "My favorite movie is Star Wars!"  
  }  
}
```

# The `_source` Endpoint

- You can get just the source of a document using the `_source` endpoint

```
curl -XGET 'http://localhost:9200/my_index/doc/1/_source' -H  
"Content-Type: application/json"
```

```
{  
  "username": "harrison",  
  "comment": "My favorite movie is Star Wars!"  
}
```

# Searching Data

Yasaswy Raval - 24-Apr-2017 - Capital One

# “Hello, Search!”

- Let's run our first search
  - sort of a “Hello, World!” for Elasticsearch
- The simplest search is a **match\_all**, which simply matches all documents in the index (or indices) being searched:

Use **GET** or **POST**

```
curl -XGET 'http://localhost:9200/my_index/_search' -H  
"Content-Type: application/json" -d '  
{  
  "query": {  
    "match_all": {}  
  }  
}'
```

Invoke the **\_search** endpoint

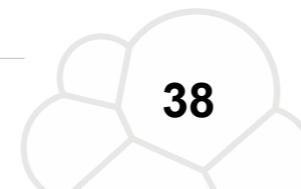
Add a “query” clause

“**match\_all**” matches all documents



# ...and the search results are also JSON

```
{  
  "took": 1, ← took = the number of milliseconds it  
  "timed_out": false,  
  "_shards": {  
    "total": 10,  
    "successful": 10,  
    "failed": 0  
  },  
  "hits": {  
    "total": 3, ← total = the number of documents found  
    "max_score": 1,  
    "hits": [  
      {  
        "_index": "my_index",  
        "_type": "doc",  
        "_id": "AV1BSnbcM0CN4-9FrtP", ← _id = the document's unique ID  
        "_score": 1,  
        "_source": {  
          "username": "maria",  
          "comment": "The North Star is right above my house."  
        }  
      },  
      ...  
    ]  
  }  
}
```



# A Simpler “Hello, World”

- If you want a search that matches all documents, you can simply send a request to the `_search` endpoint
  - and omit the “`query`” body
  - same results as running a `match_all`
  - by default, Elasticsearch returns 10 results

```
curl -XGET 'http://localhost:9200/my_index/_search' -H  
"Content-Type: application/json"
```



# CRUD Operations

Yasaswy Raval - 24-Apr-2017 - Capital One

# CRUD Operations

- We will discuss these in more detail later in the course:

Create

```
PUT my_index/doc/4
{
  "username" : "kimchy",
  "comment" : "I like search!"
}
```

Read

```
GET my_index/doc/4
```

Update

```
POST my_index/doc/4/_update
{
  "doc" : {
    "comment" : "I love search!!"
  }
}
```

Delete

```
DELETE my_index/doc/4
```

# The Multi Get API

- The *Multi Get API* allows you to GET multiple documents in a single request
  - Avoid multiple round trips
- Use the `_mget` endpoint:

```
GET _mget
{
  "docs" : [
    {
      "_index" : "INDEX_NAME",
      "_type" : "TYPE",
      "_id" : "ID"
    },
    {
      "_index" : "INDEX_NAME",
      "_type" : "TYPE",
      "_id" : "ID"
    }
  ]
}
```

“`docs`” is an array of documents to GET



# Example of \_mget

```
GET _mget
{
  "docs" : [
    {
      "_index" : "products",
      "_type" : "product",
      "_id" : 101
    },
    {
      "_index" : "my_index",
      "_type" : "doc",
      "_id" : 4
    }
  ]
}
```

```
{
  "docs" : [
    {
      "_index" : "products",
      "_type" : "product",
      "_id" : "101",
      "_version": 1,
      "found": true,
      "_source": {
        "brandName": "Kraft",
        "customerRating": 5,
        "price": 4.29,
        "grp_id": "101",
        "quantitySold": 844255,
        "upc12": "021000023233",
        "productName": "Kraft Velveeta
Shells & Cheese 2% Milk"
      }
    },
    {
      "_index": "my_index",
      "_type": "doc",
      "_id": "4",
      "_version": 1,
      "found": true,
      "_source": {
        "username": "kimchy",
        "comment": "I like search!"
      }
    }
  ]
}
```

# The Bulk API

Yasaswy Raval - 24-Apr-2017 - Capital One

# 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 of hundreds or thousands
- Four actions: **create, index, update and delete**
- The response is a large JSON structure with the individual results of each action that was performed
  - The failure of a single action does not affect the remaining actions



# Syntax of \_bulk

- Has a unique syntax based on lines of commands:
  - Each command appears on a single line

```
POST _bulk
{"create" : {...}}
{DOCUMENT}
{"index" : {...}}
{DOCUMENT}
{"delete" : {...}}
```

The line following “**create**” or “**index**” is the document that gets indexed

“**delete**” does not have a following line

# Example of \_bulk

```
POST _bulk
{"create" : {"_index" : "customers", "_type": "customer_type", "_id":102}}
{"firstname" : "Maureen", "lastname" : "O'Hara", "address" : "12 Elm St", "city": "Brooklyn"}
{"index" : {"_index" : "customers", "_type": "customer_type", "_id":103}}
{"firstname" : "Lucy", "lastname" : "Liu", "address" : "39 Pine Tree Ln", "city": "Albany"}
{"delete" : {"_index" : "customers", "_type": "customer_type", "_id":104}}
```

The final \n is actually necessary

# Error Messages

- If things go wrong in your requests:

Issue	Reason	Action
<b>Unable to connect</b>	networking issue or cluster down	retry until connection is available (round robin across nodes)
<b>5xx error</b>	internal server error in Elasticsearch	retry until successful (using round robin)
<b>4xx error</b>	invalid JSON or incorrect request structure	fix bad request before retrying
<b>429 error</b>	Elasticsearch is too busy	retry (ideally with linear or exponential backoff)
<b>connection unexpectedly closed</b>	node died or network issue	retry (with risk of creating a duplicate document)

# Chapter Review

Yasaswy Raval - 24-Apr-2017 - Capital One

# Summary

- Elasticsearch is a search and analytics engine built on top of Apache Lucene
- Elasticsearch is distributed and scales horizontally
- Clients communicate with a cluster using Elasticsearch's REST APIs
- Starting with version 5.0, all of Elastic Stack products will be aligned, tested, and released together.
- A ***document*** can be any JSON-formatted data you want to search and/or analyze
- An ***index*** in Elasticsearch is a logical way of grouping data
- Adding documents to Elasticsearch is called ***indexing***



# Quiz

1. **True or False:** Elasticsearch uses Apache Lucene behind the scenes to index and search data.
2. What are two different uses of the term “index”?
3. **True or False:** Every document indexed in Elasticsearch belongs to an index.
4. **True or False:** Using the Bulk API is more efficient than sending multiple, separate requests.
5. What happens if you attempt to index a new document into an index that does not exist?
6. Can you **PUT** a document into an index without specifying an ID?



# Lab 1

## An Introduction to Elasticsearch

Yasaswy Raval - 24-Apr-2017 Capital One

# Chapter 2

# The Search API

Yasaswy Raval - 24-Apr-2017 - Capital One

- 1 Introduction to Elasticsearch
- 2 The Search API**
- 3 Text Analysis
- 4 Mappings
- 5 More Search Features
- 6 The Distributed Model
- 7 Working with Search Results
- 8 Aggregations
- 9 More Aggregations
- 10 Handling Relationships

# Topics covered:

- Introduction to the Search API
- URI Searches
- Request Body Searches
- Relevance
- The match Query
- The match\_phrase Query
- The range Query
- The bool Query
- Source Filtering

Yasaswy Raval - 24-Apr-2017 - Capital One



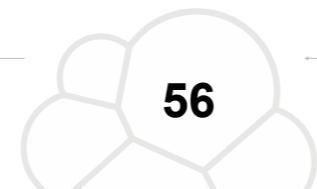
# Introduction to the Search API

Yasaswy Raval - 24-Apr-2017 - Capital One

# The Search API

- The **Search API** allows you to execute a search query and get back search hits that match the query
  - You used the Search API in Lab 1 when you ran a simple `_search` query
- Elasticsearch provides a full **Query DSL** (Domain Specific Language) to define queries
- Two types of searches:
  - URI searches
  - Request body search

Yasaswy Raval - 24-Apr-2017 - Capital One



# URI Searches

Yasaswy Raval - 24-Apr-2017 - Capital One

# URI Searches

- You can submit a search request using a URI and providing request parameters
  - Use a “q” to specify the query string
  - Uses a special syntax called “query string syntax”

“q” is for query string

```
curl -XGET 'http://localhost:9200/my_index/_search?q=comment:hollywood'
```

search the “comment”  
field for “hollywood”

# Why URI Searches?

- We will not discuss URI searches in detail
  - The documentation has a good discussion on the query string syntax and its many options
- URI searches are limited
  - Not all search options are exposed
- But, it is good to know that URI searches exist
  - They can be handy for a quick curl query
- We will focus on request body searches...



# Request Body Searches

Yasaswy Raval - 24-Apr-2017 / Capital One

# Request Body Searches

- To take full advantage of the Search API, write your queries as **request body searches** using the Query DSL

```
GET index_to_search/_search
{
  "query": {
    define your query here...
  }
}
```

Yasaswy Raval - 24th April 2017 - Capital One

the name of the index to be searched

\_search routes to the Search API

the **query** element is where you define your query

The diagram illustrates a GET request to the '\_search' endpoint of an index. The URL is 'index\_to\_search/\_search'. A callout box points to the '\_search' part of the URL with the text '\_search routes to the Search API'. Another callout box points to the 'query' field in the request body with the text 'the query element is where you define your query'. A third callout box points to the 'index\_to\_search' part of the URL with the text 'the name of the index to be searched'.

# Specifying Search Indices

- A search can be performed over multiple indices:

Syntax	Indices searched
<code>/_search</code>	<b><i>everything</i></b>
<code>/index-1/_search</code>	<b><i>only index-1</i></b>
<code>/index-1/type-1/_search</code>	<b><i>only type-1 docs in index-1</i></b>
<code>/index-1,index-2/_search</code>	<b><i>all of index-1 and index-2</i></b>
<code>/index-*/_search</code>	<b><i>all indices that start with index-*</i></b>



# The Search API

- The **Search API** allows you to execute a search query and get back search hits that match the query
- We will start by looking at the matching queries:
  - **match**
  - **match\_phrase**
- Then we will discuss two other commonly-used queries:
  - **range**
  - **bool**

# Relevance

Yasaswy Raval - 24-Apr-2017 - Capital One

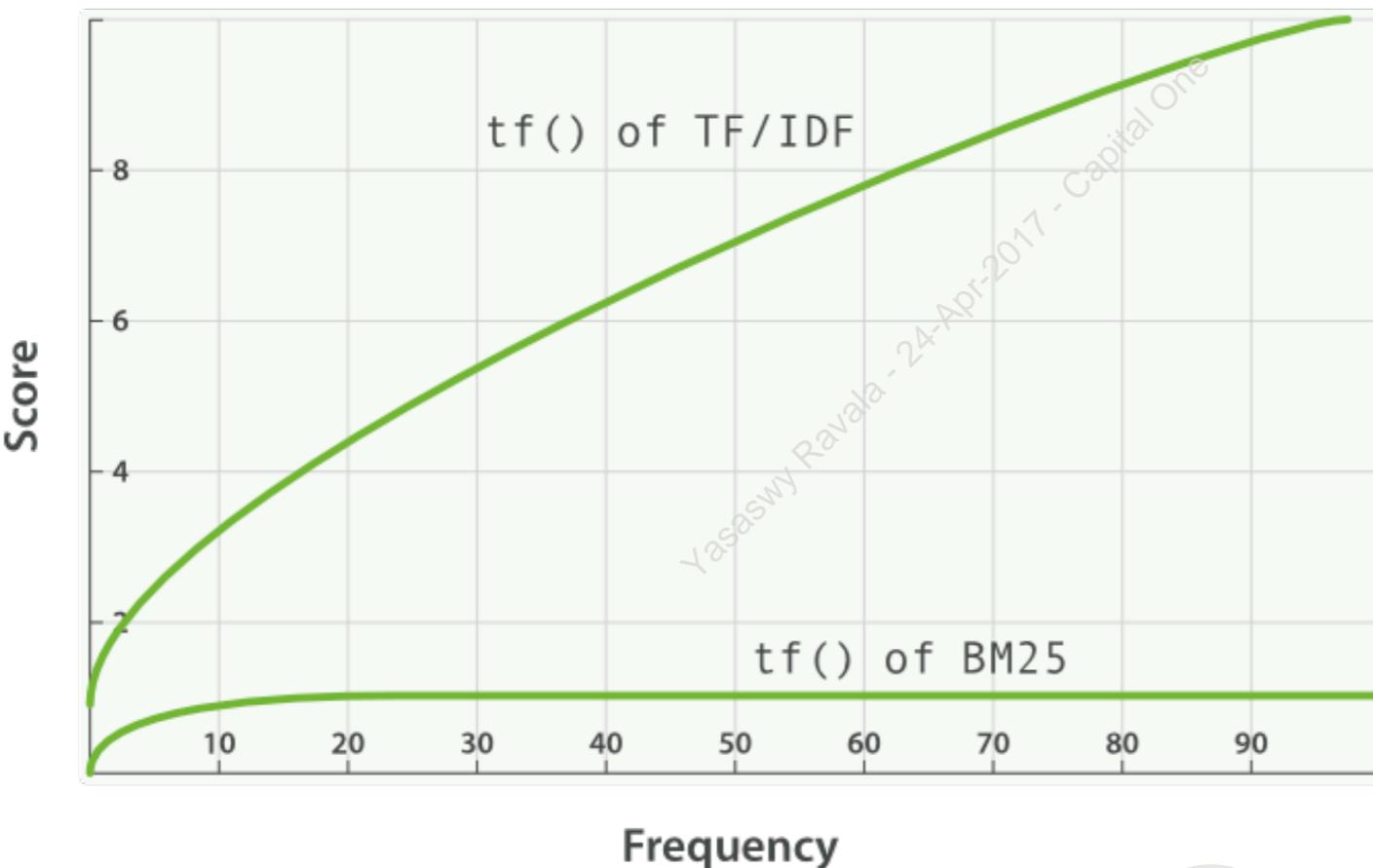
# What is Relevance?

- **Relevance** refers to the **scoring of a document** based on how closely it matches the query
  - The **\_score** is computed for each document that is a hit
- 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



# The BM25 Algorithm

- There is some fairly complex mathematics happening behind each search
  - An effective search is only as good as the underlying algorithm used to compute the \_score
- **BM25** (as in “**Best Matching**”) is the 25th iteration of tweaking the formula



BM25 assigns less weight to the frequency of a term after a certain number of occurrences

# The match Query

Yasaswy Raval - 24-Apr-2017 - Capital One

# 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
  - The main use case for the **match** query is for full-text search
- The syntax looks like:

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

The field you want to  
search

The text you want to  
search for

# Example of match

- Let's search the “comment” fields of the documents in **my\_index**

*Find all documents that have “**favorite**” or “**star**” in the comment.*

```
GET my_index/_search
{
  "query": {
    "match": {
      "comment": "favorite star"
    }
  }
}
```

# Query Response

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

```
"hits": {  
    "total": 3,  
    "max_score": 0.5446649,  
    "hits": [  
        {  
            "_index": "my_index",  
            "_type": "doc",  
            "_id": "3",  
            "_score": 0.5446649, ← Each hit has a _score  
            "_source": {  
                "username": "eniko",  
                "comment": "My favorite movie star in Hollywood is Harrison Ford."  
            }  
        },  
        {  
            "_index": "my_index",  
            "_type": "doc",  
            "_id": "1",  
            "_score": 0.53484553, ← Results are sorted by _score (by default)  
            "_source": {  
                "username": "harrison",  
                "comment": "My favorite movie is Star Wars!"  
            }  
        },  
        {  
    ]  
}
```



# Changing match to “and”

- Add the **operator** property and specify “and” or “or”
  - You have to add a “query” property as well

*Find all documents that have “favorite” **and** “star” in the comment.*

```
GET my_index/_search
{
  "query": {
    "match": {
      "comment": {
        "query": "favorite star",
        "operator": "and"
      }
    }
  }
}
```

Notice the slightly different syntax

```
"hits": {
  "total": 2,
  "max_score": 0.5446649,
  "hits": [
    {
      "_index": "my_index",
      "_type": "doc",
      "_id": "3",
      "_score": 0.5446649,
      "_source": {
        "username": "eniko",
        "comment": "My favorite movie star in Hollywood is Harrison Ford."
      }
    },
    {
      "_index": "my_index",
      "_type": "doc",
      "_id": "1",
      "
```



# The `minimum_should_match` Property

- If a match query searches on multiple terms, the “or” or “and” options might be too strict
- You can specify a `minimum_should_match` value
  - `minimum_should_match` can be used to trim the long tail of practically irrelevant results

```
GET my_index/_search
{
  "query": {
    "match": {
      "comment": {
        "query": "my favorite movie",
        "minimum_should_match": 2
      }
    }
  }
}
```

*Find comments that contain at least 2 terms out of “my favorite movie”*

Can be an integer or a percentage



# The match\_phrase Query

Yasaswy Raval - 24-Apr-2017 Capital One

# 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.

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

The field you want to search

The phrase you are searching for

# Example of match\_phrase

- Let's look at an example:

*I am looking for  
“**north star**” in the  
comments*

```
GET my_index/_search
{
  "query": {
    "match_phrase": {
      "comment": "north star"
    }
  }
}
```

Two things must happen for “**north star**” to cause a hit:

- “**north**” and “**star**” must appear in the “**comment**” field
- The position of “**star**” must be 1 greater than “**north**”

# match\_phrase vs. match

- Notice the difference between **match\_phrase** and **match** when searching for “north star”:

“**match\_phrase**” from previous slide only has 1 hit

```
"hits": {  
  "total": 1,  
  "max_score": 0.51623213,  
  "hits": [  
    {  
      "_index": "my_index",  
      "_type": "doc",  
      "_id": "2",  
      "_score": 0.51623213,  
      "_source": {  
        "username": "maria",  
        "comment": "The North  
Star is right above my house."  
      }  
    }  
  ]  
}
```

“**match**” uses a simple “or” logic

```
GET my_index/_search  
{  
  "query": {  
    "match": {  
      "comment": "north star"  
    }  
  }  
}
```

```
"hits": {  
  "total": 3,  
  "max_score": 0.51623213,  
  "hits": [
```

all 3 documents hit

# The slop Parameter

- If **match\_phrase** is too strict, you can introduce some flexibility into the phrase (called **slop**)
  - The **slop** parameter tells how far apart terms are allowed to be while still considering the document a match

```
GET my_index/_search
{
  "query": {
    "match_phrase": {
      "comment": {
        "query": "favorite star",
        "slop": 1
      }
    }
  }
}
```

*I am searching for  
“favorite star” in the  
comments*

Two things must happen for “favorite star” to find a hit:

1. “favorite” and “star” must appear in the “comment” field
2. The distance between “star” and “favorite” must be less than 2



# Example of slop

- The search for “**favorite star**” found a hit
  - The comment contains “**favorite movie star**”
- Without the **slop** set to at least 1, there would be no hits

```
GET my_index/_search
{
  "query": {
    "match_phrase": {
      "comment": {
        "query": "favorite star",
        "slop": 1
      }
    }
  }
}
```



```
"hits": {
  "total": 1,
  "max_score": 0.346985,
  "hits": [
    {
      "_index": "my_index",
      "_type": "doc",
      "_id": "3",
      "_score": 0.346985,
      "_source": {
        "username": "eniko",
        "comment": "My favorite
movie star in Hollywood is Harrison
Ford."
      }
    }
  ]
}
```



# The range Query

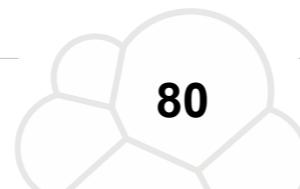
Yasaswy Raval - 24-Apr-2017 - Capital One

# The range Query

- The *range* query is for finding documents with fields that fall in a given range
  - Typically used for numeric and date fields, but can be used for text as well
- The syntax for range looks like:

```
GET index/_search
{
  "query": {
    "range": {
      "FIELD": {
        "gte": LOWER,
        "lte": UPPER
      }
    }
  }
}
```

You can use “gt” or “lt” as well



# Example of range

- The documents in the **stocks** index have a field named “**open**” that represents the opening price for that day

*Yogeshwari Ravula - 24-Apr-2017 - Capital One*

*Find all stocks with an opening price between \$50 and \$60*

```
GET stocks/_search
{
  "query": {
    "range": {
      "open": {
        "gt": 50.00,
        "lt": 60.00
      }
    }
  }
}
```

```
"hits": {
  "total": 12829,
  "max_score": 1,
  "hits": [
    ...
  ]
}
```

12,829 stocks in that range

# The range Query on Dates

- Range queries on dates can use different formats:
  - Specify the dates as a string
  - Use a special syntax called “date math”

```
GET stocks/_search
{
  "query": {
    "range": {
      "trade_date": {
        "gte": "2009-08",
        "lt": "2009-09"
      }
    }
  }
}
```

*I am looking for all stock prices from August, 2009.*



# Date Math

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

*I want all stock prices from the last month*

```
GET stocks/_search
{
  "query": {
    "range": {
      "trade_date": {
        "gte": "now-1M"
      }
    }
  }
}
```

*I want all stock prices from the last 24 hours*

```
GET stocks/_search
{
  "query": {
    "range": {
      "trade_date": {
        "gte": "now-1d"
      }
    }
  }
}
```

# Date Math Expressions

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

<b>y</b>	<b>years</b>
<b>M</b>	<b>months</b>
<b>w</b>	<b>weeks</b>
<b>d</b>	<b>days</b>
<b>h or H</b>	<b>hours</b>
<b>m</b>	<b>minutes</b>
<b>s</b>	<b>seconds</b>

<b>If now = 2016-12-08T12:56:23</b>	
now-1h	2016-12-08T11:56:23
now+1d	2016-12-09T12:56:23
now+1h+30m	2016-12-08T14:26:23
2017-01-15    +1M	Jan 15, 2017, plus 1 month



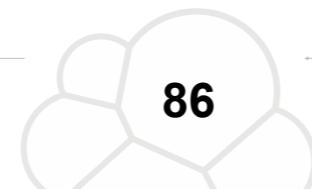
# The bool Query

Yasaswy Raval - 24-Apr-2017 - Capital One

# The bool Query

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

<b>must</b>	The query <b>must</b> appear in matching documents and will contribute to the <code>_score</code>
<b>must_not</b>	The query <b>must not</b> appear in matching documents
<b>should</b>	The query <b>should</b> optionally appear in matching documents, and matches contribute to the <code>_score</code>
<b>filter</b>	The query <b>must</b> appear in matching documents, but it will have no effect on the <code>_score</code>



# Syntax for bool

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

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

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

We will discuss filters in Chapter 7



# The must clause

```
GET my_index/_search
{
  "query" : {
    "bool" : {
      "must" : [
        {"match" : {"comment" : "star"}},
        {"match" : {"username" : "harrison"}}
      ]
    }
  }
}
```

*I am looking for  
comments from “harrison”  
that contain “star”*

"My favorite movie is Star Wars!"



# Multiple bool Clauses

```
GET my_index/_search
{
  "query": {
    "bool": {
      "must": [
        { "match": { "comment": "star"} },
        "must_not": [
          { "match": { "comment": "wars"} }
        ]
      }
    }
  }
}
```

*Find any comments  
that mention “star” but not  
“wars”*

"My favorite movie star in Hollywood is Harrison Ford."

"The North Star is right above my house."

# The should Clause

- A term that “**should**” match increases the `_score`
  - the score from each matching **must** or **should** clause are added together to provide the final `_score` for each document

*The comments I am  
searching for must have “star”  
and should have “wars”*

```
GET my_index/_search
{
  "query": {
    "bool": {
      "must": [
        { "match": { "comment": "star"} },
        "should": [
          { "match": { "comment": "wars"} }
        ]
      }
    }
  }
}
```

# minimum\_should\_match with bool

- You can use `minimum_should_match` in a `bool` query

```
GET my_index/_search
{
  "query": {
    "bool": {
      "must": [
        { "match": { "comment": "star"}},
        "should": [
          { "match": { "comment": "favorite"}},
          { "match": { "comment": "hollywood"}},
          { "match": { "comment": "famous"}},
          { "match": { "username": "harrison"}}
        ],
        "minimum_should_match": "50%"
      }
    }
  }
}
```



# Results from minimum\_should\_match

```
"hits": [
  {
    "_index": "my_index",
    "_type": "doc",
    "_id": "1",
    "_score": 0.82252765,
    "_source": {
      "username": "harrison",
      "comment": "My favorite movie is Star Wars!"
    }
  },
  {
    "_index": "my_index",
    "_type": "doc",
    "_id": "3",
    "_score": 0.8169974,
    "_source": {
      "username": "eniko",
      "comment": "My favorite movie star in Hollywood is Harrison Ford."
    }
  }
]
```

“harrison” + “favorite” >= 50%

“hollywood” + “favorite” >= 50%

# Searching tip for phrases

- You can take advantage of the boost that a **should** clause provides by combining **match** and **match\_phrase** when searching for a phrase:

```
GET my_index/_search
{
  "query": {
    "bool": {
      "must": [
        {"match": {
          "comment": "movie star"
        }}
      ],
      "should": [
        {"match_phrase": {
          "comment": "movie star"
        }}
      ]
    }
  }
}
```

Hits will include “**movie**” or  
“**star**”, but comments that  
include the phrase “**movie star**”  
will score higher

# Only “should” Query

- In a **bool** query with no **must** or **filter** clauses, one or more **should** clauses must match a document:
  - or you can set **minimum\_should\_match**

```
GET my_index/_search
{
  "query": {
    "bool": {
      "should": [
        {"match": {"comment": "north"}},
        {"match": {"comment": "south"}},
        {"match": {"comment": "east"}},
        {"match": {"comment": "west"}}
      ]
    }
  }
}
```

At least one clause  
must match for a hit



# Source Filtering

Yasaswy Raval - 24-Apr-2017 - Capital One

# Requesting Specific Fields

- You can request specific fields using the `_source` parameter
  - separate multiple fields by commas

```
GET stocks/_search?_source=open,close
{
  "query": {
    "range": {
      "open": {
        "gte": 50.00,
        "lte": 60.00
      }
    }
  }
}
```



```
"hits": [
  {
    "_index": "stocks",
    "_type": "stock_type",
    "_id": "AVmAGxB992cAvoluWZvm",
    "_score": 1,
    "_source": {
      "close": 57.1,
      "open": 57.58
    }
  },
  ...
]
```



# `_source` in a Query

- You can also specify the source fields within the `query` body using `_source`
  - this query has the same output as the previous slide:

```
GET stocks/_search
{
  "_source": ["open", "close"],
  "query": {
    "range": {
      "open": {
        "gte": 50.00,
        "lte": 60.00
      }
    }
  }
}
```

Only return the “open”  
and “close” fields



# The includes and excludes Patterns

- For complete control, you can specify which fields to include and exclude within `_source`:
  - notice also that you can use wildcards in a `_source` field

```
GET stocks/_search
{
  "_source": {
    "includes": [ "volume", "*date*" ],
    "excludes": [ "trade_date_text" ]
  },
  "query": {
    "range": {
      "open": {
        "gte": 50.00,
        "lte": 60.00
      }
    }
  }
}
```

```
{
  "_index": "stocks",
  "_type": "stock_type",
  "_id": "AVmAGxB992cAvoluWZvn",
  "_score": 1,
  "_source": {
    "volume": 48749,
    "trade_date": "2010-06-10T06:00:00.000Z"
  }
},
```



# Chapter Review

Yasaswy Raval - 24-Apr-2017 - Capital One

# Summary

- The ***match*** query is a simple but powerful search for fields that are text, numerical values, or dates
- ***match\_all*** matches all documents in the searched indices, and returns 10 by default
- The **slop** parameter tells how far apart terms are allowed to be while still considering the document a match
- The ***range*** query is for finding documents with fields that fall in a given range
- A ***bool*** query is a combination of one or more of the following boolean clauses: **must**, **must\_not**, **should** and **filter**
- You can request specific fields using the ***\_source*** parameter

# 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 would you write a date range that matches the last 12 hours?
4. Suppose you have a “**should**” clause in a “**bool**” with 5 queries, and you set **minimum\_should\_match** to 70%. How many of the “**should**” clauses need to match for a hit?
5. What gets returned in the following request?

```
GET my_index/doc/1/_source?_source_include=*e
```

# Lab 2

## The Search API

Yasaswy Raval - 24-Apr-2017 - Capital One

# Chapter 3

# Text Analysis

Yasaswy Raval - 24-Apr-2017 - Capital One

- 1 Introduction to Elasticsearch
- 2 The Search API
- 3 Text Analysis
- 4 Mappings
- 5 More Search Features
- 6 The Distributed Model
- 7 Working with Search Results
- 8 Aggregations
- 9 More Aggregations
- 10 Handling Relationships

# Topics covered:

- What is Analysis?
- Building an Inverted Index
- Analyzers
- Custom Analyzers
- Character Filters
- Tokenizers
- Token Filters
- Defining Analyzers
- Synonyms
- How to Choose an Analyzer
- Segments

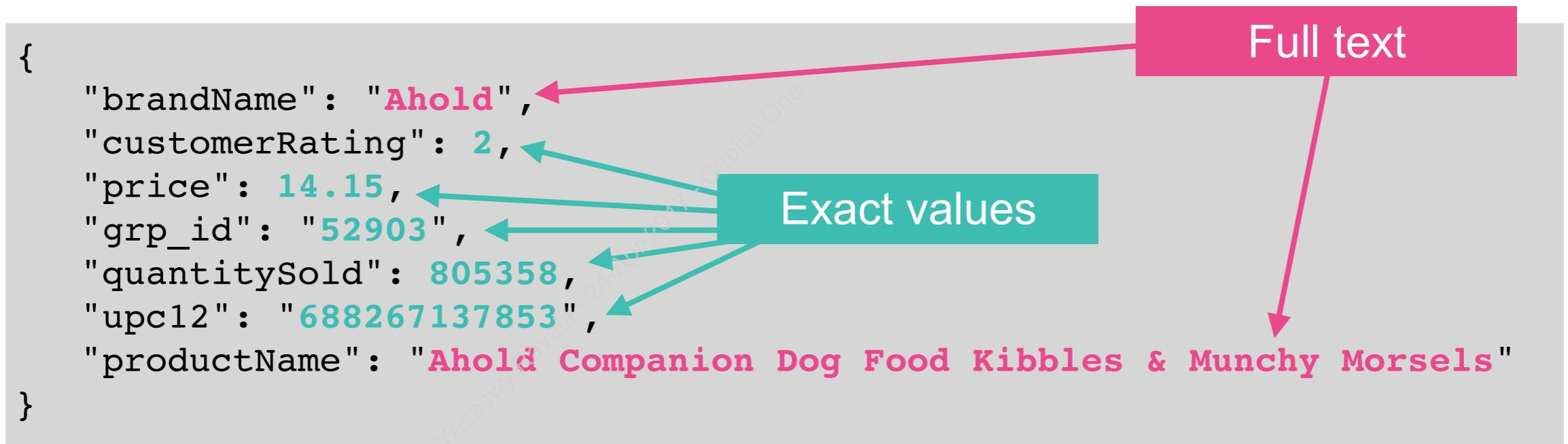
Yasaswy Raval - 24-Apr-2017 - Capital One

# What is Analysis?

Yasaswy Raval - 24-Apr-2017 - Capital One

# Exact Values vs. Full Text

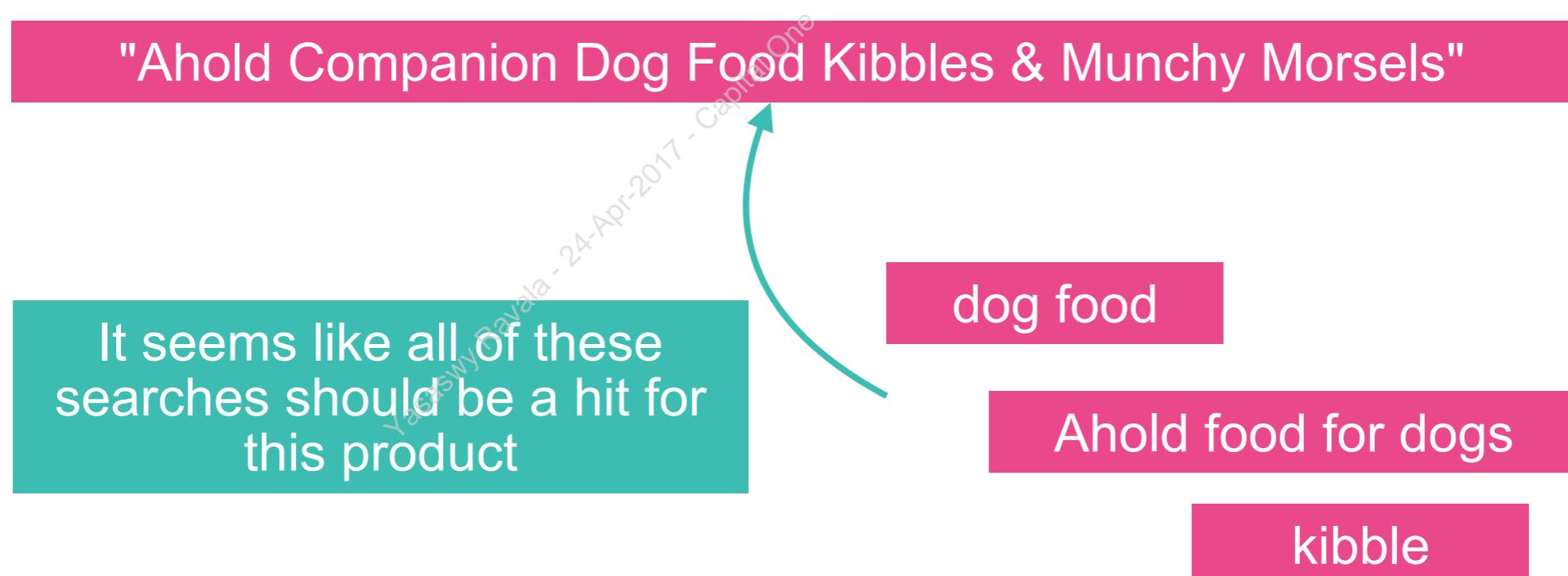
- In general, data in Elasticsearch can be categorized into two groups:
  - Exact values:** includes numeric values, dates, and certain strings
  - Full text:** unstructured text data that we want to search



- Why does it matter if a value is **exact** or **full text**?

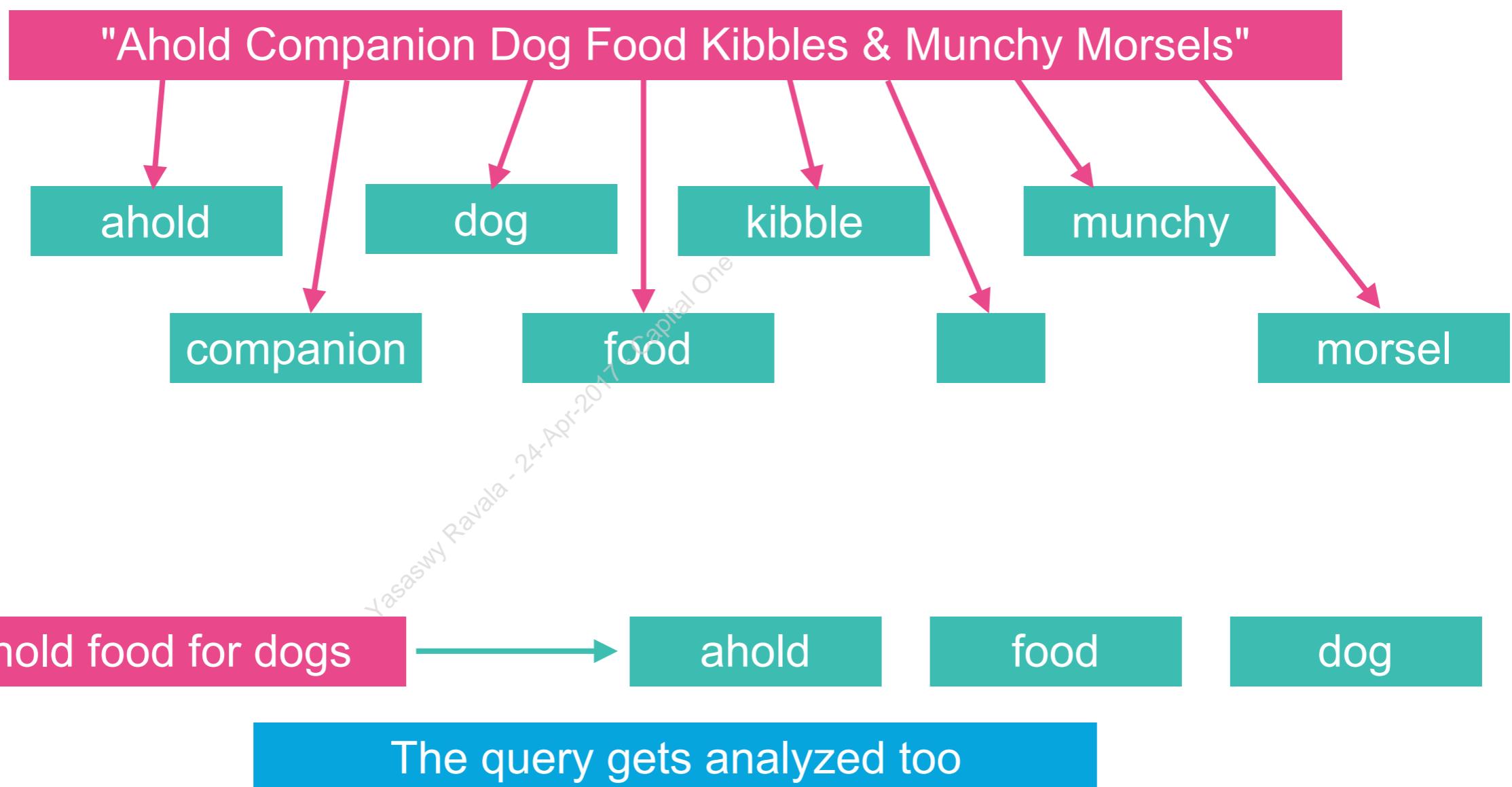
# Full Text is Analyzed

- **Analysis** is the process of converting full text into **terms** for the inverted index
- Analysis is performed by an **analyzer**
  - either a built-in analyzer
  - or a custom analyzer you configure



# Analysis Makes Full Text Searchable

- Full text gets *analyzed* during ingestion
- Similarly, the text in your queries gets analyzed using the same analyzer



# Building an Inverted Index

Yasaswy Raval - 24-Apr-2017 - Capital One

# What is an Inverted Index?

- You are familiar with the index in the back of a book
  - Useful terms from the book are sorted, and page numbers tell you where to find that term
- Lucene creates a similar inverted index with your documents

Ariel motorbikes 274  
Ariel Motors, Birmingham 275  
Armstrong Siddeley motor cars 304  
around-the world cycle ride 203–4  
Arroyo Seco Parkway, California 173  
*Art and Pastime of Cycling, The* 87–8, 209  
*Art of Cycling, The* 211  
asphalt xx, 43–4, 112–13, 114–15, 120–3, 123–4  
assembly-line manufacturing 195–6  
Aster company 275  
Aston Martin company 275

Page 275 contains information about “Aston Martin company”



# Building an Inverted Index

- Let's look at how terms are analyzed and added to an inverted index:
  - text is broken into tokens
  - indexed by a unique document id

1 The quick brown fox jumps over the lazy dog

2 Fast jumping spiders



token	postings
Fast	2
The	1
brown	1
dog	1
fox	1
jumping	2
jumps	1
lazy	1
over	1
quick	1
spiders	2
the	1



# Lowercase

- Searching for “Fast” and “fast” should yield the same results
  - common to lowercase all tokens

1 **The** quick brown fox jumps over the lazy dog

2 **Fast** jumping spiders

“**the**” was already  
indexed for doc 1

token	postings
brown	1
dog	1
<b>fast</b>	2
fox	1
jumping	2
jumps	1
lazy	1
over	1
quick	1
spiders	2
<b>the</b>	1

# Stopwords

- Searching for “the” rarely makes sense
  - typically we do not bother to index “the”
  - or other **stopwords**

- 1 **The** quick brown fox jumps over **the** lazy dog
- 2 Fast jumping spiders

token	postings
brown	1
dog	1
fast	2
fox	1
jumping	2
jumps	1
lazy	1
over	1
quick	1
spiders	2



# Stemming

- “jumps” and “jumping” share the same stem: “jump”
  - want “jump” to match both
  - so only index the stem

1 The quick brown fox **jumps** over the lazy dog

2 Fast **jumping spiders**

token	postings
brown	1
dog	1
fast	2
fox	1
<b>jump</b>	<b>1,2</b>
lazy	1
over	1
quick	1
<b>spider</b>	<b>2</b>

# Synonyms

- The terms “**quick**” and “**fast**” have similar meaning
  - Should a search for “**quick**” return doc 2?
  - Should a search for “**fast**” return doc 1?

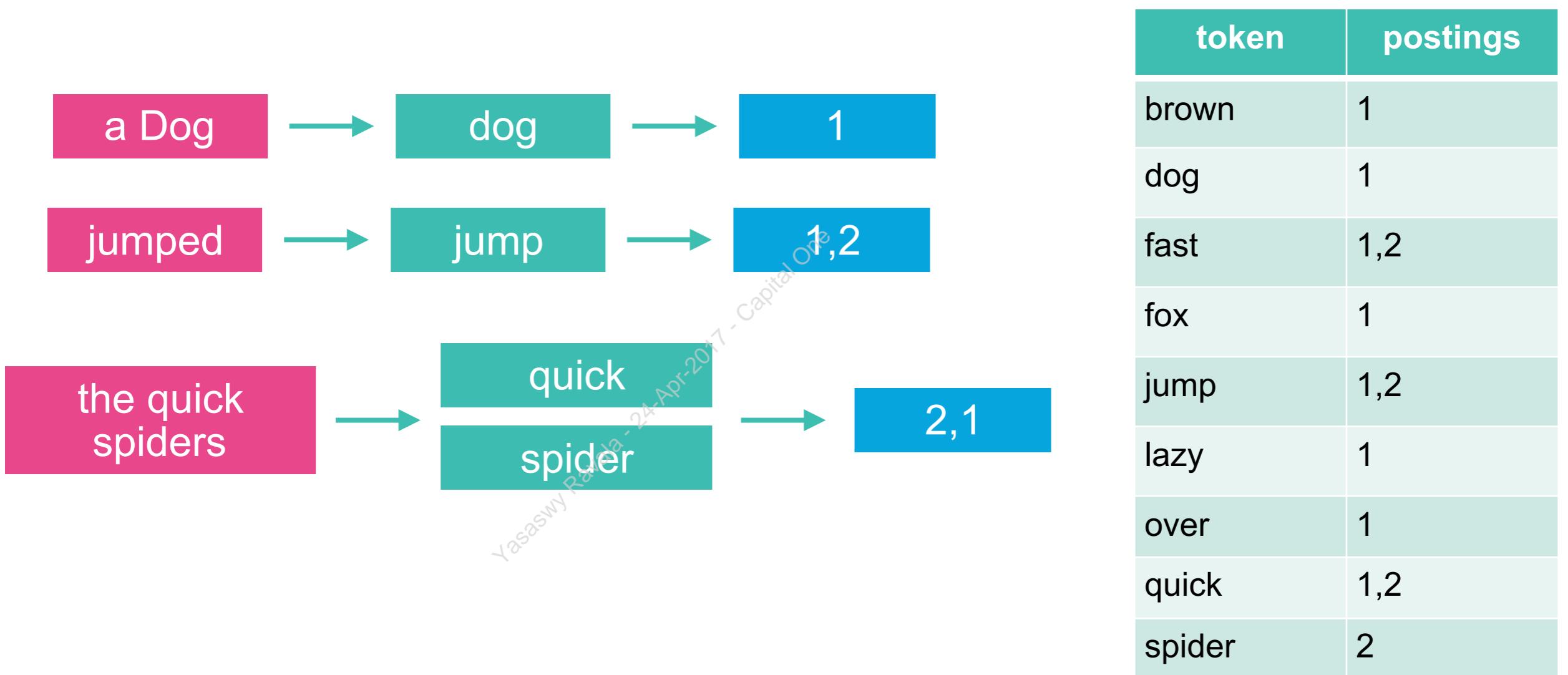
1 The **quick** brown fox jumps over the lazy dog

2 **Fast** jumping spiders

token	postings
brown	1
dog	1
<b>fast</b>	1,2
fox	1
jump	1,2
lazy	1
over	1
<b>quick</b>	1,2
spider	2

# Analyze the Search Query Too

- If we are going to analyze our text before indexing, we better also analyze our search queries in the same manner

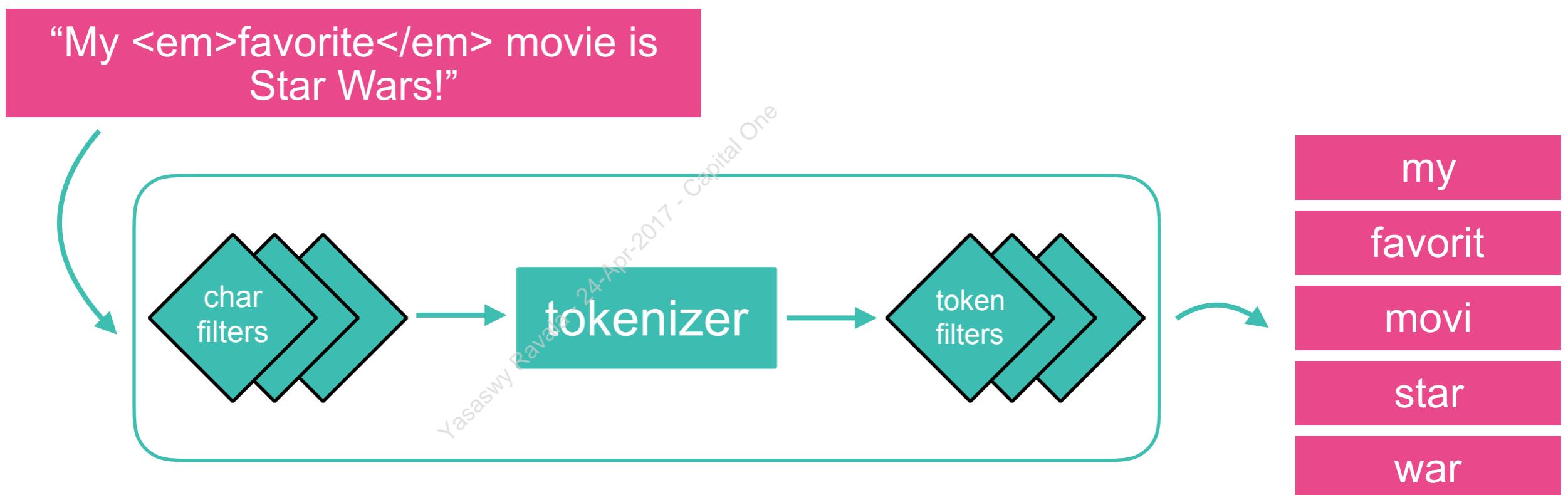


# Analyzers

Yasaswy Raval - 24-Apr-2017 - Capital One

# Anatomy of an Analyzer

- An analyzer consists of three parts:
  1. Character Filters
  2. Tokenizer
  3. Token Filters



# Pre-built Analyzers

- standard
- simple
- whitespace
- stop
- keyword
- pattern
- language

<https://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-analyzers.html>



# The Analyze API

- Use the `_analyze` API to test what an analyzer will do to your text:

Use the `_analyze` API

```
GET _analyze
{
  "analyzer": "simple",
  "text": "My favorite movie is Star Wars!"
}
```

Test the “simple” analyzer on the given “text”

“My favorite movie is Star  
Wars!”

“simple”  
analyzer

my  
favorite  
movie  
is  
star  
wars

# The simple Analyzer

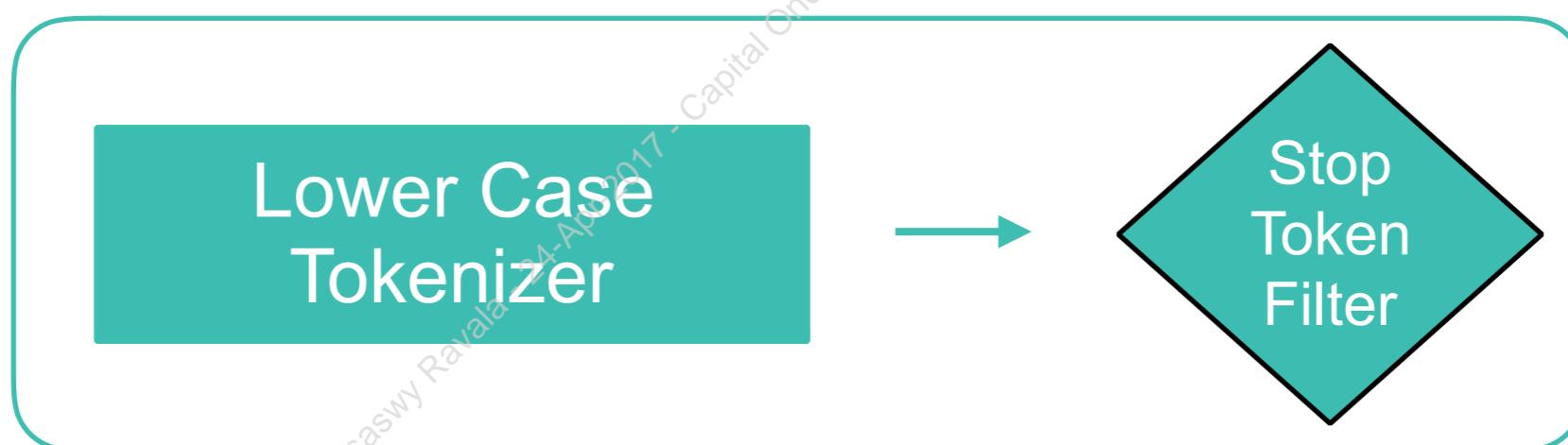
- Notice the simple analyzer does two things:
  - Breaks text into terms whenever it encounters a character which is not a letter
  - All terms are lower-cased

Lower Case  
Tokenizer

The simple analyzer has no character or token filters

# The stop Analyzer

- Notice the **stop analyzer** removes stop words
  - similar to the simple analyzer,
  - but adds support for removing stop words, which in English are:**a, an, and, are, as, at, be, but, by, for, if, in, into, is, it, no, not, of, on, or, such, that, the, their, then, there, these, they, this, to, was, will, with**

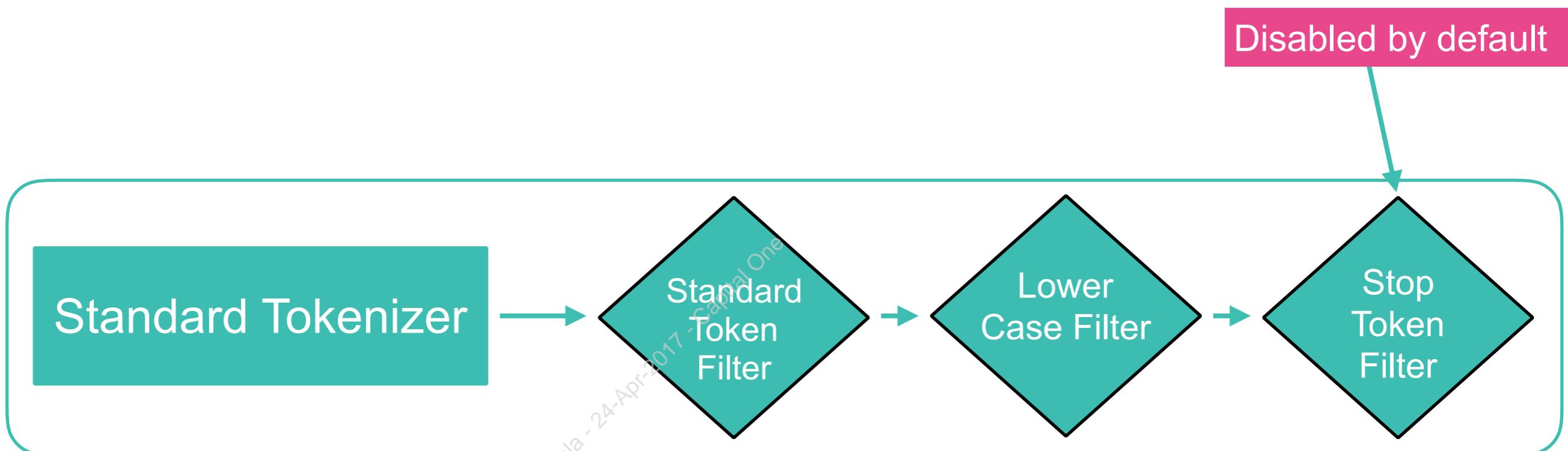


The stop analyzer includes the Stop Token Filter



# The standard Analyzer

- The default analyzer in Elasticsearch
  - Optionally, you can configure the stop words as well



# The language Analyzer

- A set of analyzers aimed at analyzing specific language text
  - 30+ languages supported

```
GET _analyze
{
  "analyzer": "english",
  "text": "My favorite movie is Star Wars!"
}
```

```
GET _analyze
{
  "analyzer": "french",
  "text": "Mon film préféré est Star Wars!"
}
```

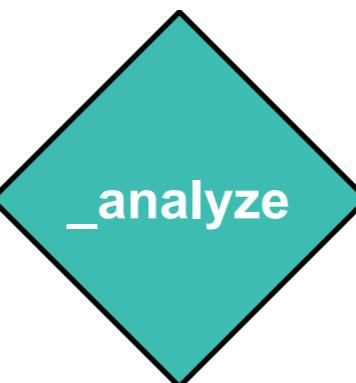
# The Analyze API

- The *Analyze API* can be used to test individual components
  - The following command tests a char filter and tokenizer

Test the “**keyword**” tokenizer with the “**html\_strip**” character filter on the given “**text**”

```
GET _analyze
{
  "tokenizer": "keyword",
  "char_filter": [ "html_strip" ],
  "text": "<strong>Welcome</strong> to the <em>jungle</em>"
}
```

"<strong>Welcome</strong>  
to the <em>jungle</em>"



"Welcome to the jungle"

# Character Filters

Yasaswy Raval - 24-Apr-2017 - Capital One®

# 1. Character Filters

- Preprocess characters before they are passed to the tokenizer
  - You can configure multiple character filters
- Three built-in character filters:
  - **html\_strip**
  - **mapping**
  - **pattern\_replace**

Yasaswy Raval - 24-Apr-2017 - Capital One

# The html\_strip Character Filter

- Strips out any HTML
- Replaces HTML entities with their decoded value

```
GET _analyze
{
  "tokenizer": "keyword",
  "char_filter": [ "html_strip" ],
  "text": "Tom&apos;s <em>favorite</em> movie is Star Wars!"
}
```

“Tom&apos;s <em>favorite</em> movie is Star Wars!”



# The mapping Character Filter

- Map characters to other characters before tokenizing

```
GET /_analyze
{
  "char_filter": [
    {
      "type": "mapping",
      "mappings": [ "C++ => cpp", "c++ => cpp" ]
    }
  ],
  "tokenizer": "standard",
  "text": "I like to code in C++ and Java."
}
```



# The pattern\_replace Character Filter

- Use a regular expression to manipulate characters in a string before processing

```
GET /_analyze
{
  "char_filter": [
    {
      "type": "pattern_replace",
      "pattern": "(\\d+)-(?=\\d)",
      "replacement": "$1_"
    }
  ],
  "tokenizer": "standard",
  "text": "123-45-6789"
}
```

Looks for dashes in a string of digits

Replaces the dash with an underscore

123-45-6789

pattern\_replace

123\_45\_6789

# Tokenizers

Yasaswy Raval - 24-Apr-2017 - CapitalOne

## 2. Tokenizer

- The second phase of an analyzer
  - Divides a stream of text into tokens
- Some of the built-in tokenizers include:
  - **whitespace**: divides text on any whitespace
  - **standard**: divides text into terms on word boundaries
  - **path\_hierarchy**: for hierarchical values like filesystem paths
  - **uax\_url\_email**: recognizes URLs/email addresses as tokens
  - **keyword**: a no-op tokenizer
  - **pattern**: splits text based on a given regular expression
- You can also write your own tokenizer in Java



# Token Filters

Yasaswy Raval - 24-Apr-2017 - Capital One

# 3. Token Filters

- **Token filters** are the third phase of an analyzer:
  - They can add tokens (e.g. the synonym filter)
  - They can remove tokens (e.g. stop words)
  - They can change tokens (e.g. stemming to root form)
- Examples include:
  - **lowercase**: make all text lower case
  - **snowball**: stems words using a Snowball-generated stemmer
  - **stop**: removes stop words
  - **nGram** and **edgeNGram**:
  - and many more...

# Order Matters in Token Filters

```
GET _analyze
{
  "tokenizer": "whitespace",
  "filter": ["lowercase", "stop"],
  "text": "To Be Or Not To Be"
}
```

[ ]

```
GET _analyze
{
  "tokenizer": "whitespace",
  "filter": [ "stop", "lowercase" ],
  "text": "To Be Or Not To Be"
}
```

to

be

or

not

to

be

# ICU Analysis

Yasaswy Raval - 24-Apr-2017 - CapitalOne

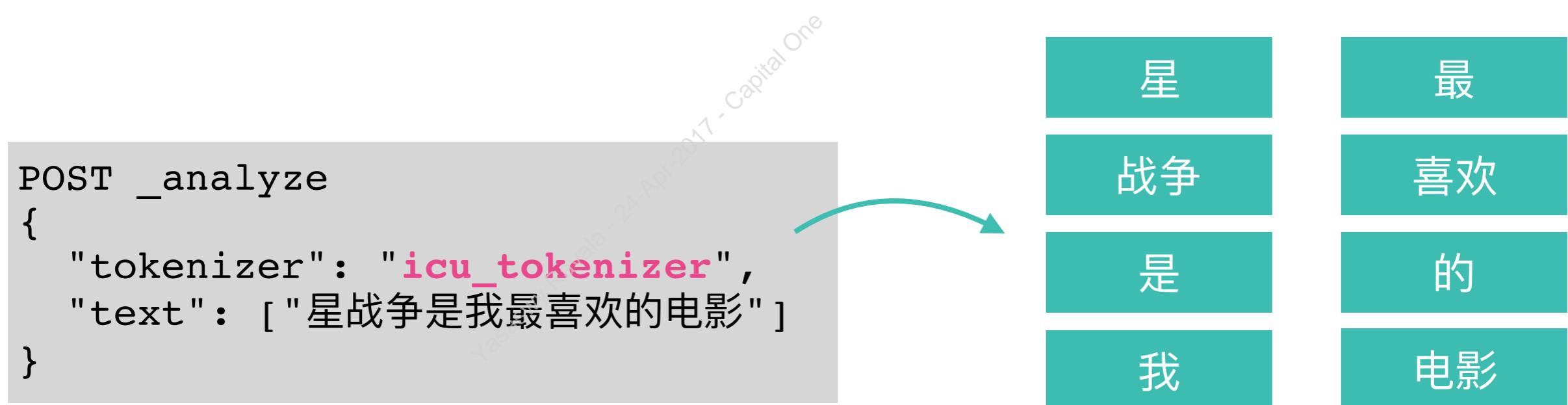
# ICU Analysis Plugin

- The *ICU Analysis plugin* adds extended Unicode support for text analysis in Elasticsearch
  - including better analysis of Asian languages
- The plugin consists of 1 character filter:
  - Normalization
- 1 tokenizer:
  - ICU Tokenizer
- and 4 token filters:
  - Normalization
  - Folding
  - Collation
  - Transform

Yasaswy Raval - 24-Apr-2017 - Capital One

# Example of ICU Tokenizer

- You can use the components of the ICU plugin just like any other character filter, tokenizer, or token filter
- For example, the following Chinese translates to “Star Wars is my favorite movie”
  - The “**standard**” tokenizer incorrectly splits the sentence into single characters



# Defining Analyzers

Yasaswy Raval - 24-Apr-2017 - CapitalOne

# Configuring a Custom Analyzer

- Analyzers are defined in the “analysis” part of an index’s settings

```
PUT test_index
{
  "settings": {
    "analysis": {
      "analyzer": {
        "my_custom_analyzer": {
          "type" : "custom",
          "char_filter" : ["html_strip"],
          "tokenizer" : "standard",
          "filter" : ["lowercase", "snowball"]
        }
      }
    }
  }
}
```

Whatever name you want to use for your analyzer

Set “type” to “custom”

Configure the three components of your custom analyzer



# Defining Custom Filters

- If your character or token filters need to define properties, then you have to define them differently:

```
{  
  "settings": {  
    "analysis": {  
      "char_filter": {  
        "cpp_filter": {  
          "type" : "mapping",  
          "mappings" : ["C++ => cpp", "c++ => cpp"]  
        }  
      },  
      "analyzer": {  
        "my_custom_analyzer": {  
          "char_filter": ["cpp_filter"],  
          "tokenizer" : "standard"  
        }  
      }  
    }  
  }  
}
```

Define a custom **char\_filter** and name it whatever you want

Use your custom filter later in the “analyzer” section



# More Advanced Example

- You can use your custom filters along with the built-in filters

```
{  
  "settings": {  
    "analysis": {  
      "char_filter": {  
        "cpp_filter" : {  
          "type" : "mapping",  
          "mappings" : ["C++ => cpp", "c++ => cpp"]  
        }  
      },  
      "filter": {  
        "my_stopwords" : {  
          "type" : "stop",  
          "stopwords" : ["my", "is"]  
        }  
      },  
      "analyzer": {  
        "my_custom_analyzer" : {  
          "char_filter": ["html_strip", "cpp_filter"],  
          "tokenizer" : "standard",  
          "filter" : ["lowercase", "my_stopwords", "snowball"]  
        }  
      }  
    }  
  }  
}
```



# Testing a Custom Analyzer

- To test an analyzer that is already defined for an index, use `_analyze` at the index level:

```
POST test_index/_analyze
{
  "analyzer": "my_custom_analyzer",
  "text" : "C++ is my favorite language."
}
```



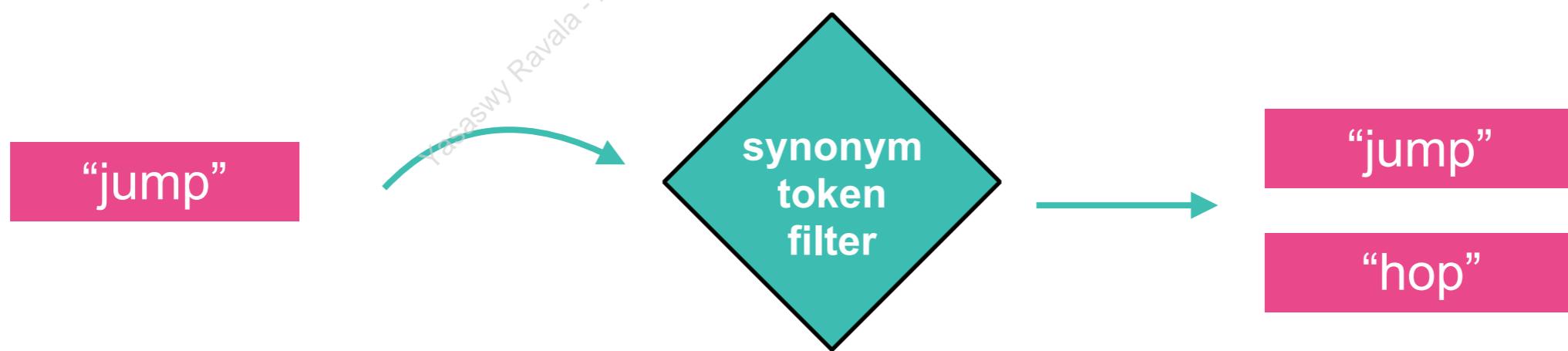
```
{
  "tokens": [
    {
      "token": "cpp",
      "start_offset": 0,
      "end_offset": 3,
      "type": "<ALPHANUM>",
      "position": 0
    },
    {
      "token": "favorit",
      "start_offset": 10,
      "end_offset": 18,
      "type": "<ALPHANUM>",
      "position": 3
    },
    {
      "token": "languag",
      "start_offset": 19,
      "end_offset": 27,
      "type": "<ALPHANUM>",
      "position": 4
    }
  ]
}
```

# Synonyms

Yasaswy Raval - 24-Apr-2017 - Capital One

# Working with Synonyms

- Searching for “jump” should probably match “hop” (and vice-versa)
  - The two words are ***synonyms***: two words with similar meaning
- You define synonyms using the ***synonym token filter***
  - You can define the synonyms inline in the token filter settings, or
  - Define synonyms in a separate text file



# Defining Synonyms

- Use the “synonym” filter as the “type”:

```
PUT test_index
{
  "settings" : {
    "analysis" : {
      "filter" : {
        "my_synonyms" : {
          "type" : "synonym",
          "synonyms" : [
            "jump, hop", "quick => fast"
          ]
        }
      },
      "analyzer" : {
        "my_analyzer" : {
          "tokenizer" : "whitespace",
          "filter" : ["lowercase", "snowball", "my_synonyms"]
        }
      }
    }
  }
}
```

“jump” and “hop” are synonyms

“quick” is a synonym for “fast”  
but not the other direction

# A Synonyms File

- Use the "**"synonyms\_path"**" property to specify the file
- The syntax is the same as the "**"synonyms"**" property:

```
"filter" : {  
    "my_synonyms" : {  
        "type" : "synonym",  
        "synonyms_path" : "/path/to/file.txt"  
    }  
}
```

```
#Separate synonyms by commas  
jump, hop  
u.s., u.s.a., united states  
car, vehicle, automobile  
  
#Use the arrow (=>) for mapping synonyms  
quick => fast
```

# How do we decide which analyzer to use?

Yasaswy Raval - 24-Apr-2017 - Capital One

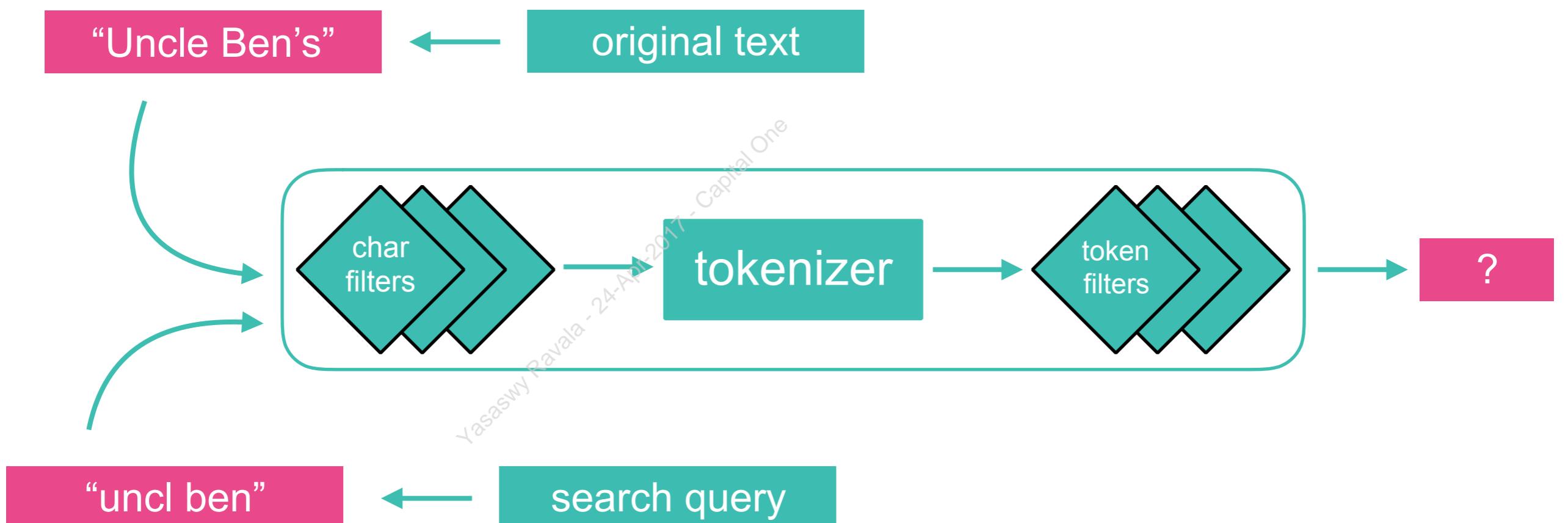
# Step 1: Exact value or full text?

- The first question you need to ask is: “*Which text fields are exact values and which fields are full text?*”
  - Exact values do not need to be analyzed

```
{  
  "brandName": "Uncle Ben's",  
  "customerRating": 3,  
  "price": 8.57,  
  "grp_id": "26333",  
  "quantitySold": 514342,  
  "upc12": "054800085453",  
  "productName": "Uncle Ben's Rice Pilaf Ready Rice"  
}
```

# Step 2: What will our searches look like?

- The second question is: “*What will a search look like and when should that match a document?*”
  - Look at specific examples in your dataset
  - Does “Uncle Ben’s” = “uncle bens” = “uncl ben” = “benjamin”?



# Step 3: Test several options

- There are still a lot of questions to ask
  - Where will the text be split?
  - Should we use stemming?
  - Should we build n-grams?
  - What are some critical synonyms?
- The most important step: **test your analyzer thoroughly!**
  - Try several options
  - Test a lot of searches!

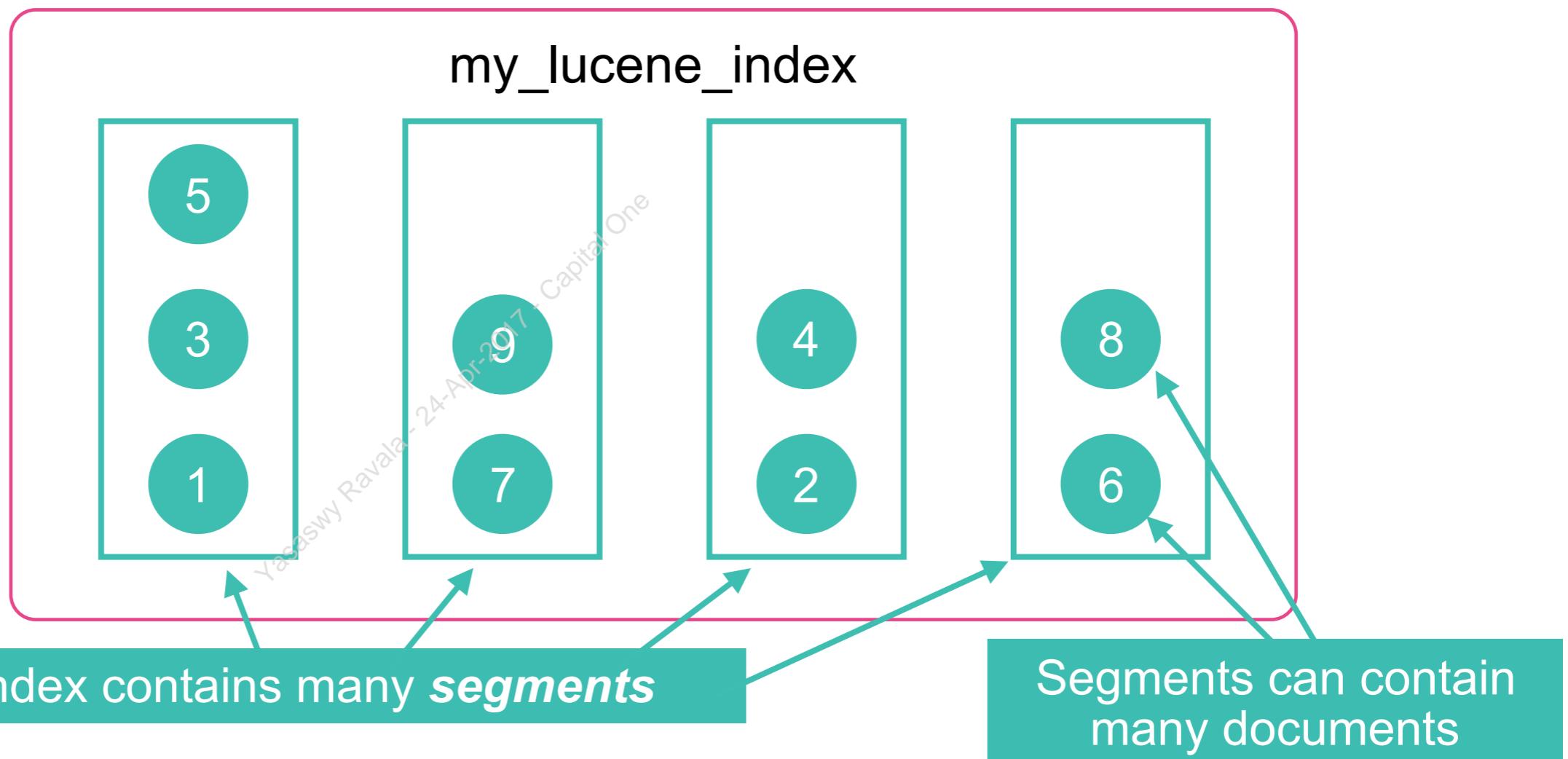


# Segments

Yasaswy Raval - 24-Apr-2017 - Capital One

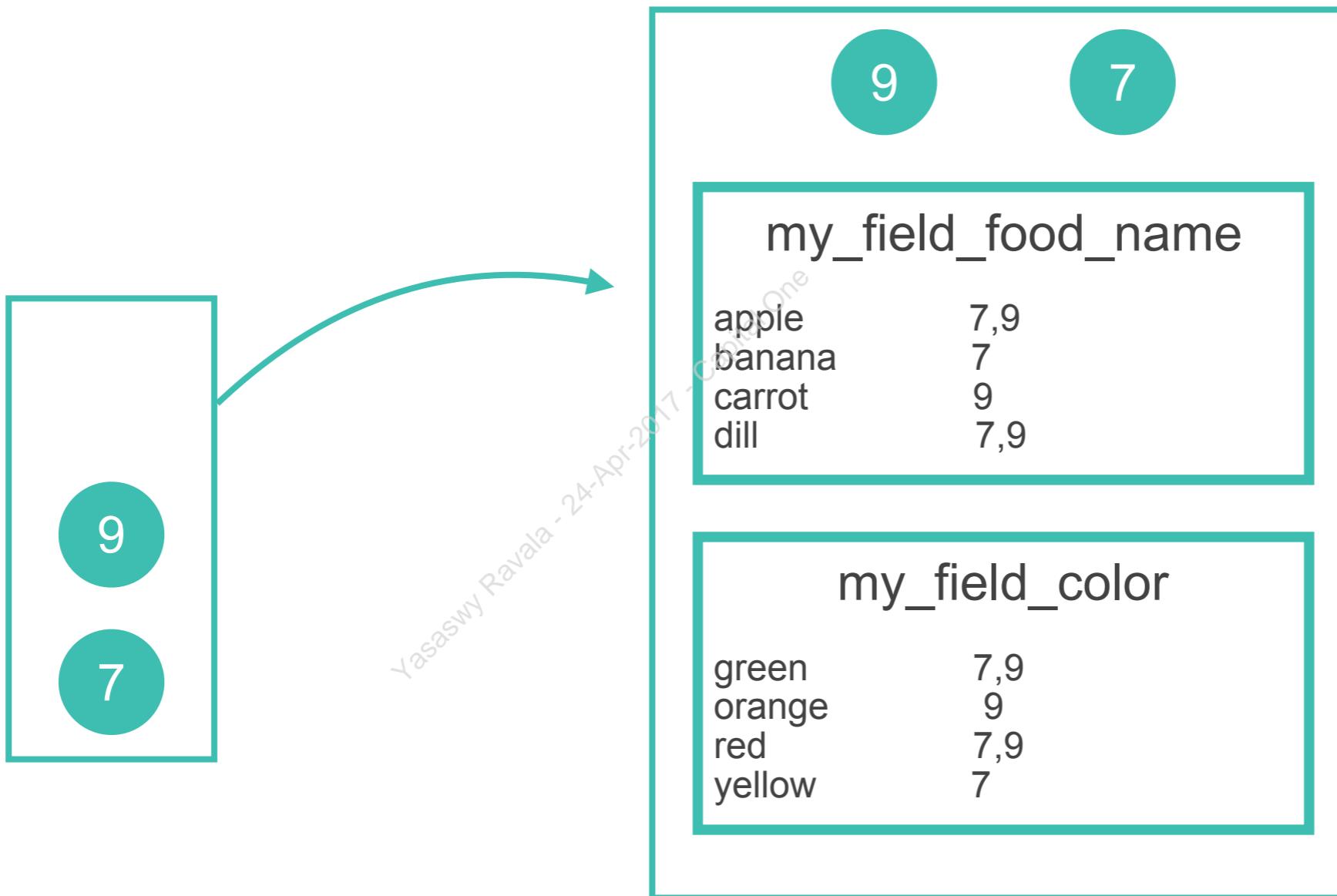
# Inverted Index in Practice

- The inverted index concept discussed in this section is practically represented inside a package called a Lucene **segment**
  - a self-contained immutable inverted index



# Segment is a Package

- A **segment** contains many documents
  - and is a package of **many data structures**,
  - including an **inverted index for each field** that is searchable



# Why Segments?

- Lucene **does not** build one giant inverted index
  - that would be too difficult to maintain and store
- Similarly, Lucene **does not** build an inverted index for each document after every indexing operation
  - that would be inefficient
- Segments provide a **nice solution** between having one large inverted index and too many small ones



# Segments are Refreshing

- The lightweight process of writing and opening a new segment is called a **refresh**
  - by default, every shard is refreshed once every second
  - configured at the index level with **refresh\_interval** property
- Documents are not searchable until a refresh occurs
  - you can force a refresh using the **\_refresh** endpoint, although this is **not a common use case**:

POST **my\_index/\_refresh**

Manually force a refresh, which makes newly-indexed documents searchable

# The refresh Parameter

- The Document APIs that create, update or delete have an optional **refresh** parameter:
  - controls when changes made by this request are made visible to searches
- Three possible values:
  - **false**: the default value, any changes to the document are *not* visible immediately
  - **true**: refresh the affected primary and replica shards so that the changes are visible immediately
  - **wait\_for**: synchronous request that waits for the changes to occur



# Example of the refresh Parameter

Wait for the indexing to complete before returning a response

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

- **wait\_for** does not force a refresh, but rather it waits for a refresh to happen
- **true** forces a refresh

# Chapter Review

Yasaswy Raval - 24-Apr-2017 - Capital One

# Summary

- Exact values include numeric values, dates, and certain strings
- Full text is unstructured text data that we want to search
- Analysis is the process of converting full text into terms for the inverted index
- analyzer = char\_filter to tokenizer to token filter
- Use the `_analyze` API to test an analyzer (or the components individually)
- The inverted index concept is represented inside a package called a Lucene **segment**
- A **segment** is a package of many data structures



# Quiz

1. The process of converting full text into terms for the inverted index is called \_\_\_\_\_.
2. Why does it matter if a field is an exact value vs. full text?
3. What are the three components that make up an analyzer?
4. What does the snowball token filter do to the word "happy"?
5. What is the effect of configuring "love => like" as a synonym?
6. **True or False:** You have to invoke `_refresh` occasionally while indexing a lot of documents to refresh the in-memory buffer.



# Lab 3

## Text Analysis

Yasaswy Raval - 24-Apr-2017 - Capital One

# Chapter 4

# Mappings

Yasaswy Raval - 24-Apr-2017 - Capital One

- 1 Introduction to Elasticsearch
- 2 The Search API
- 3 Text Analysis
- 4 Mappings
- 5 More Search Features
- 6 The Distributed Model
- 7 Working with Search Results
- 8 Aggregations
- 9 More Aggregations
- 10 Handling Relationships

# Topics covered:

- What is a Mapping?
- Dynamic Mappings
- Defining Explicit Mappings
- Adding Fields
- Dive Deeper into Mappings
- Specifying Analyzers
- Index Templates

Yasaswy Raval - 24-Apr-2017 - Capital One

# What is a Mapping?

Yasaswy Raval - 24-Apr-2017 - Capital One

# 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
- Mappings belong to index types...



# Remember Types?

- Documents are indexed into an index
- Each document also has a *type*
  - An index can have multiple types (although that is not encouraged)
  - Every type has exactly one mapping

```
{  
  "_index": "my_index",  
  "_type": "my_type",  
  "_id": "2",  
  "_score": 1,  
  "_source": {  
    "username": "maria",  
    "comment": "The North Star is right above my house."  
  }  
}
```

Every document has a type



# Every Type has One Mapping

- Defined in the “mappings” section of the index

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "properties": {
        define your fields and types here
      }
    }
  }
}
```

The name of your type

# Example of a Mapping

```
GET my_index/_mappings
```

**“comment” is a “text”**

**“username” is a “keyword”**

```
{  
  "my_index": {  
    "mappings": {  
      "my_type": {  
        "properties": {  
          "comment": {  
            "type": "text"  
          },  
          "username": {  
            "type": "keyword"  
          }  
        }  
      }  
    }  
  }  
}
```

Yasaswy Raval - 24-Apr-2017 - CapitalOne

# Defining a Mapping

- Mappings are defined in the “mappings” section of an index
  - Define mappings at index creation, or
  - Update a mapping of an existing index using the Put API

You can define the mapping in the PUT command

```
PUT comments
{
  "mappings": {
    define mapping here
  }
}
```

# 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**, **geo\_shape** and **percolator**

“string” in older versions  
of Elasticsearch

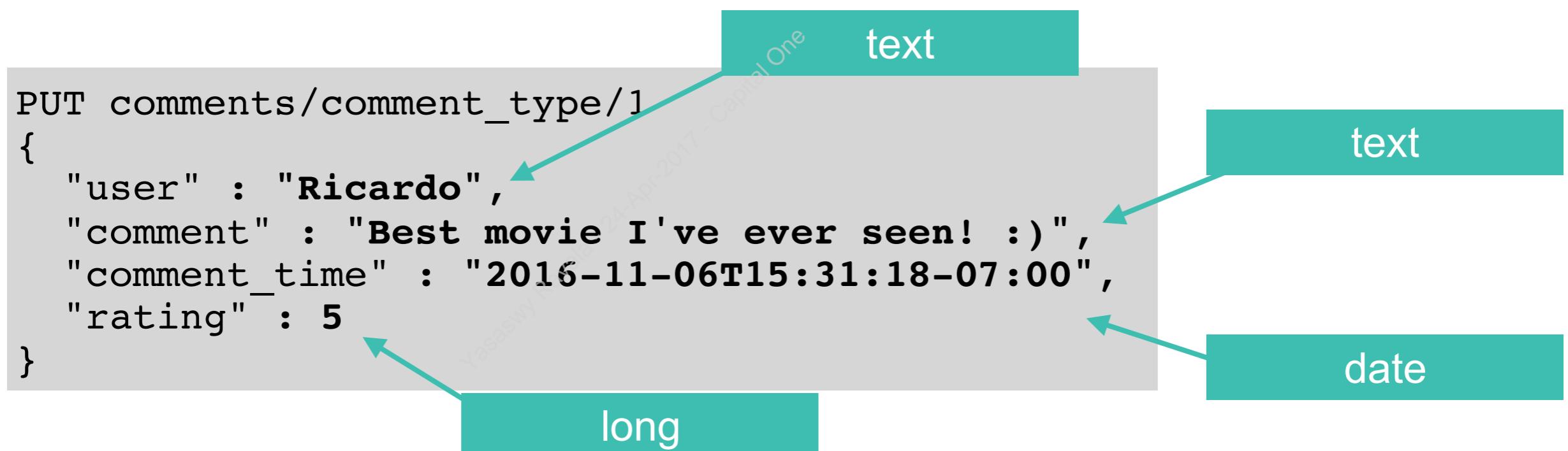
This list is not every  
data type available

# Why have we not needed to define any mappings yet?

Yasaswy Raval - 24-Apr-2017 - Capital One

# Dynamic Mappings

- You are not required to manually define mappings
- When you index a document into a type, Elasticsearch ***dynamically updates the mapping*** based on the document



# Elasticsearch “guesses” data types:

Sometimes it conveniently  
guesses what you want...

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

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

# Elasticsearch “guesses” data types:

Dates in the default format are also recognized

```
PUT comments/comment_type/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/comment_type/4
{
  "average_length" : "24.8"
```

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

# Defining Explicit Mappings

Yasaswy Raval - 24-Apr-2011 - Capital One

# Defining Explicit Mappings

- If you need to define an explicit mapping, we typically follow these steps:
  1. Index a sample document that contains the fields you want defined in the mapping (using a temporary index)
  2. Get the dynamic mapping that was created automatically by Elasticsearch in Step 1
  3. Edit the mapping definition
  4. Create your index, using your explicit mappings

Yasaswy Raval - 24-Apr-2017 - Capital One



# 1. Index a Sample Document

- Start by indexing a document into a temporary index...

...but use a **type** name that you want to keep

```
PUT temp_comments/comment_type/1
{
  "user" : "Ricardo",
  "comment" : "Best movie I've ever seen! :)",
  "comment_time" : "2016-11-06T15:31:18-07:00",
  "rating" : 5
}
```

## 2. Get the Dynamic Mapping

- Get the mapping, then copy-and-paste it into the Console:

```
GET temp_comments/_mapping
{
  "temp_comments": {
    "mappings": {
      "comment_type": {
        "properties": {
          "comment": {
            "type": "text",
            "fields": {
              "keyword": {
                "type": "keyword",
                "ignore_above": 256
              }
            }
          },
          "comment_time": {
            "type": "date"
          },
          "rating": {
            "type": "long"
          }
        }
      }
    }
  }
}
```



### 3. Edit the Mapping

- In our example, we will make a couple of changes:
  - By changing “user” to a “keyword”, it will only match exact searches

```
"rating": {  
    "type": "long"  
},  
"user": {  
    "type": "text",  
    "fields": {  
        "keyword": {  
            "type": "keyword",  
            "ignore_above": 256  
        }  
    }  
}
```

Let's change “rating” to a “byte”...

```
"rating": {  
    "type": "byte"  
},  
"user": {  
    "type": "keyword"  
}
```

and change “user” to a “keyword”

# 4. Define a New Index with the Mapping

“comments” is a new index with our explicit mapping

```
PUT comments
{
  "mappings": {
    "comment_type": {
      "properties": {
        "comment": {
          "type": "text",
          "fields": {
            "keyword": {
              "type": "keyword",
              "ignore_above": 256
            }
          }
        },
        "comment_time": {
          "type": "date"
        },
        "rating": {
          "type": "byte"
        },
        "user": {
          "type": "keyword"
        }
      }
    }
  }
}
```

# Add a document to the new index...

- Test the new mapping by adding a document
  - Notice since “user” is a “keyword” now, only exact searches return a hit:

```
GET comments/_search
{
  "query": {
    "match": {
      "user": "ricardo"
    }
  }
}
```

does not match “Ricardo”

```
GET comments/_search
{
  "query": {
    "match": {
      "user": "Ricardo"
    }
  }
}
```

a match!

# Configuring Specific Field Parameters

- Each datatype has parameters for customizing behaviors
  - For example, you can configure an “analyzer” for a “text” field:

```
PUT comments
{
  "mappings": {
    "comment_type": {
      "dynamic" : "strict",
      "properties": {
        "comment": {
          "type": "text",
          "analyzer": "english",
          "fields": {
            "comment_keyword": {
              "type": "keyword",
              "ignore_above": 256
            }
          }
        }
      }
    }
  }
}
```

The “comment” field will use the “english” analyzer



# Can you change a mapping?

Yasaswy Raval - 24-Apr-2012 Capital One

# Can you change a mapping?

- **No** - not without reindexing your documents
  - You *can* add fields and modify a few properties, but...
  - ...all other schema changes require a reindexing
- **Why not?**
  - If you switch a field's data type, all existing documents with that field already indexed would become unsearchable on that field
- **Why can I add fields without reindexing?**
  - Adding a field has no effect on the existing indexed fields or documents
  - The index can freely grow, but indexed fields can not change data types



# 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”	✗		

```
PUT comments
{
  "mappings": {
    "comment_type": {
      "dynamic" : "false",
      "properties": {
        ...
      }
    }
  }
}
```

Index any documents with unmapped fields, but ignore the fields and do not update the mapping

# Adding Fields to a Mapping

- There are different ways to add a field:
  - If the “**dynamic**” setting is “**true**”, you can add a field by simply indexing a document that contains the new field
  - Alternately, you can use the **PUT API**:

```
PUT comments/_mappings/comment_type
{
  "properties": {
    "comment_location" : {
      "type" : "geo_point"
    }
  }
}
```

We want to modify the mapping  
for “**comment\_type**”

Add a new field named  
“**comment\_location**”

# Dive Deeper into Mappings

Yasaswy Raval - 24-Apr-2017 - Capital One

# Date Formats

- Use the **format** setting to specify the format of your date fields:
  - **format** uses either a date string like “MM/dd/YYYY”, or
  - one of the many built-in date formats
- The default **format** value is  
**strict\_date\_optional\_time||epoch\_millis**

```
PUT comments/comment_type/5
{
  "comment_time" : "2016-11-06T15:31:18-07:00"
}
```



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

# Specifying a Date Format

```
{  
  "mappings": {  
    "comment_type": {  
      "properties": {  
        "last_viewed": {  
          "type": "date",  
          "format": "dd/MM/YYYY"  
        },  
        "comment_time": {  
          "type": "date",  
          "format": "basic_date||epoch_millis"  
        }  
      }  
    }  
  }  
}
```

Use || for multiple formats

View the documentation for a complete list of built-in date formats

# Specifying a Date Format

- Dates must now match the **format**:

```
PUT comments/comment_type/4
{
  "last_viewed" : "17/12/2016",
  "comment_time" : 1481071407407
}
```

```
{
  "_index": "comments",
  "_type": "comment_type",
  "_id": "4",
  "_score": 1,
  "_source": {
    "last_viewed": "17/12/2016",
    "comment_time": 1481071407407
  }
}
```

```
PUT comments/comment_type/5
{
  "last_viewed" : "01/31/2017"
}
```

Set “**ignore\_malformed**” to “**true**” to ignore the error

```
{
  "error": {
    "root_cause": [
      {
        "type": "mapper_parsing_exception",
        "reason": "failed to parse [last_viewed]"
      }
    ],
    "type": "mapper_parsing_exception",
    "reason": "failed to parse [last_viewed]",
    "caused_by": {
      "type": "illegal_field_value_exception",
      "reason": "Cannot parse \\"01/31/2017\\": value 31
for monthOfYear must be in the range [1,12]"
    }
  },
  "status": 400
}
```

# Meta-Fields

- Each document has metadata fields (`_index`, `_type`, `_id`)
- Some meta-fields we have not seen yet include:
  - `_all`: a special catch-all field which concatenates the values of all of the other fields into one big string
  - `_meta`: for storing application-specific metadata

Yasaswy Raval - 24-Apr-2017 - Capital One

# Disabling \_all

- 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 comments
{
  "mappings": {
    "comment_type": {
      "_all": {
        "enabled": false
      },
      "properties": {
        "comment": {
          "type": "text",
        },
        ...
      }
    }
  }
}
```

disable `_all` for for  
`comment_type`

# 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 comments/_mapping/comment_type
{
  "_meta": {
    "comment_type_version": "2.1"
  }
}
```

Store any application-specific  
JSON here...

# Common Field Parameters

- “**index- “**null\_valuenull**”, which means skip the field**

```
"properties": {  
    "my_password" : {  
        "type": "keyword",  
        "index": false  
    },  
    "my_rating" : {  
        "type": "integer",  
        "null_value": 1  
    }  
}
```

“**my\_password**

The value of 1 will be indexed for “**my\_rating**

# Specifying Analyzers

Yasaswy Raval - 24-Apr-2017 - CapitalOne

# Specifying a Custom Analyzer

```
PUT comments
{
  "settings" : {
    "analysis" : {
      "filter" : {
        "my_synonyms" : {
          "type" : "synonym",
          "synonyms" : [
            "jump, hop", "quick => fast"
          ]
        }
      },
      "analyzer" : {
        "my_analyzer" : {
          "tokenizer" : "standard",
          "filter" : [ "lowercase", "snowball", "my_synonyms" ]
        }
      }
    }
  },
  "mappings" : {
    "comment_type" : {
      "properties": {
        "comment" : {
          "type": "text",
          "analyzer": "my_analyzer"
        }
      }
    }
  }
}
```

The “comment” field will  
use “my\_analyzer”

Optionally, you can also add  
“search\_analyzer” to a field to  
define a different analyzer for  
search time

# Search Strings and Analyzers

- If you configure a field to use an analyzer in your mappings, then search queries on that field will also use the same analyzer:

```
PUT comments/comment_type/1
{
  "comment" : "The quick rabbit can jump high"
}
```

```
GET comments/_search
{
  "query": {
    "match": {
      "comment": "hop"
    }
  }
}
```

You can add “**analyzer**” to control which analyzer will perform the analysis process on the text

```
"hits": {
  "total": 1,
  "max_score": 0.3938048,
  "hits": [
    {
      "_index": "comments",
      "_type": "comment_type",
      "_id": "1",
      "_score": 0.3938048,
      "_source": {
        "comment": "The quick rabbit can
jump high"
      }
    }
  ]
}
```

# Multi-Fields

- It is often useful to index the same field in different ways
  - For example, it allows your full text to be analyzed in multiple ways
- You can specify an “**analyzer**” property for each field

```
"comment": {  
    "type": "text",  
    "fields": {  
        "keyword": {  
            "type": "keyword",  
            "ignore_above": 256  
        },  
        "english_comment": {  
            "type": "text",  
            "analyzer": "english"  
        },  
        "whitespace_comment": {  
            "type": "text",  
            "analyzer": "whitespace"  
        }  
    }  
}
```

This “comment” field will be indexed four different ways

# Accessing a Multi-Field

- Use the dot operator in a query to refer to a multi-field:

```
PUT comments/comment_type/1
{
  "comment" : "Hello, world!"
}

GET comments/_search
{
  "query": {
    "match": {
      "comment.english_comment": "hello"
    }
  }
}
```

“Hello, world!” gets analyzed four times

Use the dot operator to choose which multi-field to query

# Index Templates

Yasaswy Raval - 24-Apr-2017 - Capital One

# Index Templates

- ***Index templates*** allow you to define mappings and settings that will automatically be applied to newly-created indices:
  - You can have multiple index templates
  - They get “merged” and applied to the created index
  - You can provide an “order” value to control the merging process
- **When a new index is created:**
  1. The default settings and mappings are applied
  2. Settings and mappings from the lowest “order” template are applied
  3. Settings and mappings from the higher “order” templates are applied, overriding previous settings along the way
  4. The setting and mappings from the PUT command of the index are applied last and override all previous templates



# Defining a Template

- Use the `_template` endpoint to add, view and delete templates

```
PUT _template/my_template_1
{
  "template" : "*",
  "order" : 1,
  "mappings" : {
    "my_type" : {
      "_all": {
        "enabled": false
      }
    }
  }
}
```

name of the template

Apply this template to any new index



# Let's define a second template:

```
PUT _template/my_template_2
{
  "template" : "my_*",
  "order" : 5,
  "mappings": {
    "my_type" : {
      "properties": {
        "version" : {
          "type" : "integer"
        }
      }
    }
  }
}
```

Apply this template to  
any new index that  
starts with “my\_”



# Now define a new index:

- Notice our index named “my\_test” matches both templates:

```
PUT my_test
```

```
GET my_test/_mappings
```

```
{  
  "my_test": {  
    "mappings": {  
      "my_type": {  
        "_all": {  
          "enabled": false  
        },  
        "properties": {  
          "version": {  
            "type": "integer"  
          }  
        }  
      }  
    }  
  }  
}
```

# Viewing and Deleting Templates

- Use a **GET** to view your defined templates:

```
GET _template
```

- You can **DELETE** a template also:

```
DELETE _template/my_template_2
```

Yasaswy Raval - 24-Apr-2017 - Capital One



# Mapping Best Practices

- Define your own mappings!
  - The dynamically-created mappings may have the wrong data types or settings appropriate for your documents
- Use index templates!
- Define a default template for “\*” that contains global settings for all your new indexes
- Templates are a great place to define your custom analyzers
- Disable “\_all” if you do not need it
- Try to avoid using multiple types in a single index



# Chapter Review

Yasaswy Raval - 24-Apr-2017 - Capital One

# 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
- A common way to define an explicit mapping is to modify a dynamic mapping using a well-designed sample document
- Define your mappings well! Only a few changes can be made, including the adding of fields.
- ***Index templates*** allow you to define mappings and settings that will automatically be applied to newly-created indices



# Quiz

1. What data type would “**value**” : 300 get mapped to dynamically?
2. If “**dynamic**” is set to “**strict**”, what happens when an unmapped field is indexed?
3. **True or False:** You can use mappings to specify text analyzers at the field level.
4. **True or False:** You can change a field’s data type from “**integer**” to “**long**” because those two types are compatible.
5. What is the purpose of the “**\_all**” field?



# Lab 4

## Mappings

Yasaswy Raval - 24-Apr-2017 - Capital One

# Chapter 5

# More Search Features

Yasaswy Raval - 24-Apr-2017 - Capital One

- 1 Introduction to Elasticsearch
- 2 The Search API
- 3 Text Analysis
- 4 Mappings
- 5 More Search Features
- 6 The Distributed Model
- 7 Working with Search Results
- 8 Aggregations
- 9 More Aggregations
- 10 Handling Relationships

# Topics covered:

- Filters
- Term Filters
- The match\_phrase\_prefix Query
- The multi\_match\_Query
- Fuzziness
- Highlighting
- Index Aliases
- The Completion Suggester

Yasaswy Ravula - 24-Apr-2017 - Capital One



# We have a new dataset...

Yasaswy Raval - 24-Apr-2017 | Capital One

# Dataset with nutritional information:

- It is nutritional information for 500 food items
  - Index name = “**nutrition**”
  - Document type = “**nutrition\_type**”

```
{  
    "total_fat": 6,  
    "calories_from_fat": 50,  
    "brand_name": "Tostitos",  
    "item_name": "Tortilla Chips, Thick & Hearty Rounds",  
    "calories": 140,  
    "servings_per_container": 32,  
    "saturated_fat": 1,  
    "serving_size_unit": "chips",  
    "ingredients": [  
        "Whole Yellow Corn",  
        " Vegetable Oil (Corn",  
        " Soybean",  
        " Canola and/or Sunflower Oil)",  
        " and Salt."  
    ],  
    "item_description": "Thick & Hearty Rounds",  
    "serving_size_qty": 10  
}
```

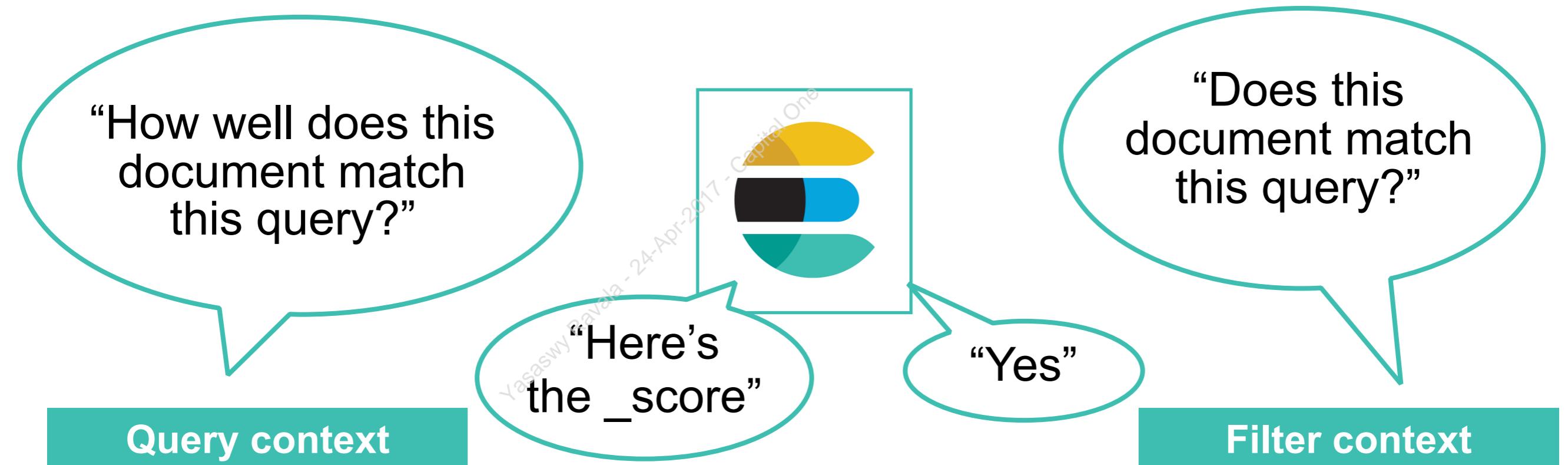


# Filters

Yasaswy Raval - 24-Apr-2017 - Capital One

# 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 filter Clause

- The **filter** clause is an optional clause in a **bool** query:

```
GET my_index/_search
{
  "query": {
    "bool": {
      "must": [
        {"match": {"comment": "star"}},
        "filter": {
          "match": {"comment": "favorite"}}
      ]
    }
  }
}
```

The **\_score** will be the result of this query only

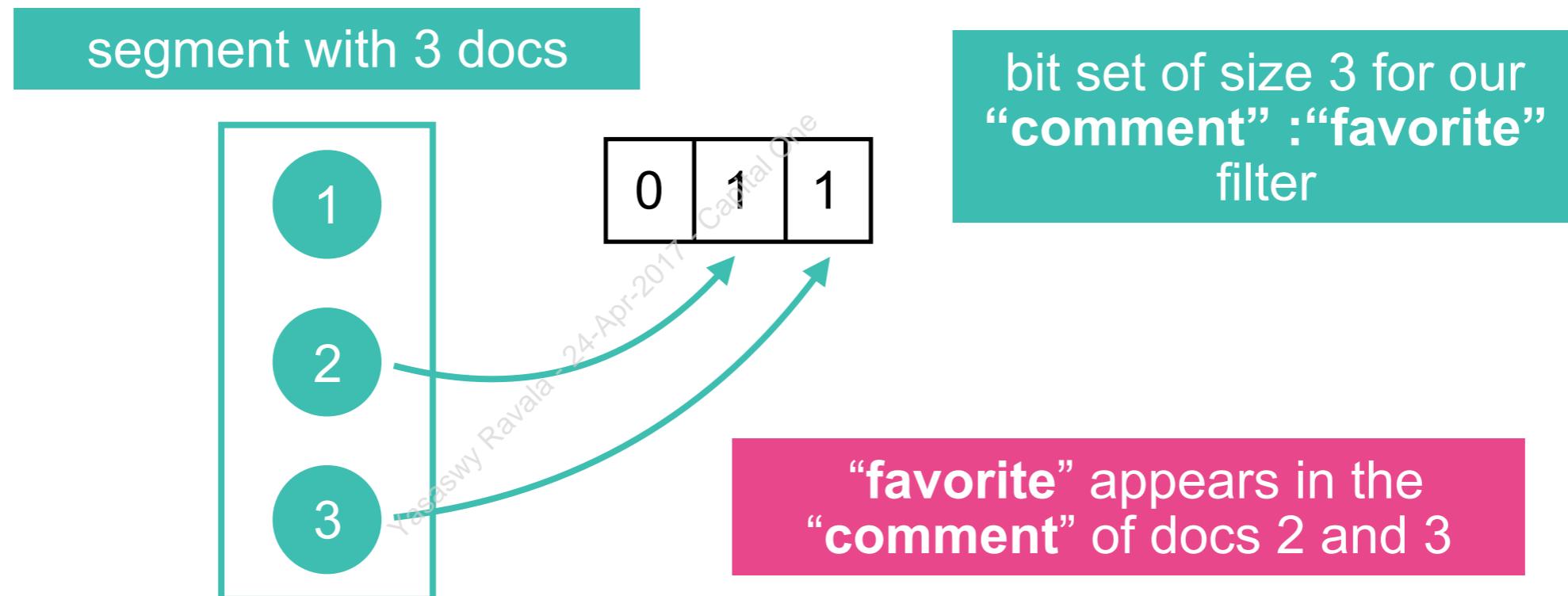
Hits must have “favorite” in the “comment” field

“My favorite movie star in Hollywood is Harrison Ford.”

“My favorite movie is Star Wars!”

# Performance Benefit of Filters

- A query in a filter context can be ***cached*** by Elasticsearch to improve performance
  - referred to as the ***node query cache*** (or ***query cache*** for short)
  - uses a ***bit set*** at the segment level



# Term Filters

Yasaswy Raval - 24-Apr-2017 - Capital One

# Term-based Queries

- Text fits into two groups:
  - ***Exact values***: includes numeric values, dates, and certain strings
  - ***Full text***: unstructured text data that we want to search
- When you submit a string in a **match** query, the mapping is checked to see if the string should be analyzed
  - This makes sense: your query string will match your indexed text
- ***Term-based queries*** are queries that do **not** go through an analysis phase
  - for searching numerics, dates, and string values that are exact
  - typically used in filter contexts



# The term Query

- In a *term* query, the search terms are not analyzed
  - A hit has to match the search term exactly

```
GET nutrition/_search
{
  "query": {
    "term": {
      "brand_name.keyword" : "Market Pantry"
    }
  }
}
```

*“Find all products from Market Pantry.”*

The nutrition index has 4 hits

```
GET nutrition/_search
{
  "query": {
    "term": {
      "brand_name.keyword" : "Market"
    }
  }
}
```

*“Find all products from Market.”*

This search returns 0 hits

# The term Query

- You can use **term** at the **query** level (see previous slide)
  - But, **term** queries are typically used in filter clauses, as we often filter on exact matches:

74 hits without the filter...

GET nutrition/\_search

```
{  
  "query": {  
    "bool": {  
      "must": [  
        {"match": {  
          "ingredients": "syrup"  
        }}  
      ],  
      "filter": {  
        "term": {  
          "brand_name.keyword": "Market Basket"  
        }  
      }  
    }  
  }  
}
```

*"I am looking for  
products from Market  
Basket with syrup in the  
ingredients"*

...only 1 hit with the filter

# The terms Query

- Use **terms** to find documents that match *any* of the given terms:

```
GET nutrition/_search
{
  "query": {
    "bool": {
      "must": [
        {"match": {
          "ingredients": "syrup"
        }}
      ],
      "filter": {
        "terms": {
          "brand_name.keyword": [
            "Market Basket",
            "Nabisco",
            "Barilla"
          ]
        }
      }
    }
  }
}
```

*"I am looking for syrup from Market Basket, Nabisco, or Barilla."*

The **terms** query allows you to search multiple terms



# 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 nutrition/_search
{
  "query": {
    "bool": {
      "must": [
        {"match": {"item_name": "cookie"}},
        "filter": {
          "wildcard": {
            "brand_name.keyword": {
              "value": "Ste*"
            }
          }
        }
      ]
    }
  }
}
```

*I am looking for cookies from a brand name that starts with “Ste”.*

“Premium Dark Chocolate, with Mint **Cookie** Crunch” from “**Stewart’s**” brand.

# The exists Query

- The **exists** query returns documents that have at least one non-null value in the specified field
  - It is a simple way to filter out documents with null fields

```
GET nutrition/_search
{
  "query" : {
    "bool": {
      "must": [
        {"match": {
          "item_name": "cookie"
        }}
      ],
      "filter": {
        "exists": {
          "field": "ingredients"
        }
      }
    }
  }
}
```

*"I am looking for  
cookies that have their  
ingredients listed."*



# The match\_phrase\_prefix Query

Yasaswy Raval - 24-Apr-2017  
Capital One

# The match\_phrase\_prefix Query

- Same as `match_phrase`, except the last term in the search string is a prefix:

```
GET nutrition/_search
{
  "query" : {
    "match_phrase_prefix": {
      "item_name": {
        "query": "chocolate ch",
        "max_expansions" : 20
      }
    }
  }
}
```

*“Find  
‘chocolate’ followed  
by any word that starts  
with ‘ch’.”*

"Gluten Free, Chocolate Chunk Cookies"

"Ice Cream, Mint Chocolate Chip"

"Mint Chocolate Chip Ice Cream"

"Pancakes, Chocolate Chip"



# It almost feels like “auto-complete”...

- `match_phrase_prefix` definitely has the appearance of an auto-complete tool

```
GET nutrition/_search
{
  "query" : {
    "match_phrase_prefix": {
      "item_name": {
        "query": "chocolate chi",
        "max_expansions" : 20
      }
    }
  }
}
```

We add an “i” to the prefix:

"Gluten Free, Chocolate Chunk Cookies"

"Ice Cream, Mint Chocolate Chip"

"Mint Chocolate Chip Ice Cream"

"Pancakes, Chocolate Chip"

# ...but it is not a perfect solution

- **match\_phrase\_prefix** is a lot like auto-complete, but it is important to understand how it works behind the scenes:
  - **max\_expansions**: 20 means that only the first 20 terms which start with “ch” (in alphabetical order) are checked
  - none of the expansions are validated to see if they even co-occur with “chocolate”

```
GET nutrition/_search
{
  "query" : {
    "match_phrase_prefix": {
      "item_name": {
        "query": "chocolate ch",
        "max_expansions" : 20
      }
    }
  }
}
```

Short prefixes may return zero results,  
even when there are valid matches



# The multi-match Query

Yasaswy Raval - 24-Apr-2017 - Capital One

# The multi\_match Query

- **multi\_match** is similar to **match** except you can query multiple fields:
  - The syntax looks like:

```
GET nutrition/_search
{
  "query" : {
    "multi_match": {
      "query": "",
      "fields": []
    }
  }
}
```

Yasaswy Raval - 24 Mar 2017 - Capital One

Enter the query string once...

...and an array of fields to search



# Example of multi\_match

- By default, the `_score` from the best field is used
  - But you can control various aspects of how the `_score` is calculated

```
GET nutrition/_search
{
  "query" : {
    "multi_match": {
      "query": "olive oil",
      "fields": [ "item_name", "ingredients" ]
    }
  }
}
```

“Search for ‘**olive oil**’ in either the item name or in the ingredients”.

“Extra Virgin Olive Oil”

“Trout, Golden Smoked Fillets 3.75 Oz”

“House Salad Dressing”

# Per-field boosting in multi-match

- Individual fields can be *boosted*
  - Use a caret (^) followed by a boost multiplier

```
GET nutrition/_search
{
  "query" : {
    "multi_match": {
      "query": "olive oil",
      "fields": ["item_name", "ingredients^2"]
    }
  }
}
```

The “**ingredients**” score is 2 times more important than “**item\_name**”

"Trout, Golden Smoked Fillets 3.75 Oz"

"House Salad Dressing"

"Olive Oil, Extra Virgin, Cold Pressed,  
Bread & Salads"



# Wildcards in Field Names

- **multi\_match** allows for wildcards in fields names:

```
GET nutrition/_search
{
  "query" : {
    "multi_match": {
      "query": "rice",
      "fields": [ "*name", "ingredients"]
    }
  }
}
```

Search any field that ends in “name”



# Types of multi\_match Queries

- **best\_fields** (default): score is from the best field
- **most\_fields**: score is a combination from all matching fields
- **cross\_fields**: searches for each term in the query string in any of the fields in the document

```
GET nutrition/_search
{
  "query" : {
    "multi_match": {
      "query": "rice",
      "fields": [ "*"name", "ingredients" ],
      "type": "most_fields"
    }
  }
}
```

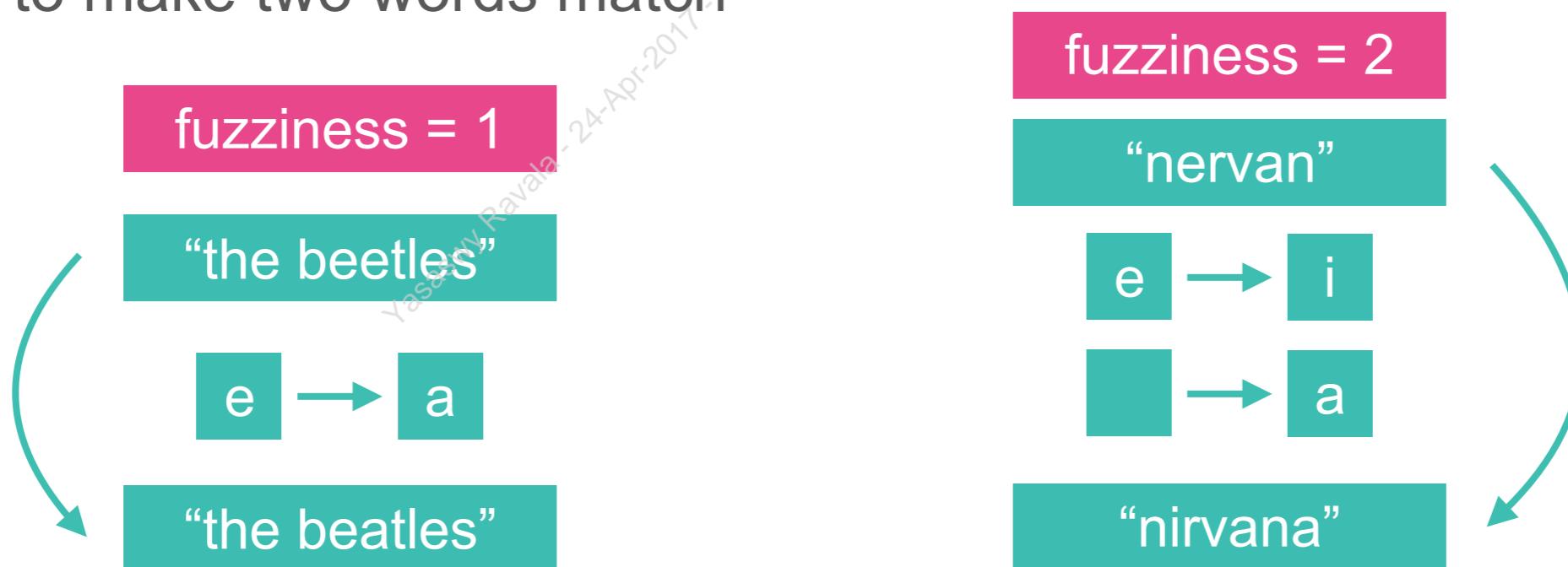
Change the scoring to use “**most\_fields**”

# Fuzziness

Yasaswy Raval - 24-Apr-2017 - Capital One

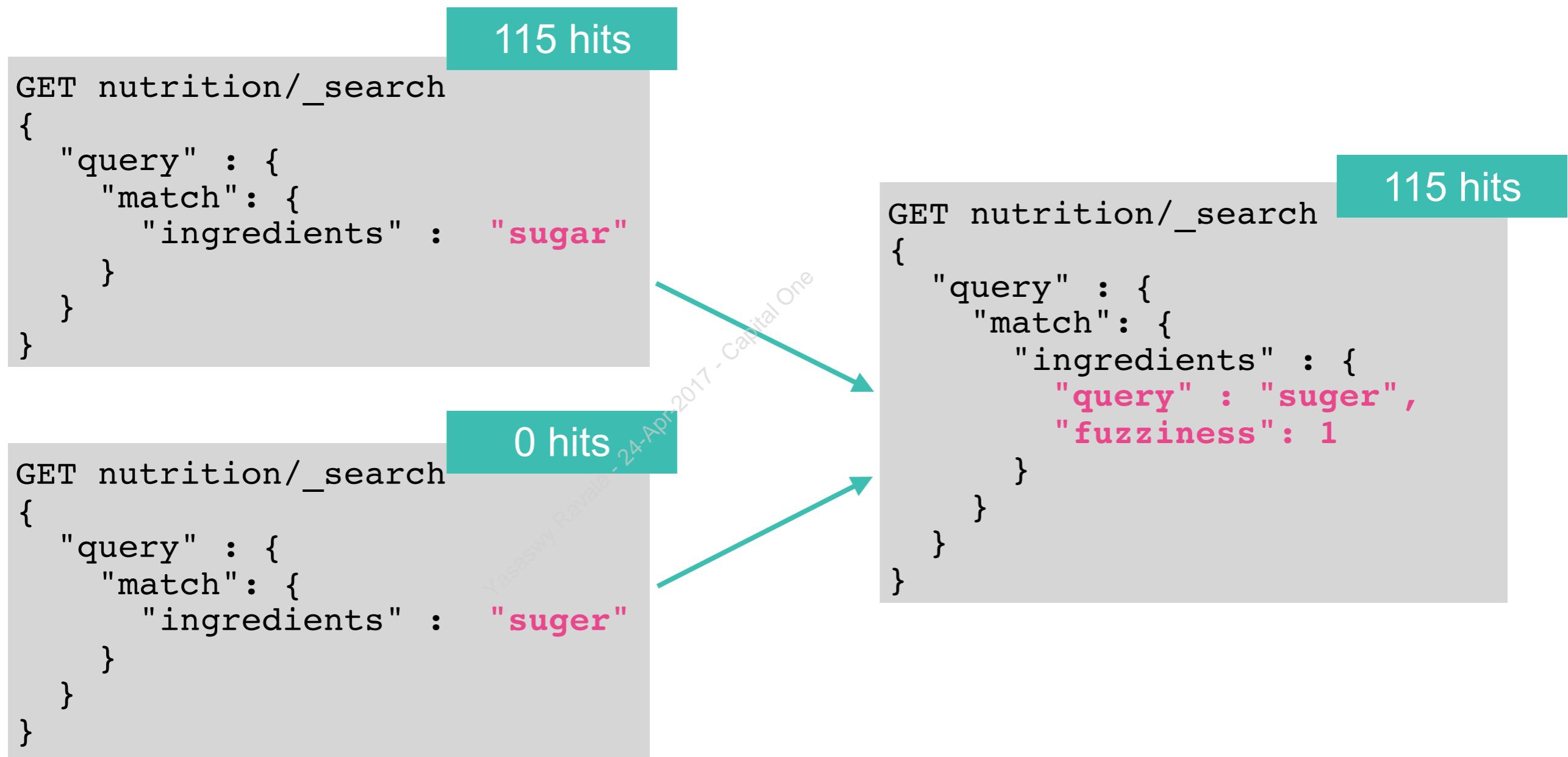
# What is Fuzziness?

- **Fuzzy** matching allows for query-time matching of misspelled words
  - It is a certainty that search users will misspell words
- **Fuzzy matching** treats two words that are “**fuzzily**” similar as if they were the same word
  - Fuzziness is something that can be assigned a value:
    - It refers to the number of character modifications, known as 'edits', to make two words match



# Fuzzy match Query

- The **match** query has a “**fuzziness**” property
  - Can be set to 0, 1 or 2; or can be set to **AUTO**



# The AUTO Setting

- **fuzziness** can be set to **AUTO**
  - generates an edit distance based on the length of the term
  - **AUTO** is the preferred way to use fuzziness
- Suppose you set the value to 2:
  - a word like "to" also matches "as", "by", "if", "the", and so on

```
GET my_index/_search
{
  "query" : {
    "match": {
      "ingredients" : {
        "query" : "a to z plumbing",
        "fuzziness": "AUTO"
      }
    }
  }
}
```

Suppose we want  
“A to Z Plumbing”



# Multiple terms in the query

- If the query has multiple terms, the fuzziness value is applied to each term
  - For example “citric” and “sitrik” are 2 edits apart
  - “acid” and “acet” are also 2 edits apart

```
GET nutrition/_search
{
  "query": {
    "match": {
      "ingredients": "citric acid"
    }
  }
}
```

120 hits

```
GET nutrition/_search
{
  "query": {
    "match": {
      "ingredients": {
        "query": "sitrik acet",
        "fuzziness": 2
      }
    }
  }
}
```

122 hits

# Highlighting

Yasaswy Raval - 24-Apr-2017 - Capital One

# Highlighting

- A common use case for search results is to *highlight* the matched terms
  - Elasticsearch makes it easy to do this

```
GET my_index/_search
{
  "query" : {
    ...
  },
  "highlight": {
    "fields": {
      ...
    }
  }
}
```

Add a “highlight” clause that lists the fields you want highlighted



# Example of Highlighting

```
GET nutrition/_search
{
  "query": {
    "match": {
      "ingredients": "oil"
    }
  },
  "highlight": {
    "fields": {
      "ingredients": {}
    }
  }
}
```

Highlight “oil” if it appears in  
“ingredients”

"highlight": {  
 "ingredients": [  
 "Vegetable <em>Oil</em> (Cottonseed <em>Oil</em> and/or Canola  
<em>Oil</em>)"  
 ]  
}

The “hit” will contain a  
“highlight” section

# Highlighting multiple fields

- Note: you have to query a field if you want it highlighted in the results (or set `require_field_match: false`)

```
GET nutrition/_search
{
  "size": 200,
  "query" : {
    "match": {
      "ingredients": "oil"
    }
  },
  "highlight": {
    "fields": {
      "ingredients" : {},
      "item_name" : {}
    }
  }
}
```

```
"highlight": {
  "ingredients": [...]
}
```

**"item\_name"** not here

```
GET nutrition/_search
{
  "query" : {
    "multi_match": {
      "query": "oil",
      "fields": [ "ingredients", "item_name" ]
    }
  },
  "highlight": {
    "fields": {
      "ingredients" : {},
      "item_name" : {}
    }
  }
}

"highlight": {
  "ingredients": [...],
  "item_name": [...]
}
```

# Changing the Tags

- Use “`pre_tags`” and “`post_tags`” to change the highlighting:

```
"highlight": {  
    "pre_tags" : ["<elasticsearch-hit>"],  
    "post_tags" : ["</elasticsearch-hit>"],  
    "fields": {  
        "ingredients" : {},  
        "item_name" : {}  
    }  
}
```

```
"highlight": {  
    "item_name": [  
        "Coconut <elasticsearch-hit>Oil</elasticsearch-hit>"  
    ]  
}
```

Results highlighted with your  
custom tags

# Index Aliases

Yasaswy Raval - 24-Apr-2017 - Capital One

# Index Aliases

- The **Index Aliases API** allows you to define an **alias** for an index
- The syntax looks like:

```
POST _aliases
{
  "actions": [
    {
      "add": {
        "index": "INDEX_NAME",
        "alias": "ALIAS_NAME"
      }
    }
  ]
}
```



# Example of Index Alias

```
POST _aliases
{
  "actions": [
    {
      "add": {
        "index": "my_index",
        "alias": "my_alias"
      }
    }
  ]
}

GET my_alias/_search
{
  "query": {
    "match_all": {}
  }
}
```

Returns hits from  
“my\_index”

# Suggesters

Yasaswy Raval - 24-Apr-2017 - Capital One

# What is a Suggester?

- A **suggester** is a feature of Elasticsearch where similar-looking terms are found based on a given query string

the loovre EN

**Louvre**  
art museum in Paris, France

**The Louprevil Press**

“Did you mean...”

Yasaswy Raval - 24-Apr-2011 Gapitaone

the bea EN

**The Beatles**  
English rock band

**The Beach Boys**  
American rock band

**The Beautiful South**  
English pop group formed at the end of the 1980s

**The Beano**  
magazine

Auto-complete



# Types of Suggesters

- Elasticsearch has four types of suggesters:
  - **Term suggester**: for a “*did you mean*” feature on terms
  - **Phrase suggester**: for a “*did you mean*” feature on phrases
  - **Completion suggester**: for implementing an ***auto-complete*** feature
  - **Context suggester**: adds context to an auto-complete feature
- We will discuss the completion suggester to demonstrate how suggesters are used



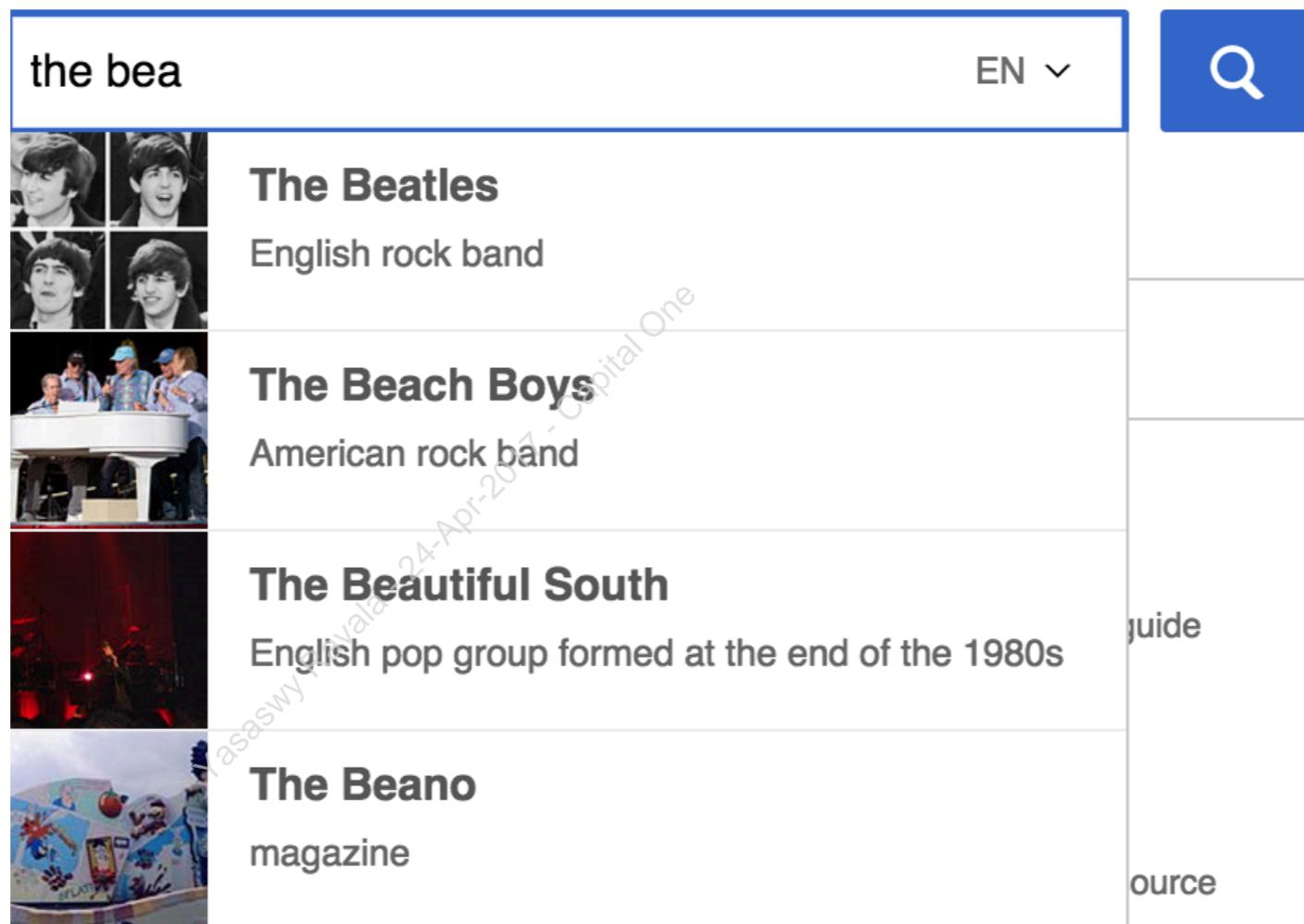
# Completion Suggester

Yasaswy Raval - 24-Apr-2017 - Capital One

<#>

# Auto-complete Feature

- Nowadays, most users expect (and rely on) the auto-complete feature of search tools
  - Elasticsearch has a built-in completion suggester



# The Completion Suggester

- The ***completion suggester*** provides auto-complete (search-as-you-type) functionality
- Each indexed document stores its own auto-complete suggestions
  - Requires a custom mapping
- Let's go through the steps:
  1. Define a custom mapping with a “**completion**” type
  2. Index some documents, along with their suggestions
  3. Run a “**suggest**” search to ask for suggestions



# The completion Data Type

- “completion” is a data type
  - The auto-complete values of a document are indexed for faster lookups

```
PUT nutrition_2
{
  "mappings": {
    "nutrition_type": {
      "properties": {
        "brand_name": {
          "type": "text",
          "fields": {
            "keyword": {
              "type": "keyword",
              "ignore_above": 256
            },
            "brand_name_suggest": {
              "type": "completion"
            }
          }
        },
        ...
      }
    }
  }
}
```

Add a field that will contain  
auto-complete suggestions for  
the **brand\_name** field



# The Suggest Feature

- Ask for suggestions using the suggest feature
  - our particular example is asking for completion suggestions

```
GET nutrition_2/_search
{
  "suggest" : {
    "my_brandname_suggest" : {
      "prefix" : "n",
      "completion" : {
        "field" : "brand_name.brand_name_suggest",
        "size" : 15
      }
    }
  }
}
```

Number of suggestions  
to retrieve

*Do you have any  
auto-complete suggestions  
for “n”?*

“Nabisco”
“Nalley”
“Namaste Foods”
“Napoleon”
“Nestle Dolcetto”
“New England Country Soup”
“New York Style”
“Nice!”
...and so on



# The Suggest Feature

- Send the same request again, but with a different “prefix”

```
GET nutrition_2/_search
{
  "suggest" : {
    "my_brandname_suggest" : {
      "prefix" : "na",
      "completion" : {
        "field" : "brand_name.brand_name_suggest",
        "size" : 15
      }
    }
  }
}
```

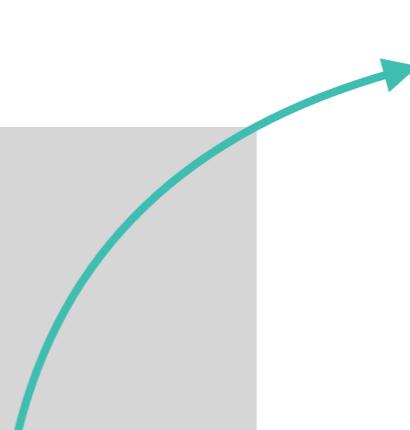


“Nabisco”
“Nalley”
“Namaste Foods”
“Napoleon”
“Nestle Dolcetto”
“New England Country Soup”
“New York Style”
“Nice!”
...and so on

# The Suggest Feature

- In this example, “nab” only returns one brand name:

```
GET nutrition_2/_search
{
  "suggest" : {
    "my_brandname_suggest" : {
      "prefix" : "nab",
      "completion" : {
        "field" : "brand_name.brand_name_suggest",
        "size" : 15
      }
    }
  }
}
```



“Nabisco”
“Nalley”
“Namaste Foods”
“Napoleon”
“Nestle Dolcetto”
“New England Country Soup”
“New York Style”
“Nice!”
...and so on



# Adding Fuzziness

- You can add a “fuzziness” factor to your suggester query

```
GET nutrition_2/_search
{
  "suggest" : {
    "my_brandname_suggest" : {
      "prefix" : "nab",
      "completion" : {
        "field" : "brand_name.brand_name_suggest",
        "size" : 15,
        "fuzzy" : {
          "fuzziness" : 1
        }
      }
    }
  }
}
```



# Chapter Review

Yasaswy Raval - 24-Apr-2017 - Capital One

# Summary

- Filters are cached and reused by Elasticsearch, for even greater performance
- **multi\_match** is similar to the **match** query and allows you to query multiple fields
- **Fuzzy** matching allows for query-time matching of similar words
- **Highlighting** allows you to highlight search results that match the query
- The **Index Aliases API** allows you to define an **alias** for an index
- The **Completion Suggester** provides auto-complete (search-as-you-type) functionality



# Quiz

1. What is the fuzziness value between a search for “yellow submarine” and “yellow submarene”?
2. **True or False:** Implementing a completion suggester requires a custom mapping for the index.
3. What is the easiest way to search for the same terms across multiple fields?
4. Suppose we are searching StackOverflow. What is the difference using between **best\_fields** and **most\_fields**?

```
GET stackoverflow/_search
{
  "query": {
    "multi_match": {
      "query": "java",
      "fields": ["body", "subject"]
    }
  }
}
```



# Lab 5

## More Search Features

Yasaswy Raval - 24-Apr-2017 - Capital One

# Chapter 6

# The Distributed Model

Yasaswy Raval - 24-Apr-2017 - Capital One

- 1 Introduction to Elasticsearch
- 2 The Search API
- 3 Text Analysis
- 4 Mappings
- 5 More Search Features
- 6 The Distributed Model
- 7 Working with Search Results
- 8 Aggregations
- 9 More Aggregations
- 10 Handling Relationships

# Topics covered:

- Starting a Node
- Creating an Index
- Starting a Second Node
- Shards: Distribution of an Index
- Distributing Documents
- Replication
- Split Brain

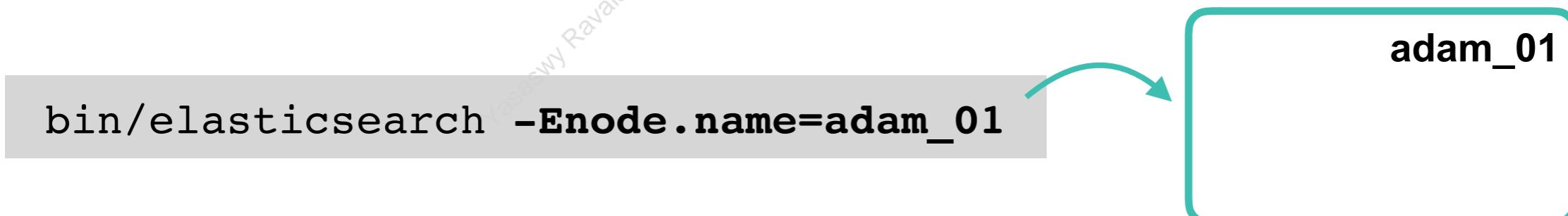
Yasaswy Raval - 24-Apr-2017 - Capital One

# Starting a Node

Yasaswy Raval - 24-Apr-2017 - CapitalOne

# What is a Node?

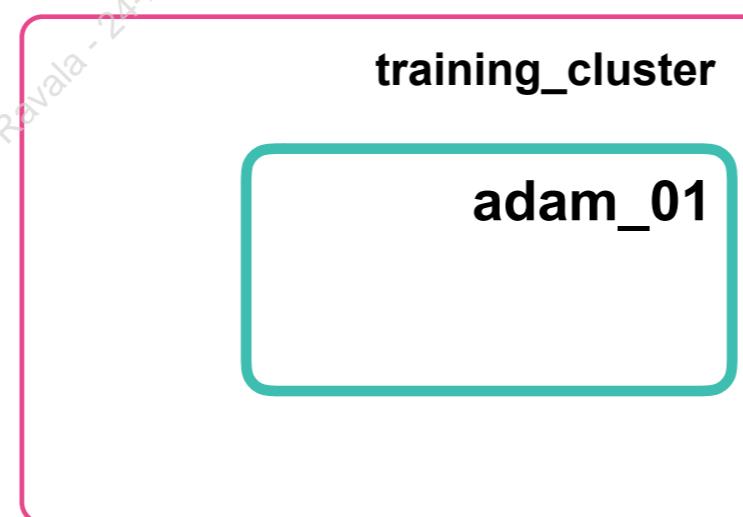
- A *node* is an instance of Elasticsearch
  - It is a java process that runs in a JVM
- Every node has a name
  - you can set **node.name** in the **elasticsearch.yml** file
  - or on the command line
- Each node has a UID assigned at startup
  - and persisted in the data folder for future restarts



# Cluster

- Everything in Elasticsearch happens in the context of a **cluster**
  - Every cluster has a name (e.g.: ***training\_cluster***)
  - Default cluster name is “elasticsearch”
  - Every node belongs to a cluster
- You can set the **cluster.name** that a node belongs to in **elasticsearch.yml** or on the command line

```
bin/elasticsearch -Enode.name=adam_01 -Ecluster.name=training_cluster
```



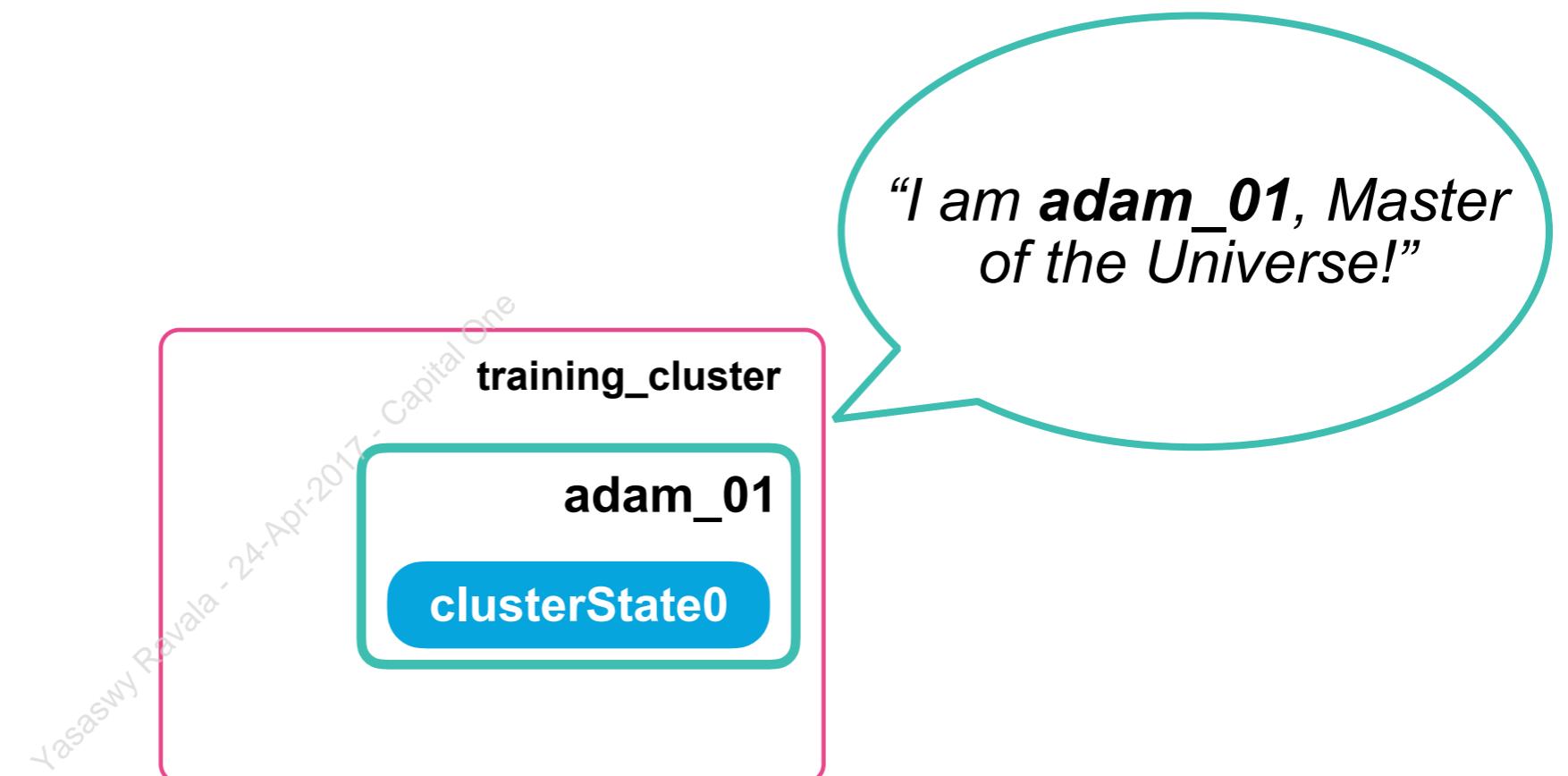
# Cluster State

- Every cluster has information referred to as the *cluster state*
  - includes the nodes in the cluster, indices, mappings, settings, and so on
- Cluster state is stored on each node



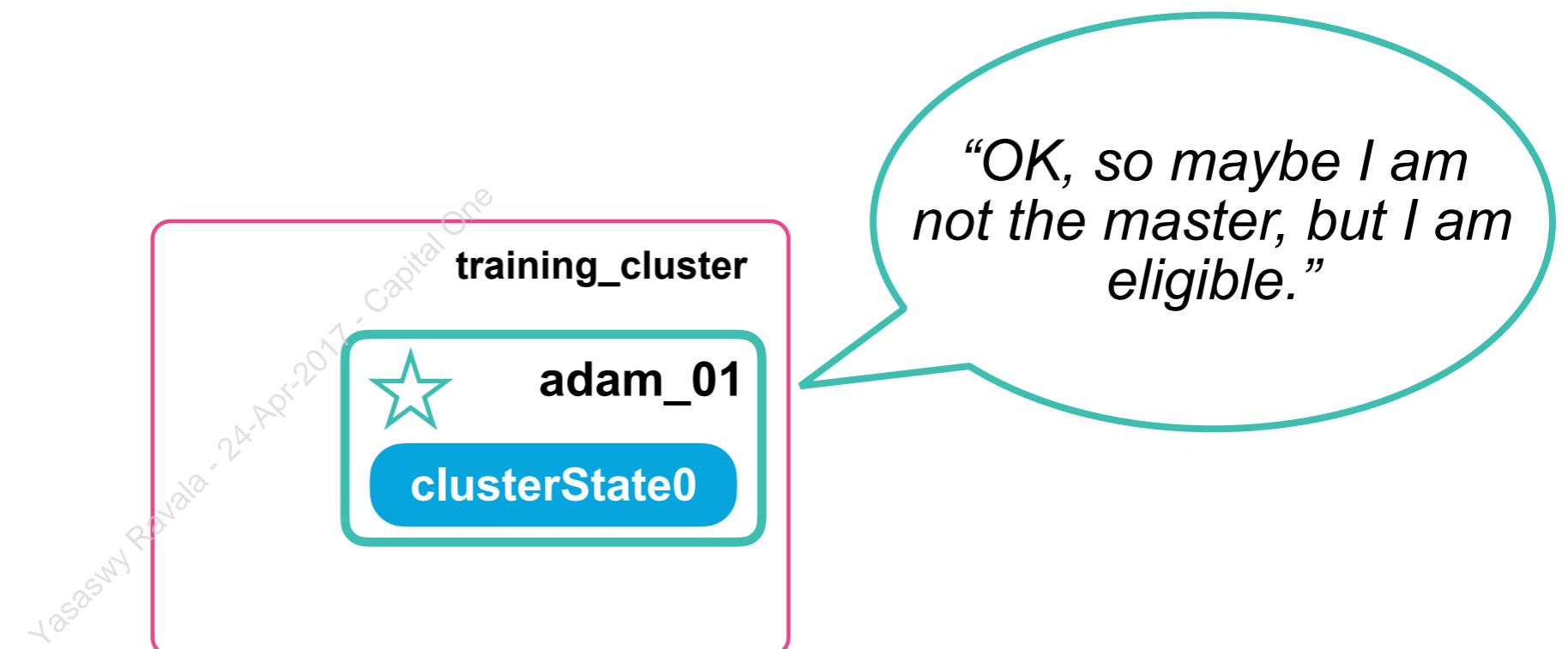
# But not every node can edit the cluster state

- In a distributed model, if everyone could edit the cluster state there could be inconsistencies and data loss
- We need a single source of truth...



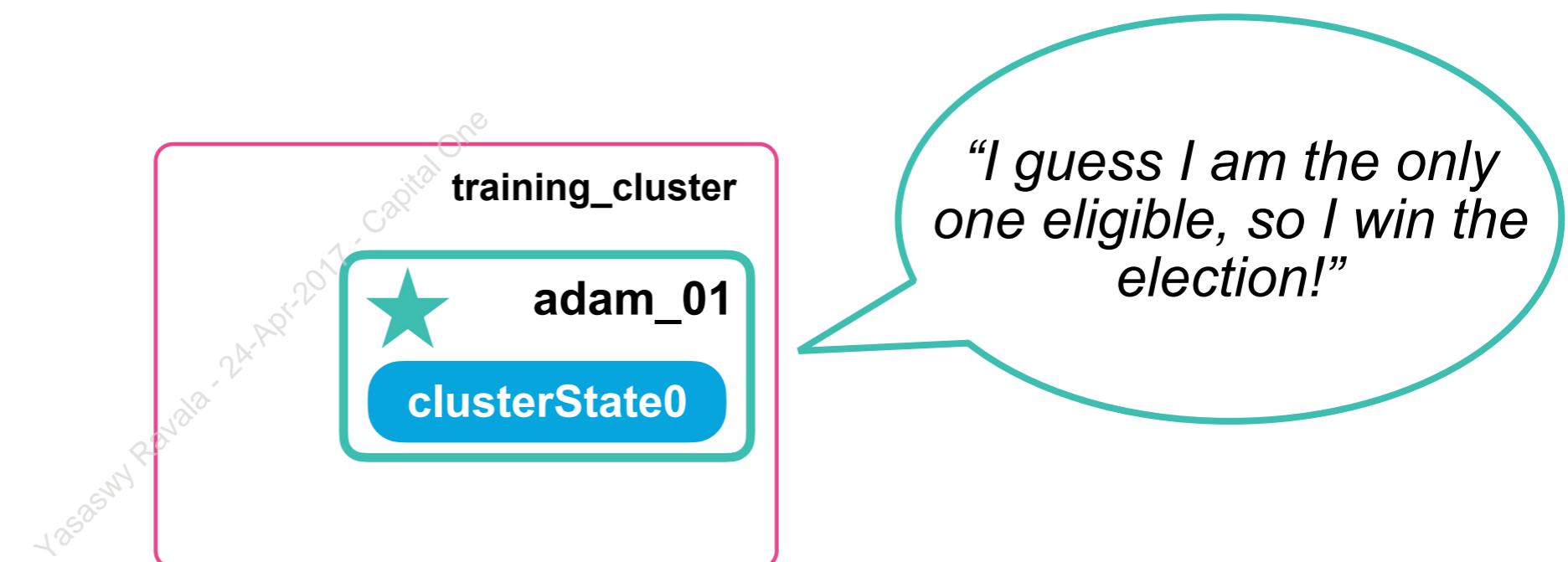
# Master-eligible Nodes

- Nodes can be eligible to become the master
- When you start **adam\_01** it is a master-eligible node by default
  - which can be disabled by setting **node.master: false**



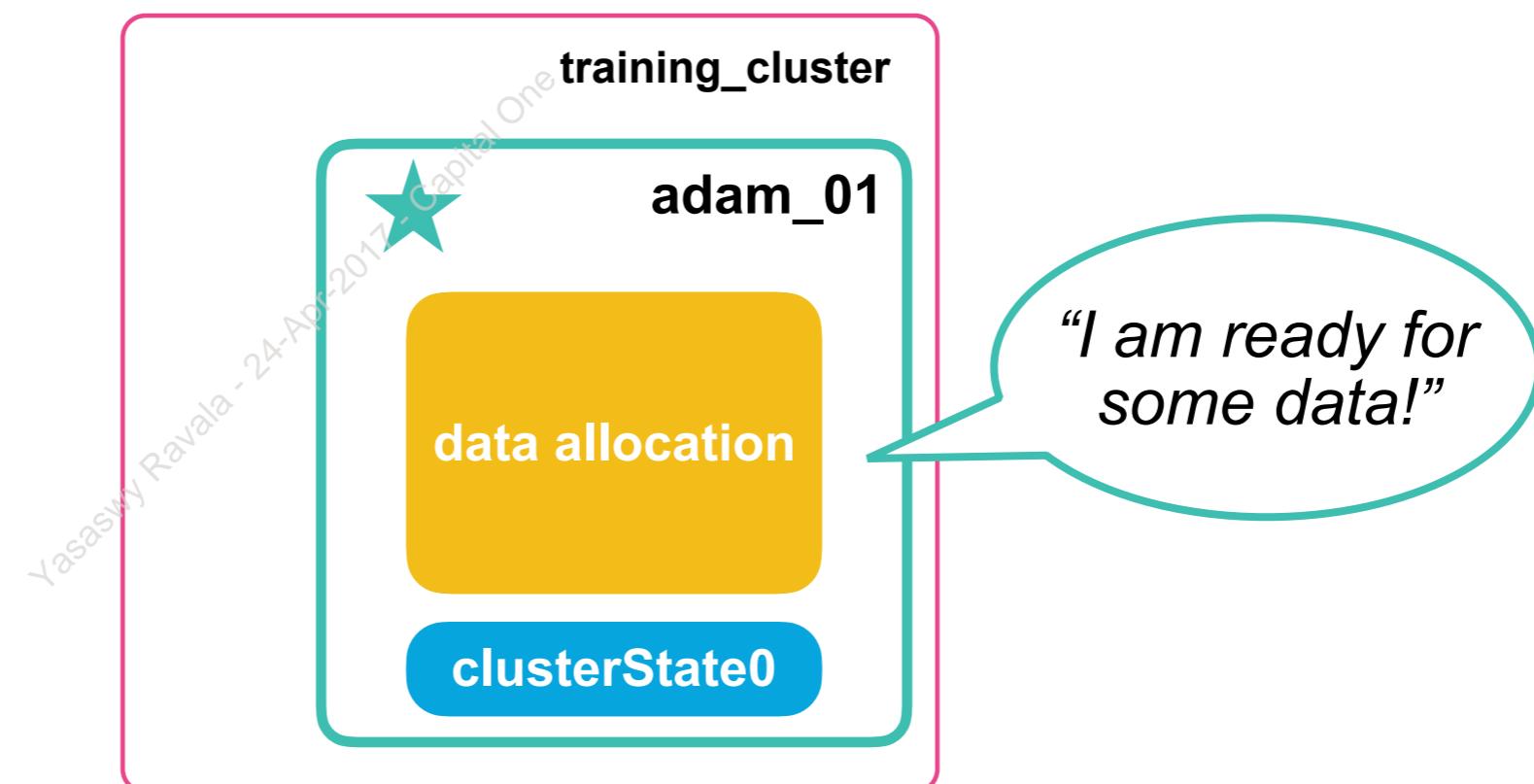
# The Election Process ★

- An *election process* occurs to select the actual master
  - When `adam_01` starts, it elects itself to be the master



# Data Nodes

- We also need to store data in the cluster
- A node that is able to store data is called a ***data node***
  - When **adam\_01** starts, it is a data node by default
  - You can disable by setting **node.data: false**

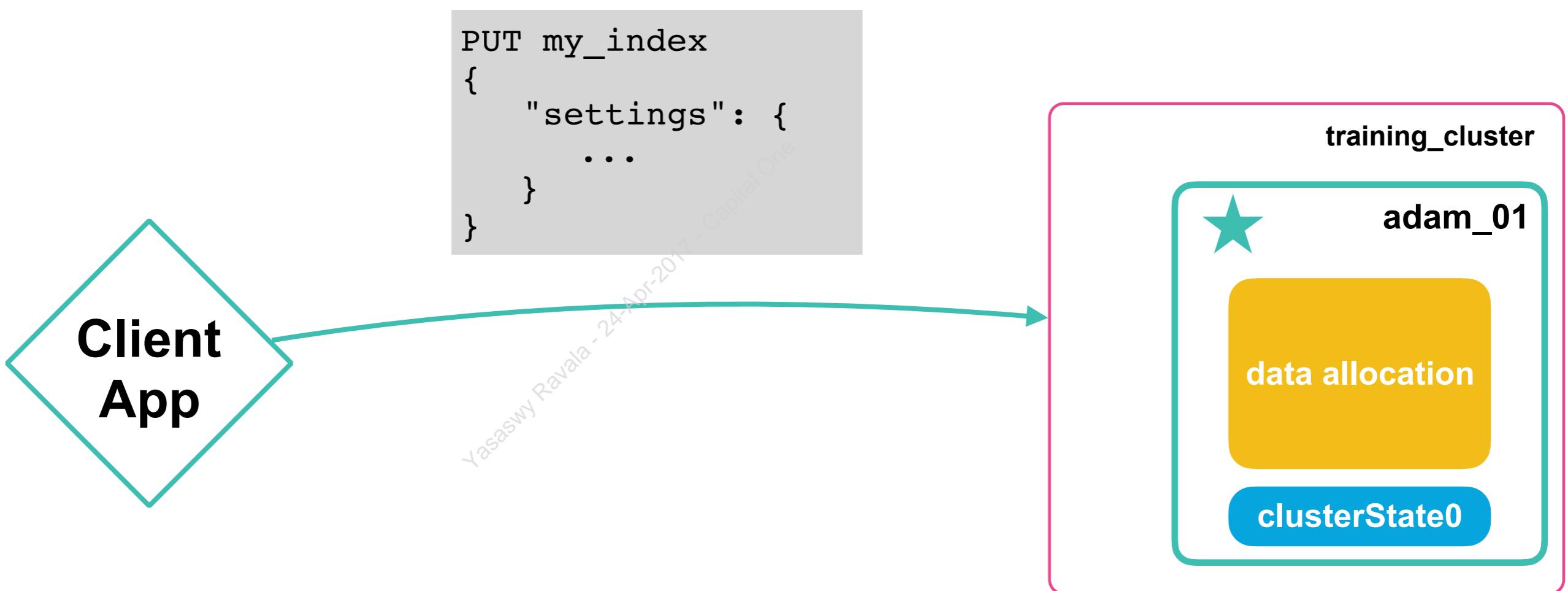


# Creating an Index

Yasaswy Raval - 24-Apr-2017 - Capital One

# Creating an Index

- Suppose you submit a **PUT** request to create a new index in your cluster:



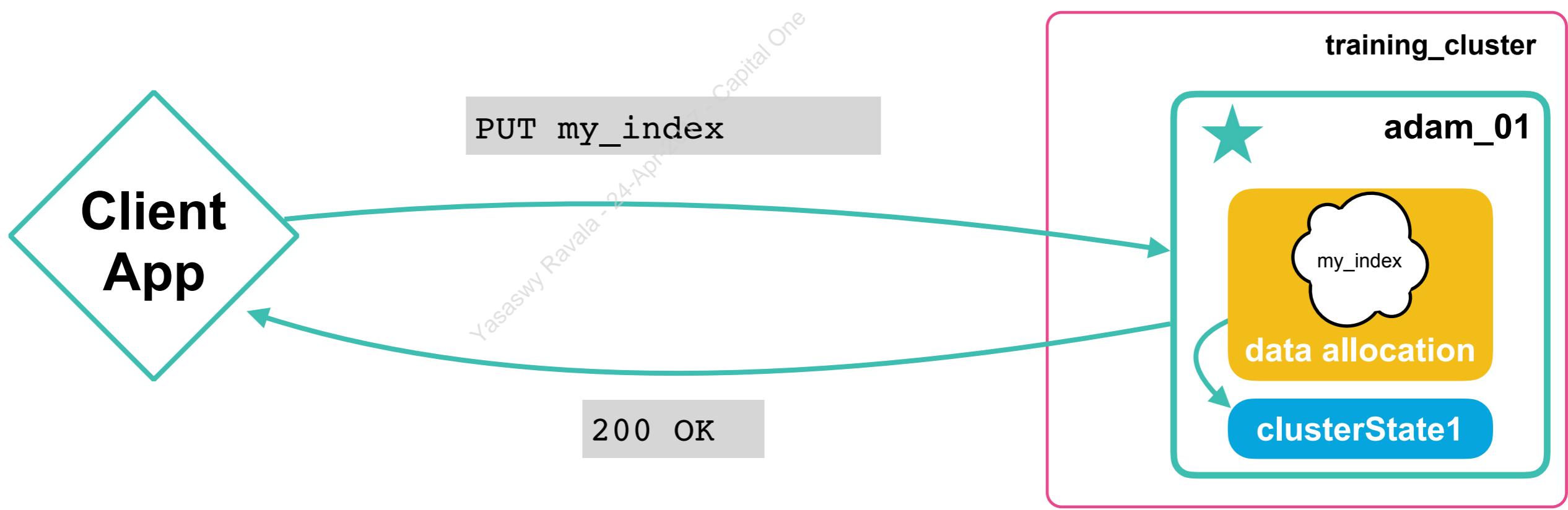
# The Coordinating Node

- A request is not sent to the cluster - it is sent to one of the nodes
- The node that handles your PUT request is called the *coordinating node* for that request
  - It inspects the request and determines what needs to be done
  - It makes sure the request is executed and sends the response back to the client



# Master Nodes Create Indices

- Our **PUT** request for a new index can only be executed by the master node (same for a **DELETE** index request)
  - Since **adam\_01** is the elected master node (and the only node in our cluster), it handles the request for this new index
- The cluster state is updated, the allocation of the index occurs, and **adam\_01** sends back a response



# Starting a Second Node

Yasaswy Raval - 24-Apr-2017 - Capital One

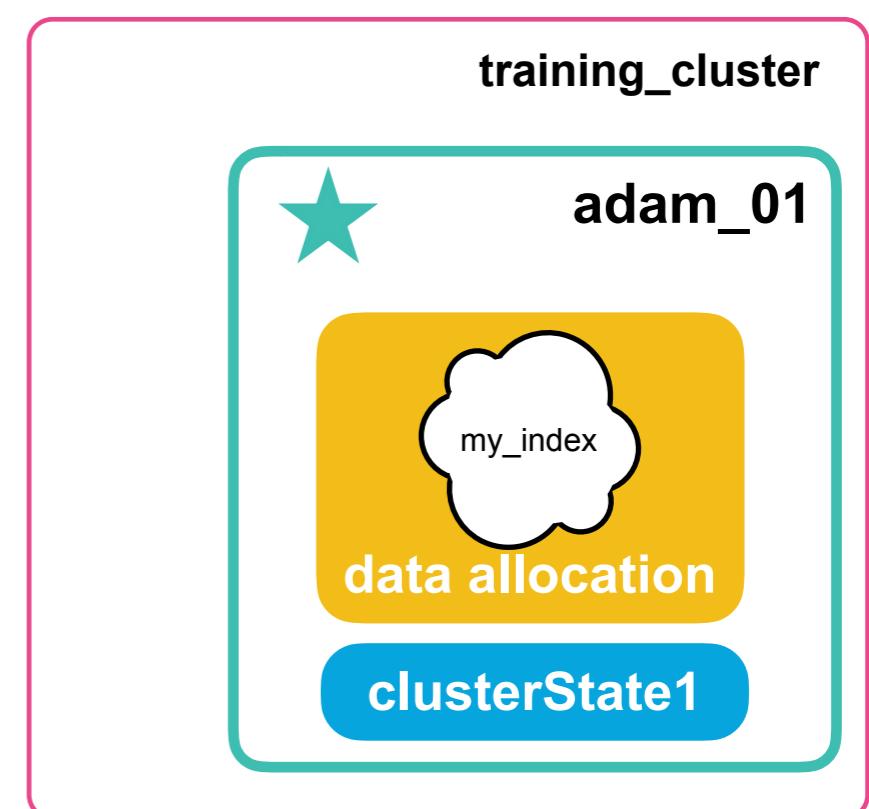
# Starting a Second Node

- Elasticsearch is a distributed system, which means it can benefit from multiple machines.
  - Let's start a new node named **princess\_10** on a new machine.
- By default, **princess\_10** is master-eligible + data node

```
bin/elasticsearch -Enode.name=princess_10 -Ecluster.name=training_cluster
```



Yasaswy Raval - 24-Apr-2017 - Capital One



# Joining the Cluster

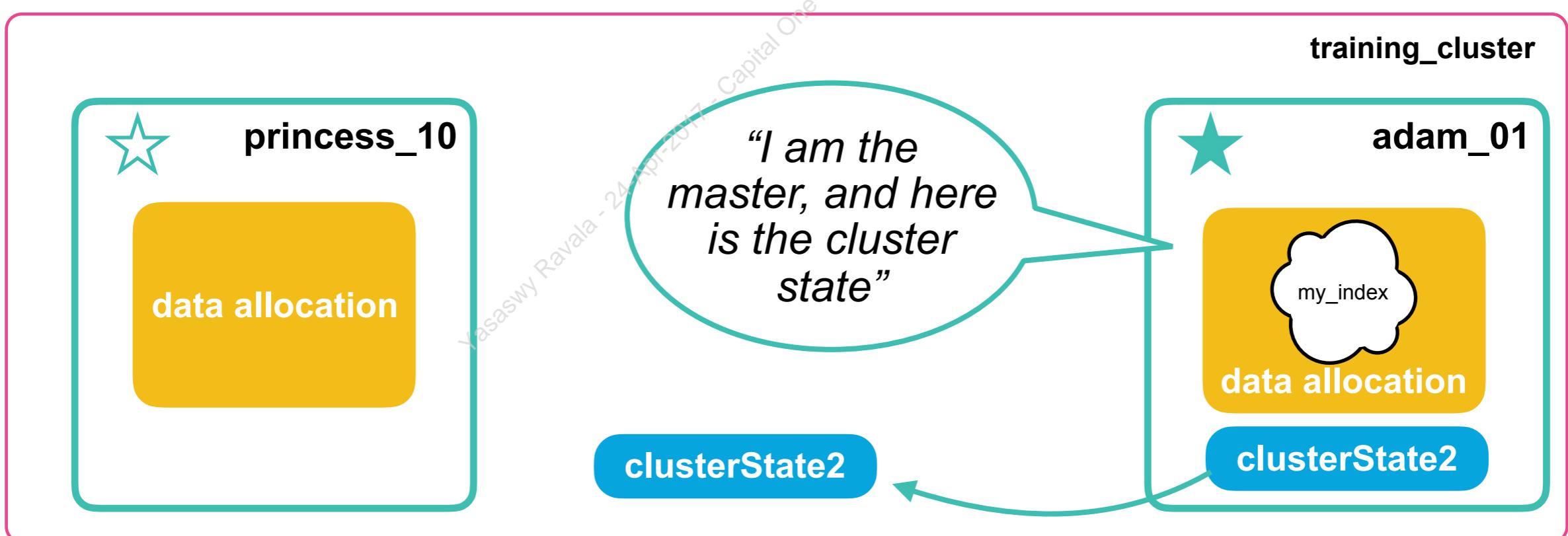
- **princess\_10** has to join the cluster, but how?
  - Nodes ping each other using unicast (on port 9300 by default)
- A seed list of hostnames should be listed in the **unicast.hosts** property:

```
discovery.zen.ping.unicast.hosts: [ "adam_01" ]
```



# Sharing the Cluster State

- **adam\_01** updates the cluster state
  - adds **princess\_10** to the list of nodes
- Then **adam\_01** sends the new cluster state to **princess\_10**

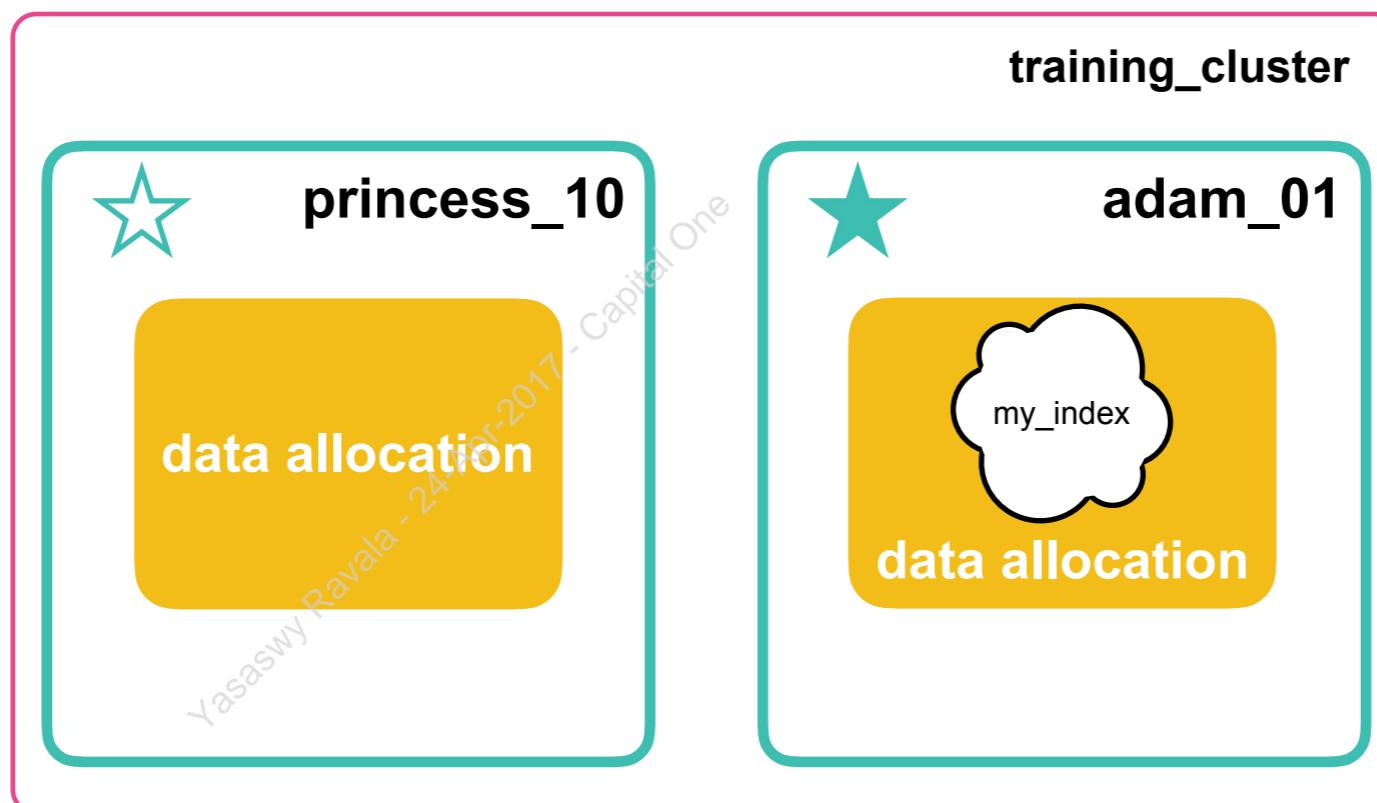


# Shards: Distribution of an Index

Yasaswy Raval - 24-Apr-2017 - Capital One

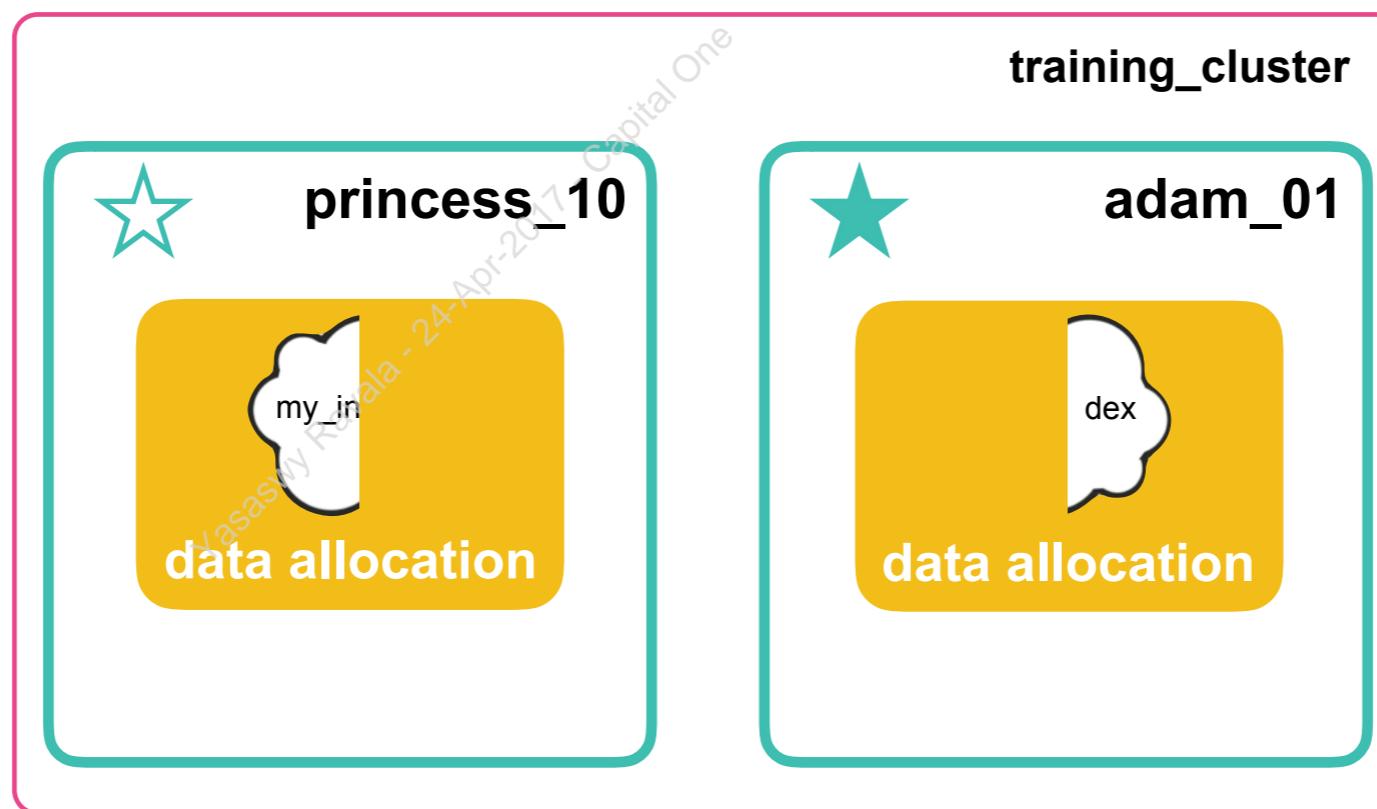
# Distributing Data

- We now have 1 cluster, 2 nodes, and 1 index
- How can Elasticsearch use both nodes?
  - By distributing the data!



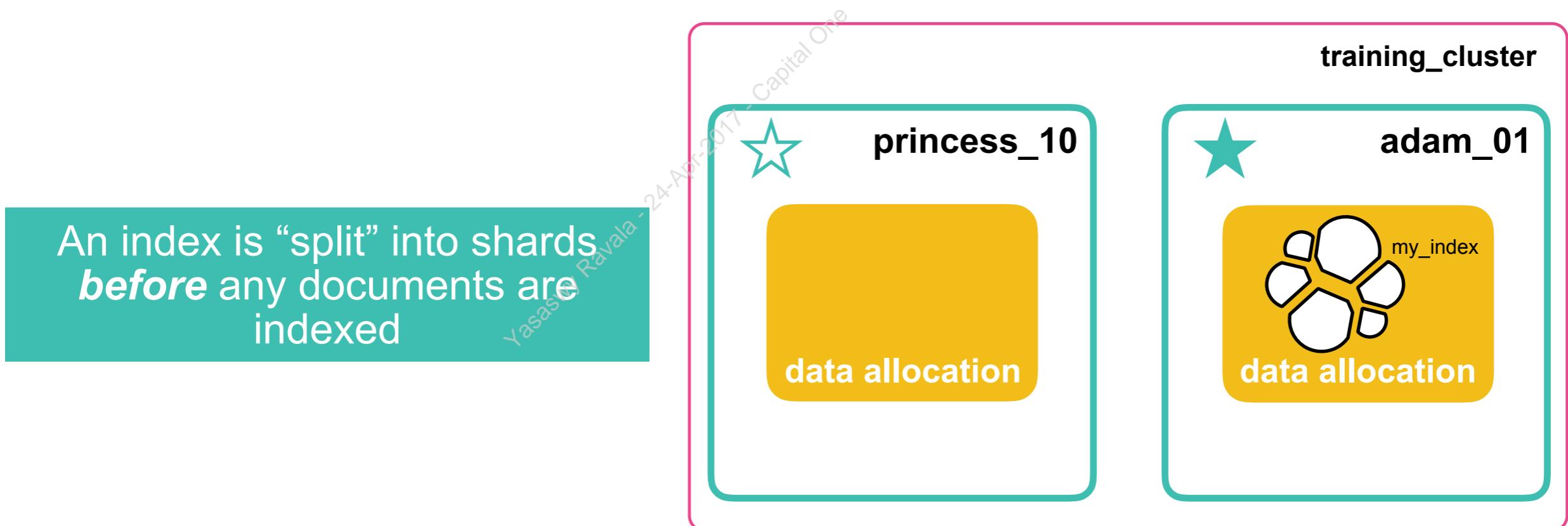
# "Late" Data Splitting is Expensive

- Elasticsearch "could" break an index into parts late *after* creating an index, but that has some challenges:
  - how many parts?
  - how to divide documents?
  - need to rebuild the entire inverted index!



# “Early” Data Splitting

- Instead, Elasticsearch breaks an index into parts early, *at creation time*
- An index is a virtual namespace which points to a number of **shards**
  - A **shard** is a worker unit that holds data and can be assigned to nodes

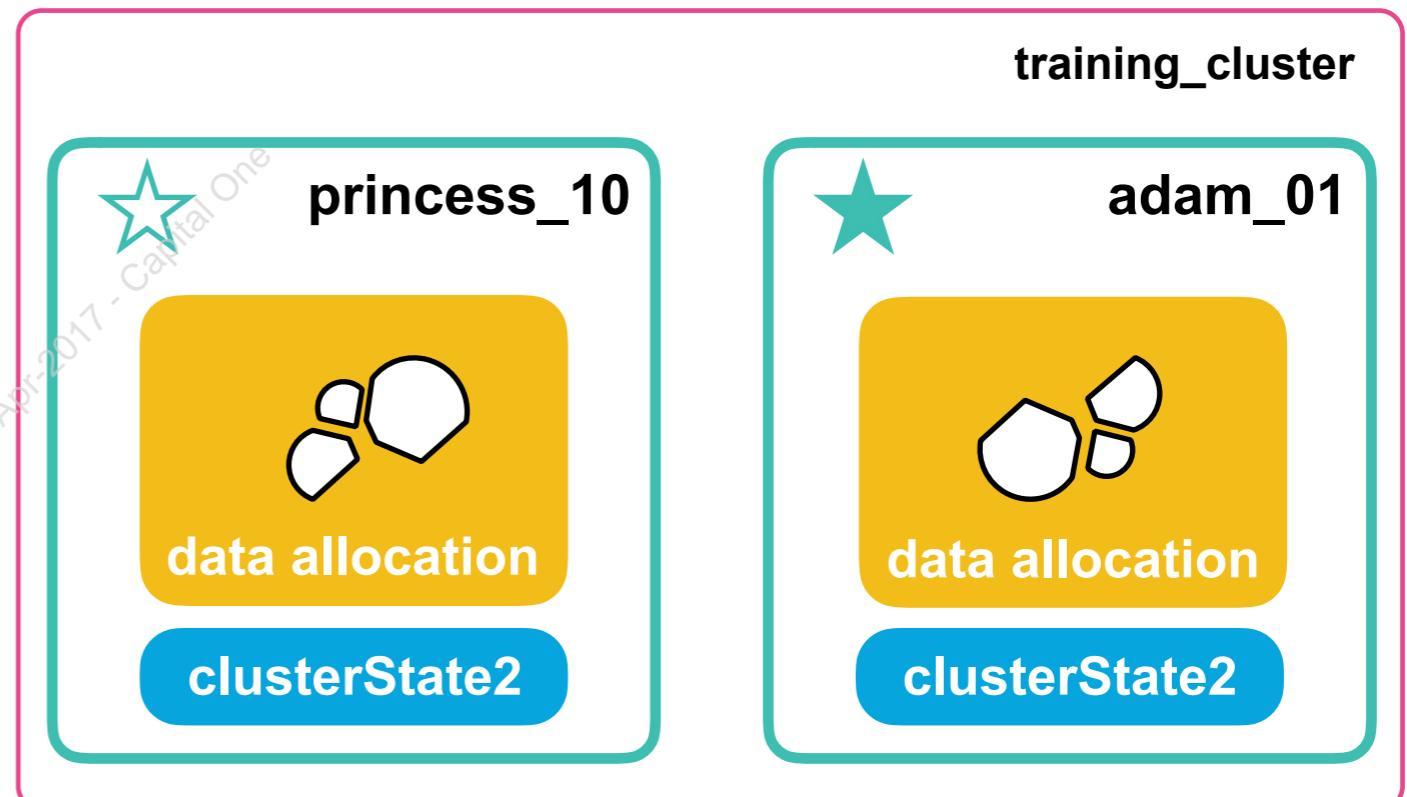


# Shards are distributed across nodes

- Because every index is partitioned into shards at creation time, it is easy to distribute them later
  - The master decides which shards to allocate to which nodes (published via cluster state)

**clusterState2** contains the shard allocation table

When **princess\_10** receives **clusterState2**, she starts the request for copying the shards assigned to her



# The Number of Primary Shards is Fixed

- The default number of primary shards for an index is 5
  - You specify the number of shards when you create the index
  - You can NOT change the number of primary shards, so choose wisely! (or *reindex your data*)

```
PUT my_index
{
  "settings": {
    "number_of_shards": 6
  }
}
```

# Distributing Documents

Yasaswy Raval - 24-Apr-2017 - Capital One

# Distributing Documents

- If an index is partitioned into shards, what happens when you index a document?
- We have already discussed the PUT vs. POST commands
  - PUT: when you specify the `_id`
  - POST: when you want Elasticsearch to generate the `_id`

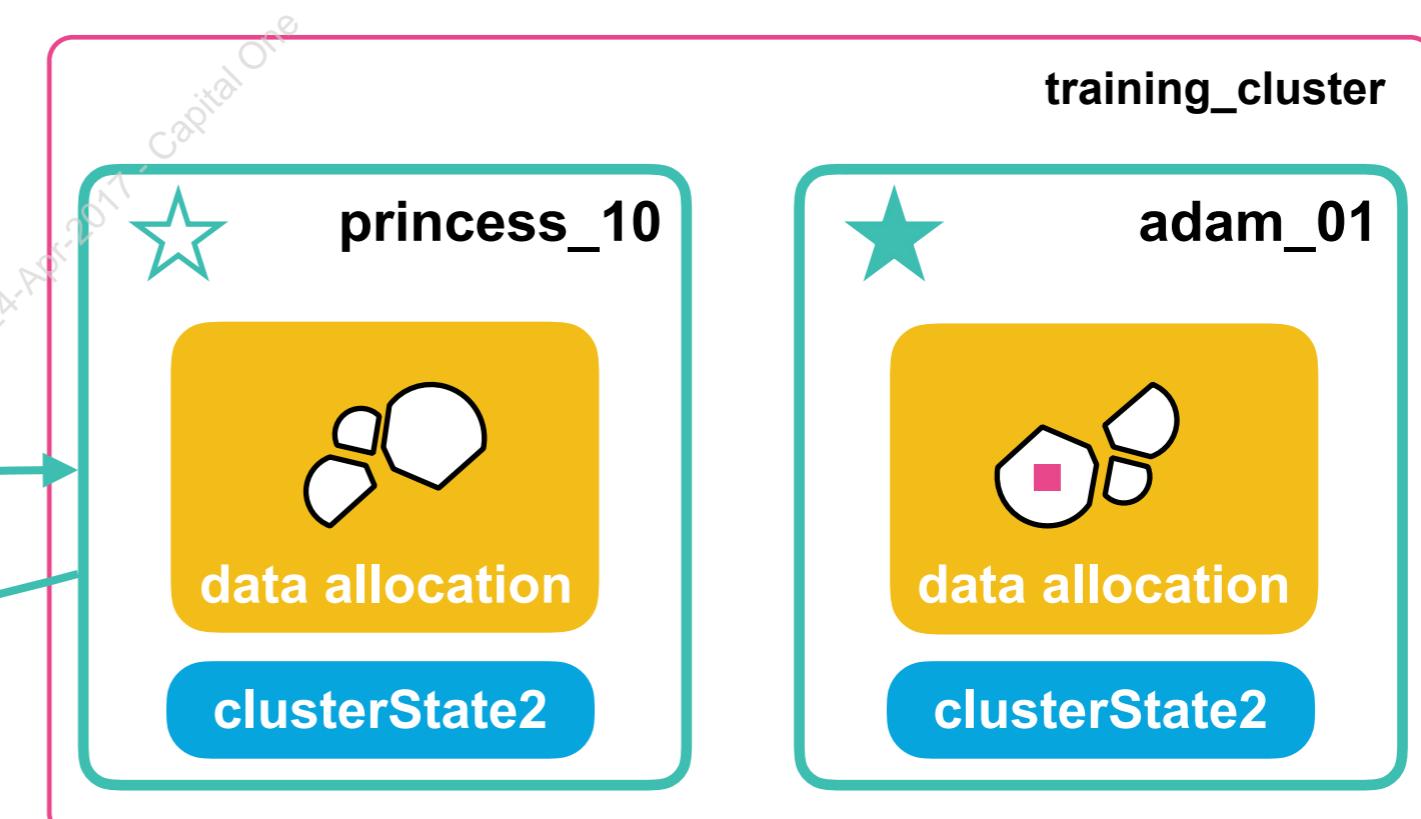
```
PUT my_index/doc/101
```

```
{  
  "firstname" : "Robert",  
  "lastname" : "Smith",  
  "city" : "Lancashire"  
}
```

Client  
App

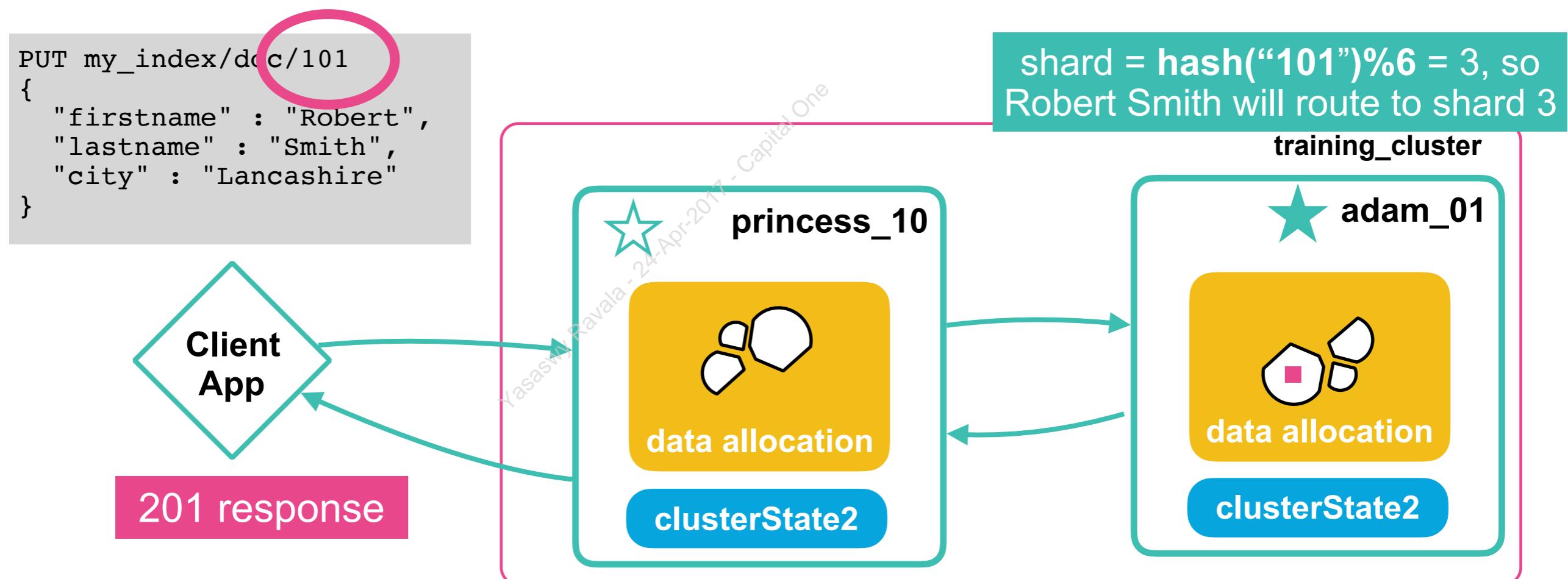
201 response

Yasaswy Pareek 24-Apr-2017 - Capital One



# Routing Documents to Shards

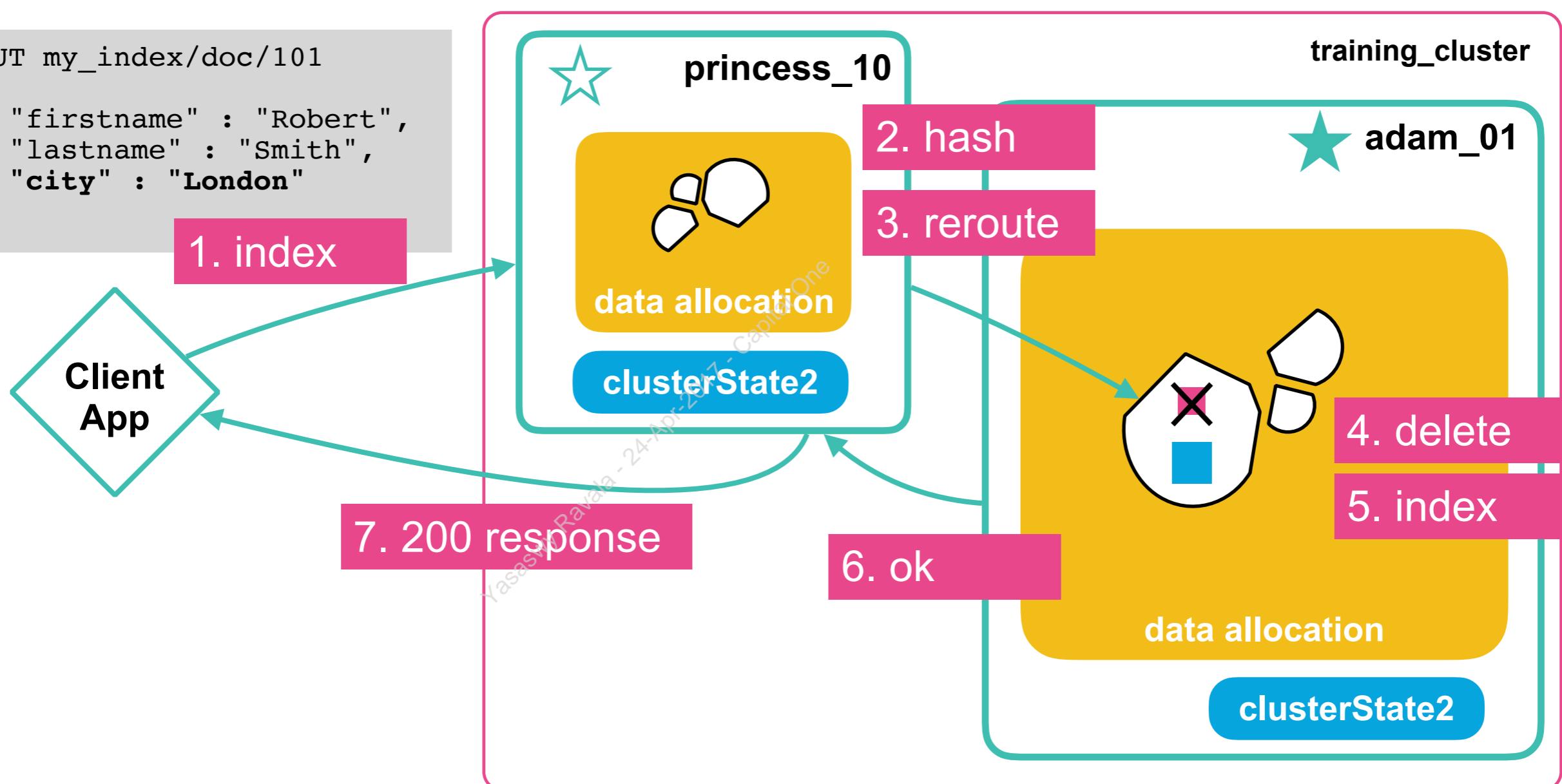
- Documents are routed to shards using the following formula: `shard = hash(_routing) % number_of_primary_shards`
  - The document's id is used as the `_routing` value by default
  - This typically results in uniformly-sized shards (our goal!)



# Reindexing a Document

- If you index an existing document, it gets reindexed
  - Behind the scenes, the existing document is deleted and then a new one is created

```
PUT my_index/doc/101
{
  "firstname" : "Robert",
  "lastname" : "Smith",
  "city" : "London"
}
```



# The `_create` Operation

- The `_create` operation can explicitly be used when indexing a document
  - If the **id** already exists, the operation fails

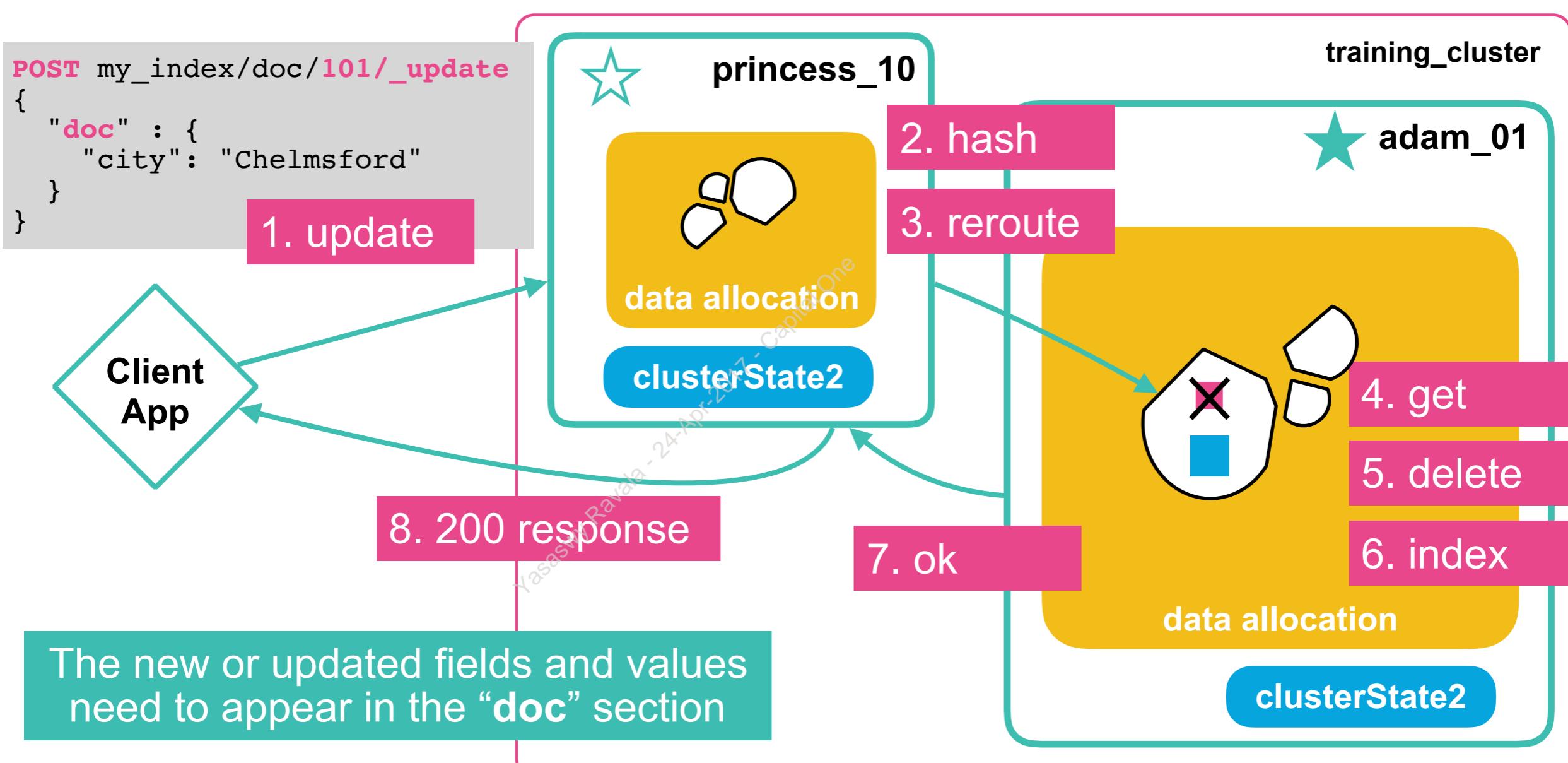
```
PUT my_index/doc/101/_create
{
  "firstname" : "Robert",
  "lastname" : "Smith",
  "city" : "Lancashire"
}
```

Returns a 409 error code if  
**id=101** is already indexed



# Update a Partial Document

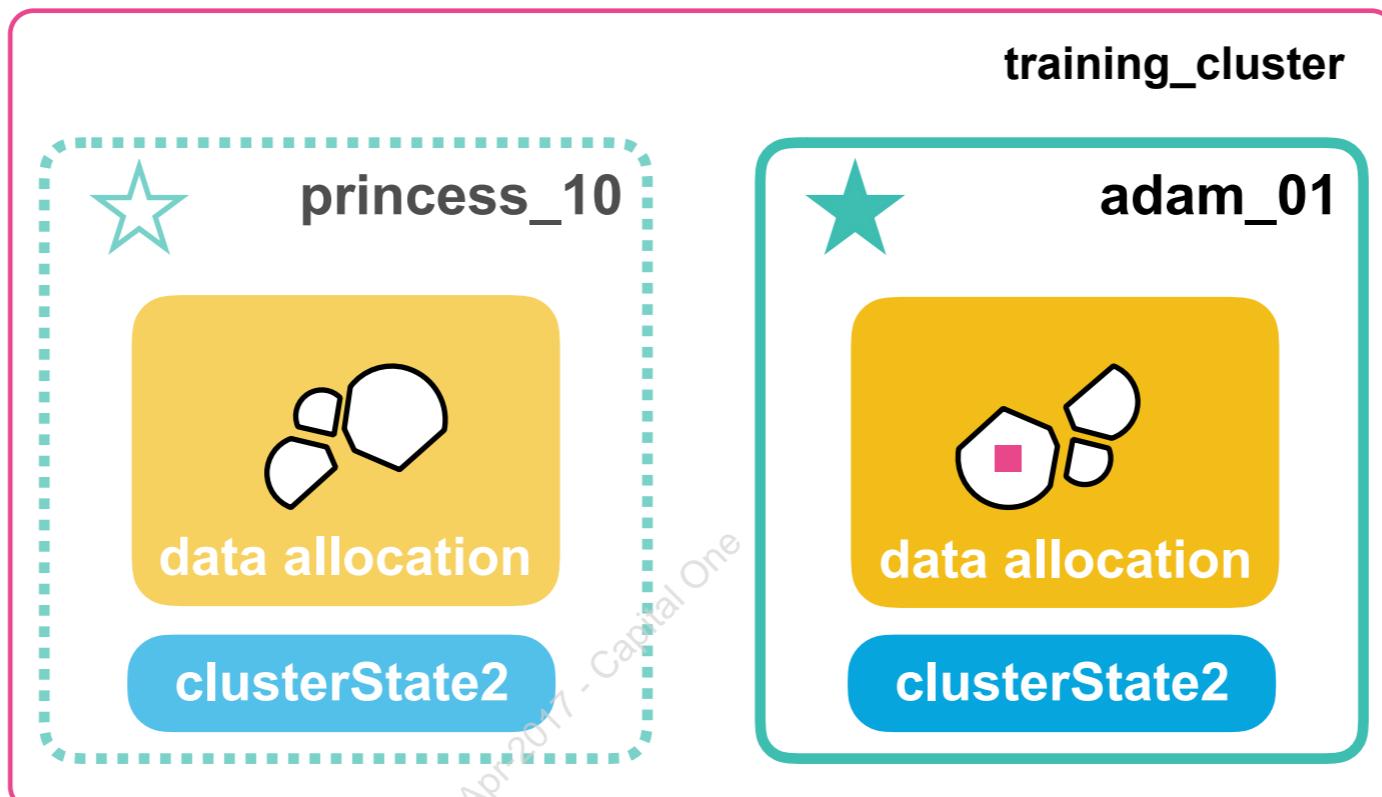
- If you just want to update one or more fields, you can use the `_update` endpoint
  - The entire document is still deleted and reindexed



# Replication

Yasaswy Raval - 24-Apr-2017 - Capital One

# Why Replication?

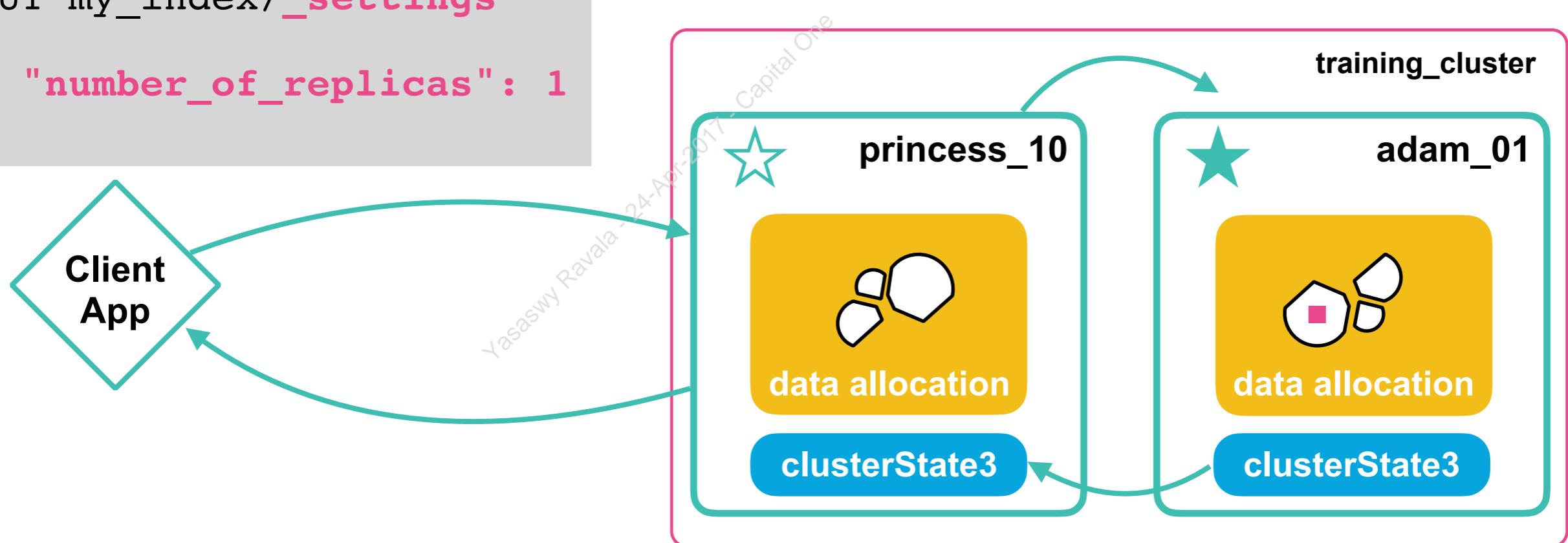


If a node fails, we could potentially lose data

# Shard Replication

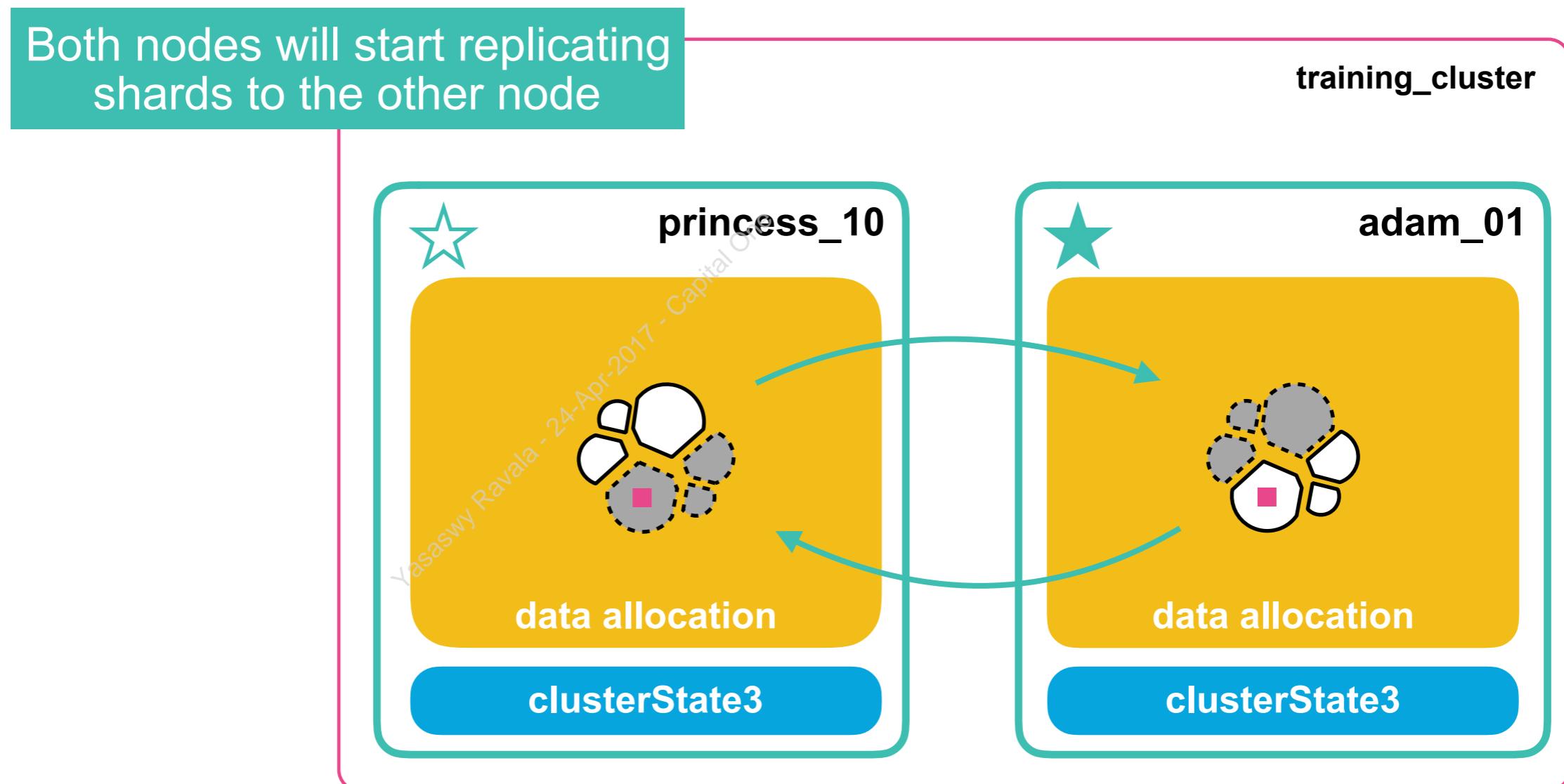
- Elasticsearch makes copies of your shards called *replicas*
  - The original shard is called the *primary*
- You can also change the number of replicas for an index at any time using the `_settings` endpoint:

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



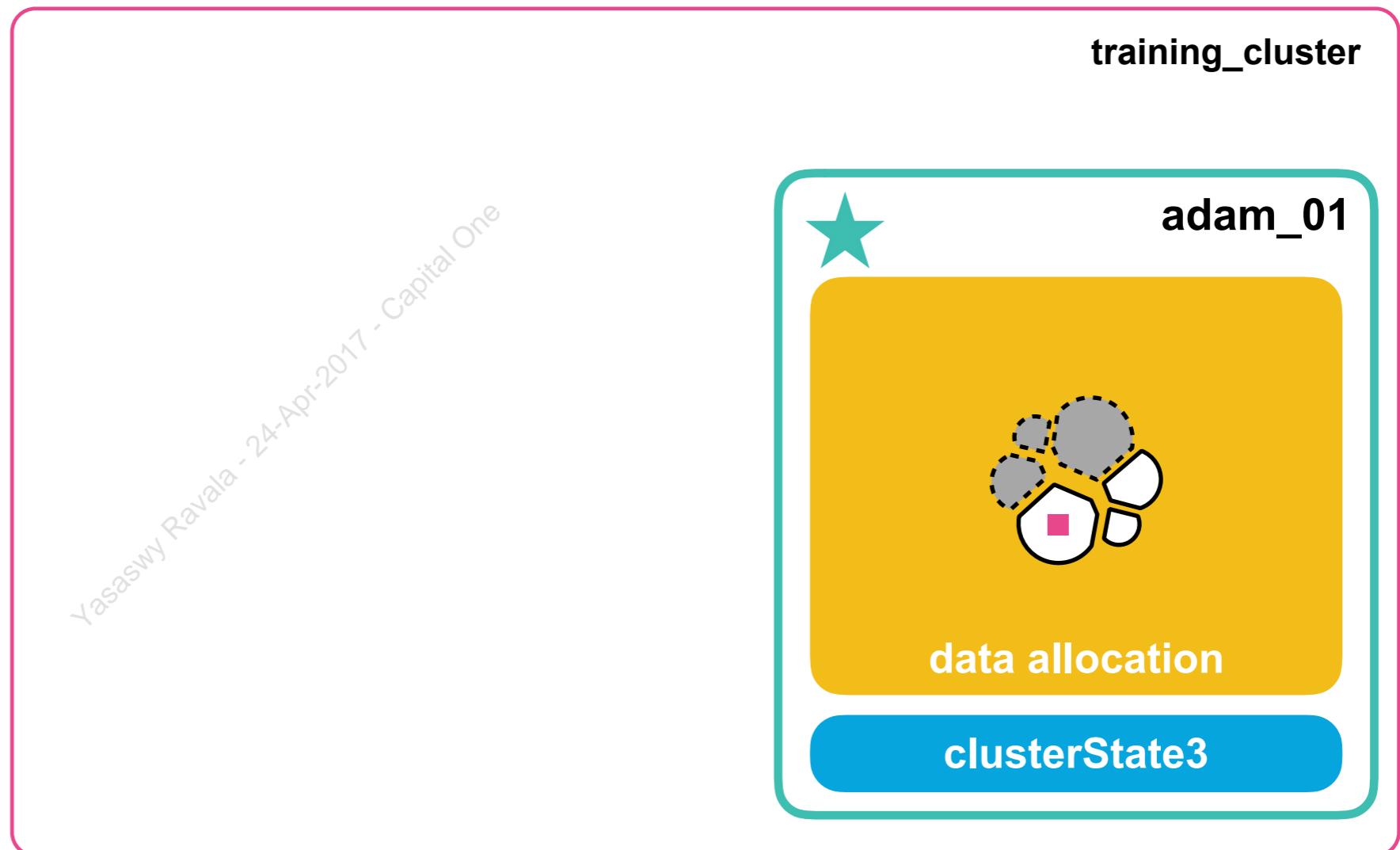
# Shard Replication

- The replicas are stored on different nodes in the cluster
  - provides high availability in case a shard or node fails
  - also improves search performance by scaling the processing



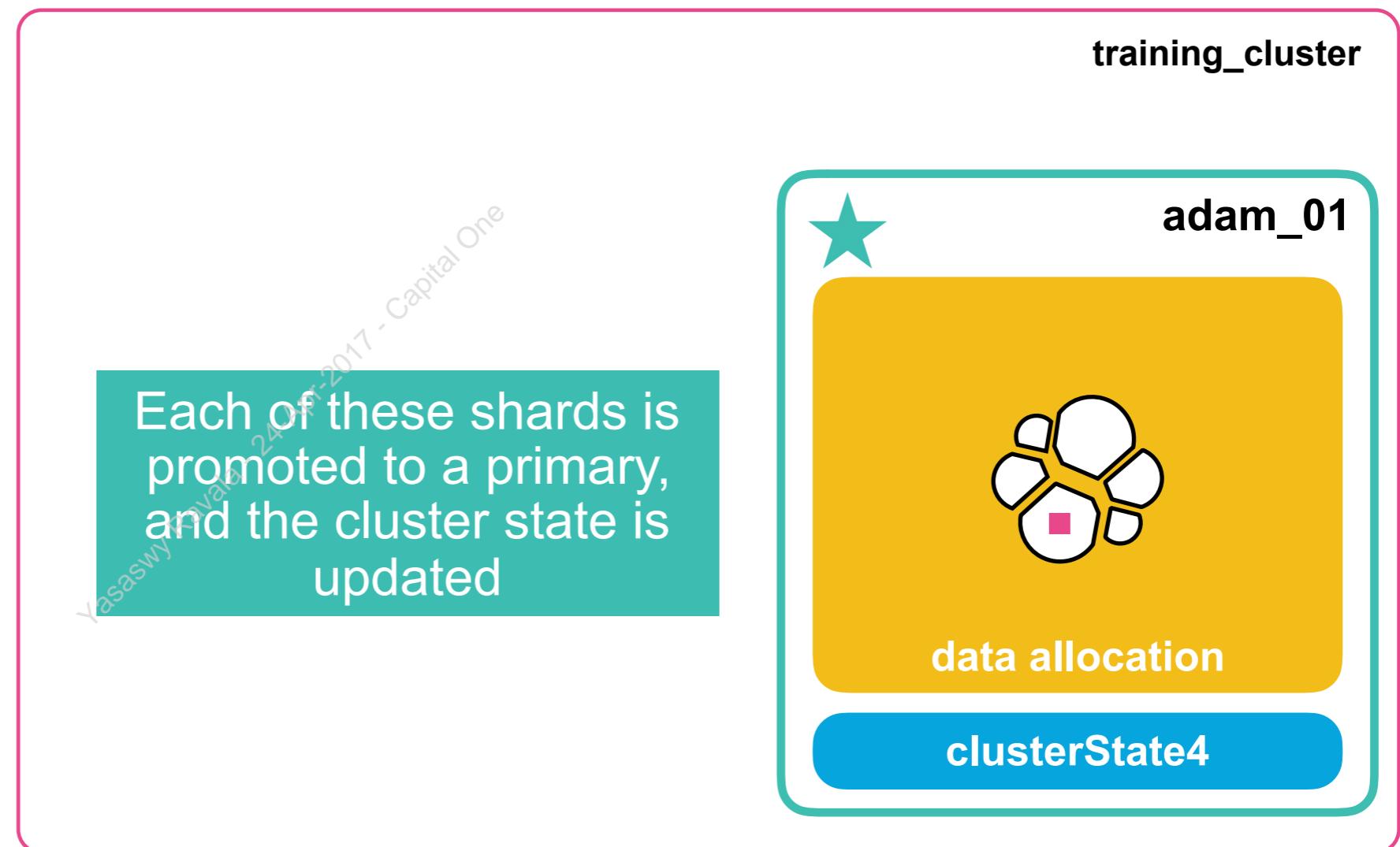
# High Availability

- If we lose **princess\_10** the entire data is available in **adam\_01**



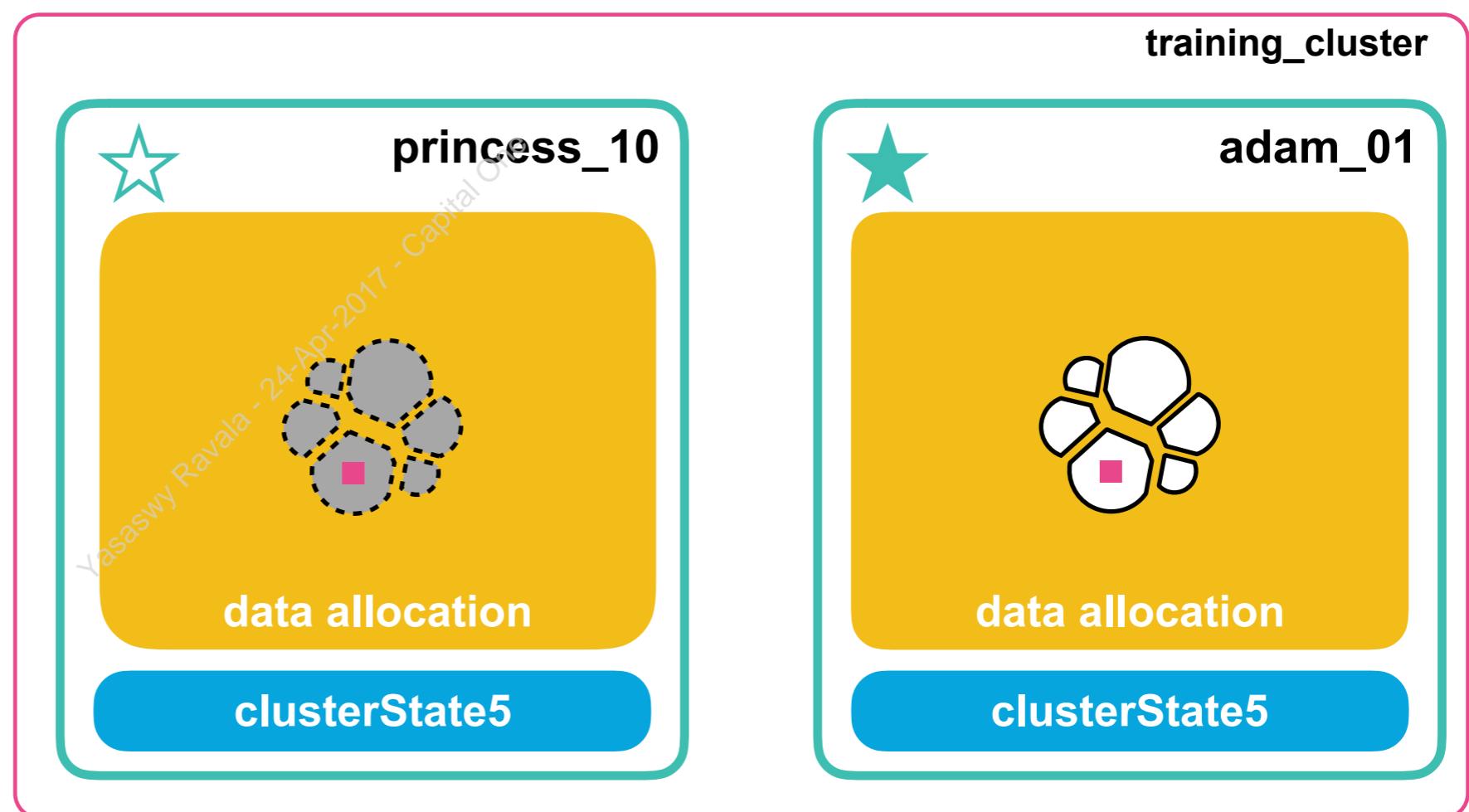
# Primary Promotion

- After losing a primary, Elasticsearch will automatically promote a replica into a primary



# Node Re-join

- When **princess\_10** is back online:
  - it joins the cluster again
  - gets the new cluster state
  - replicates shards



# Let's Delete the Existing Indices

```
DELETE my_index,my_index2
```

```
DELETE my_index*
```

```
DELETE my_index  
DELETE my_index2
```

Several ways to delete multiple indices

training\_cluster



princess\_10

data allocation

clusterState6



adam\_01

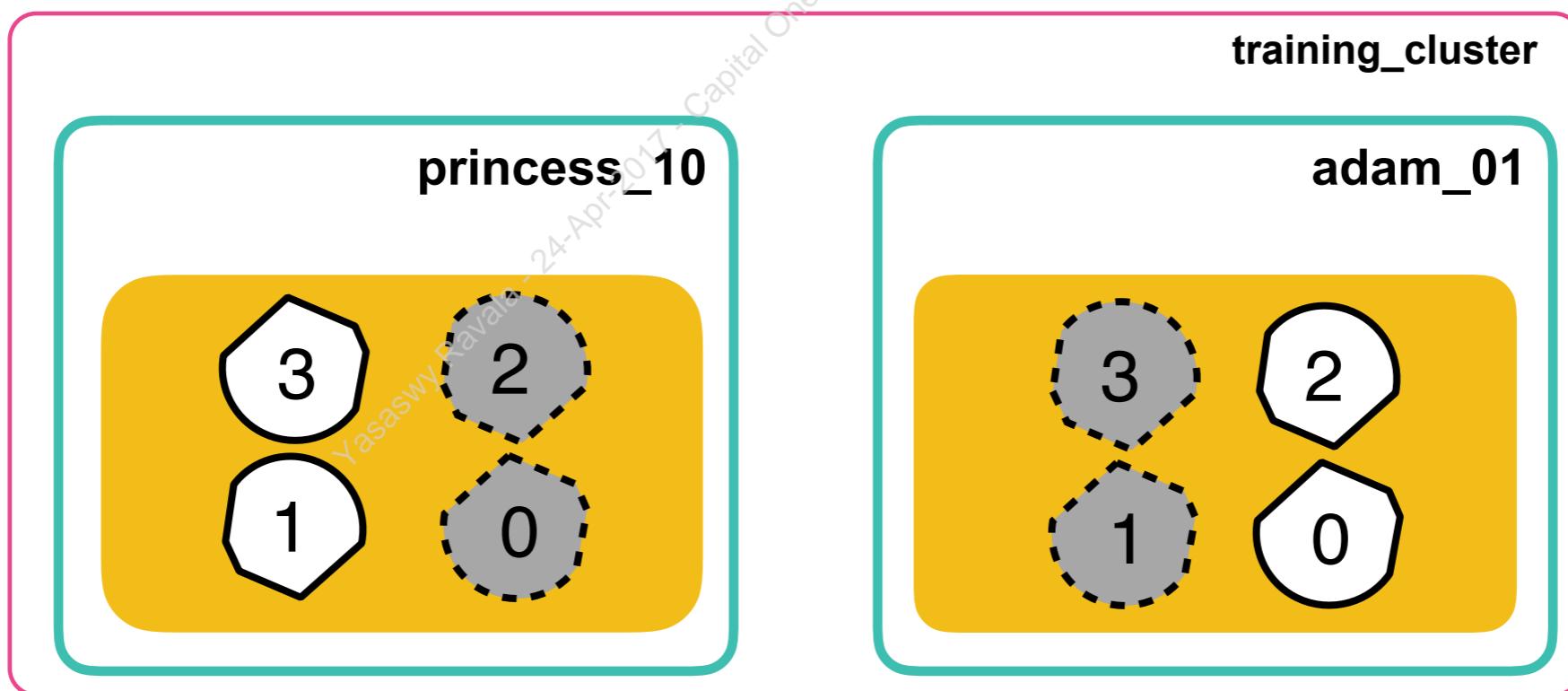
data allocation

clusterState6

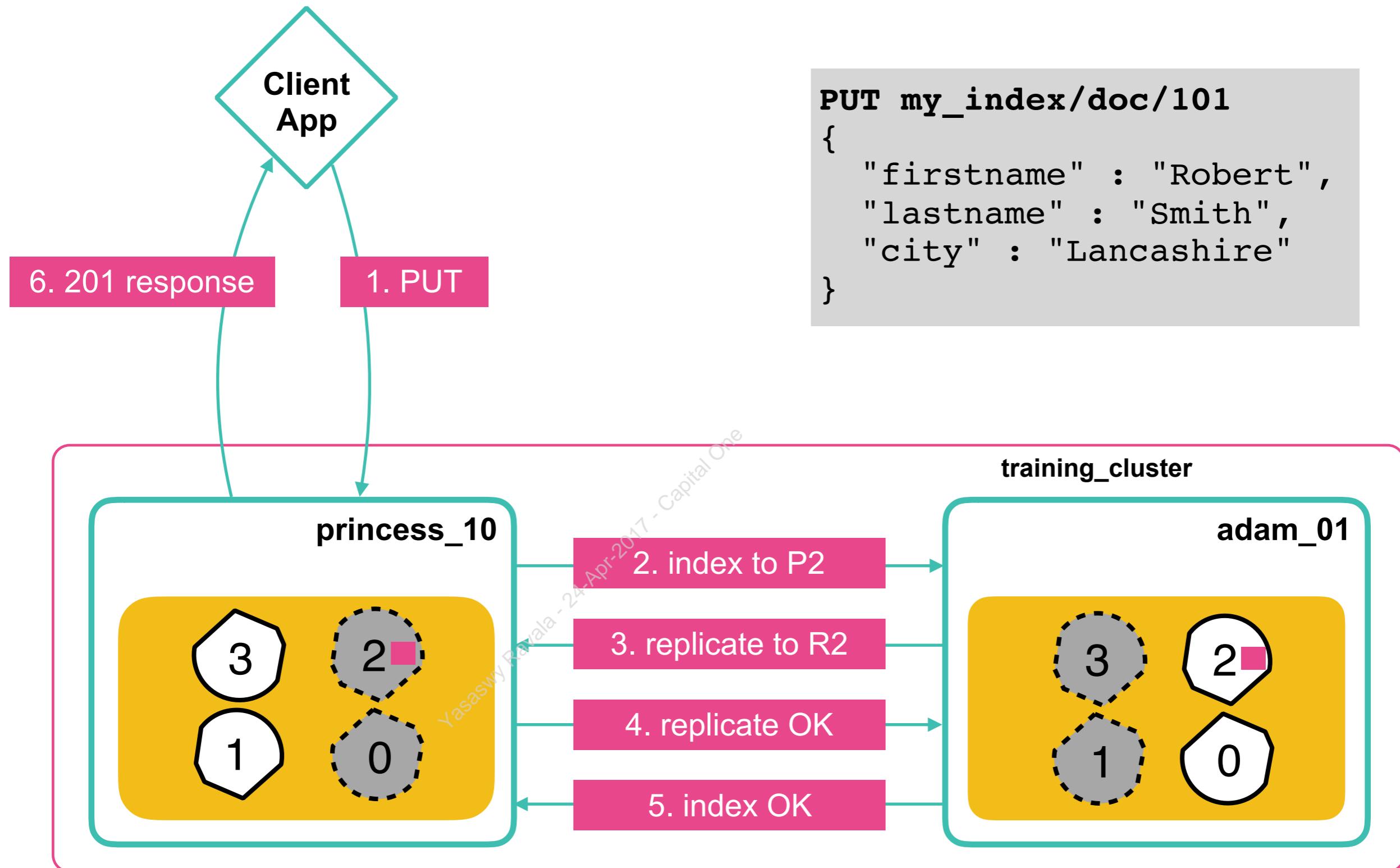


# Let's Start Over

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



# Let's add the same document again:



# The Response of a PUT

- Notice the response of a **PUT** contains a **\_shards** section
  - “**total**” = 1 primary + # of replicas
  - successful < total**, then a replica is likely not available

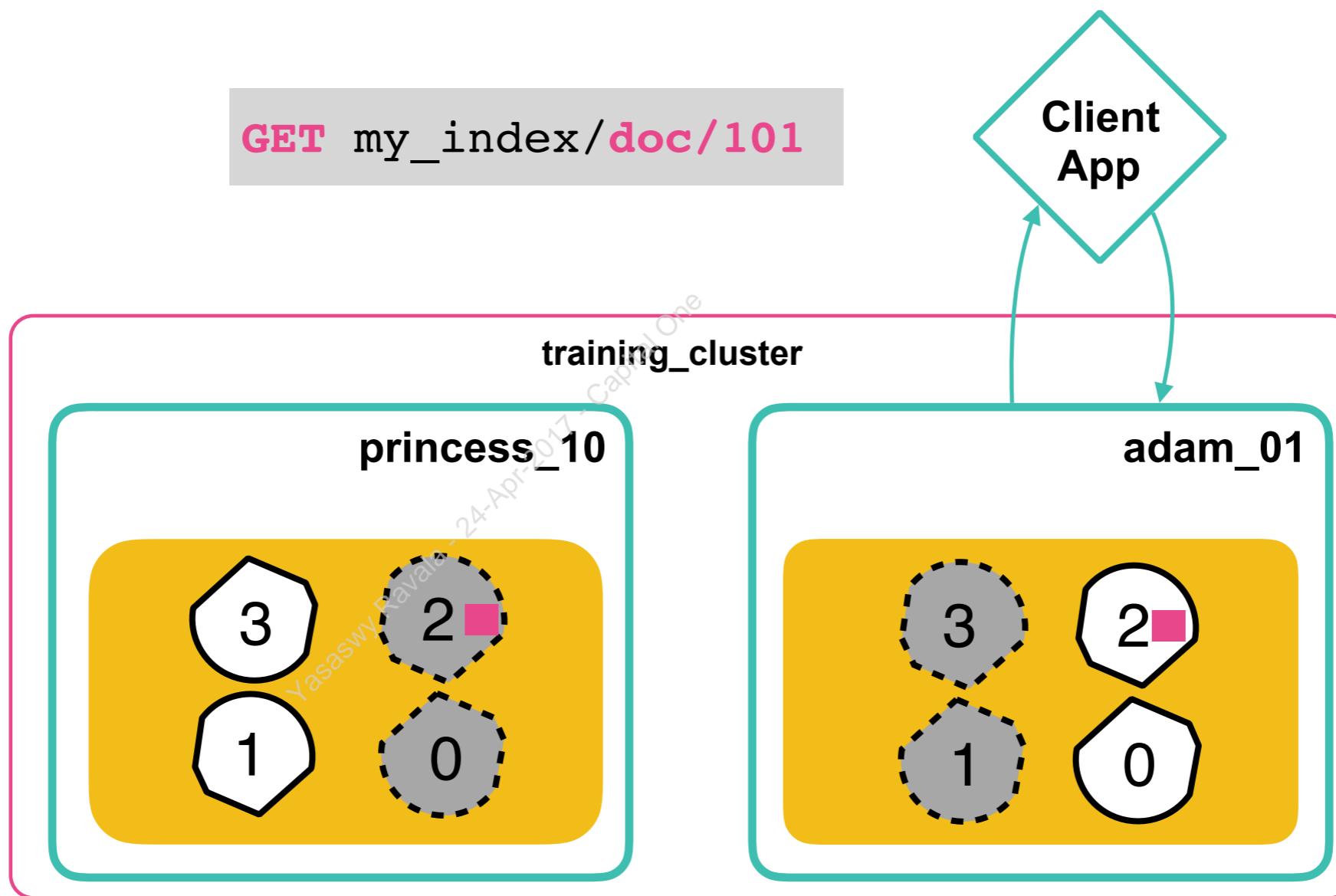
```
HTTP/1.1 201 Created ← 201 = created, 200 = updated
{
  "_index": "my_index",
  "_type": "doc",
  "_id": "202",
  "_version": 1,
  "result": "created",
  "_shards": {
    "total": 2, ← # of shard copies the doc
    "successful": 1, ← should be written to
    "failed": 0 ← # of shard copies the doc was
  }, ← successfully written to
  "created": true
}
```

“**failed**” is an array of errors if something went wrong



# The Get API

- Use the **Get API** to retrieve a specific document from the index using its **type** and **id**
  - returns a 404 error if the document is not found

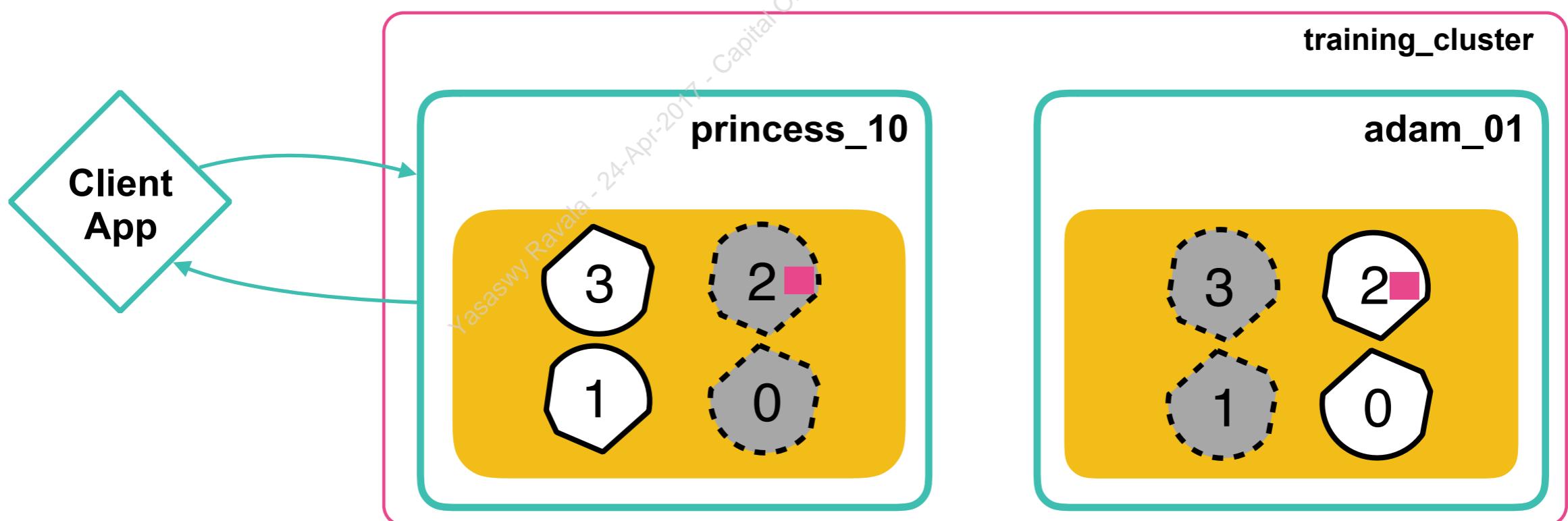


# Let's delete the document

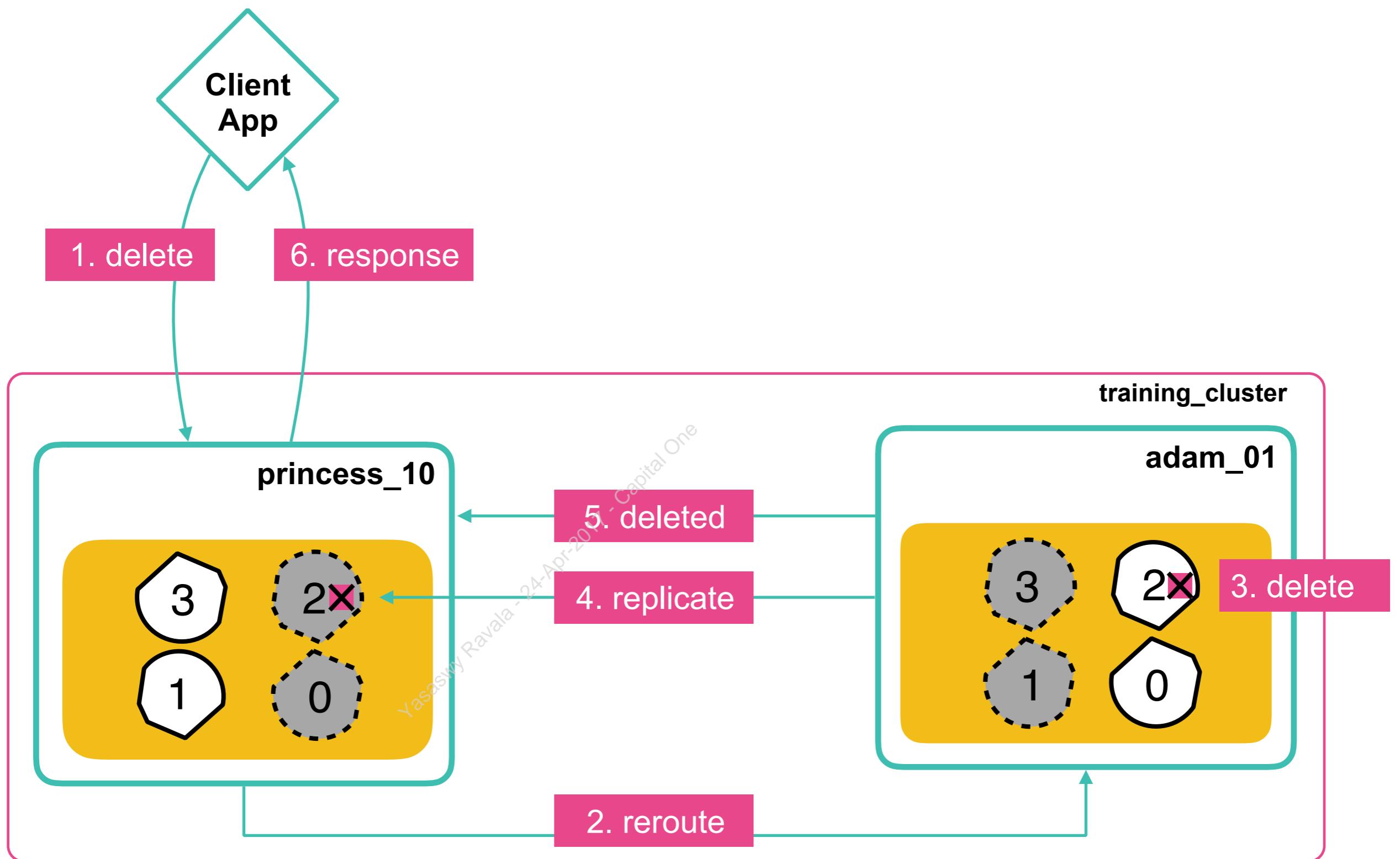
- Use **DELETE** to delete a specific document
  - 200 response if successful
  - 404 response if the document is not found
- Delete marks the document as deleted

```
DELETE my_index/doc/101
```

The document is not actually deleted from storage until a segment **merge** occurs



# The DELETE request in detail:

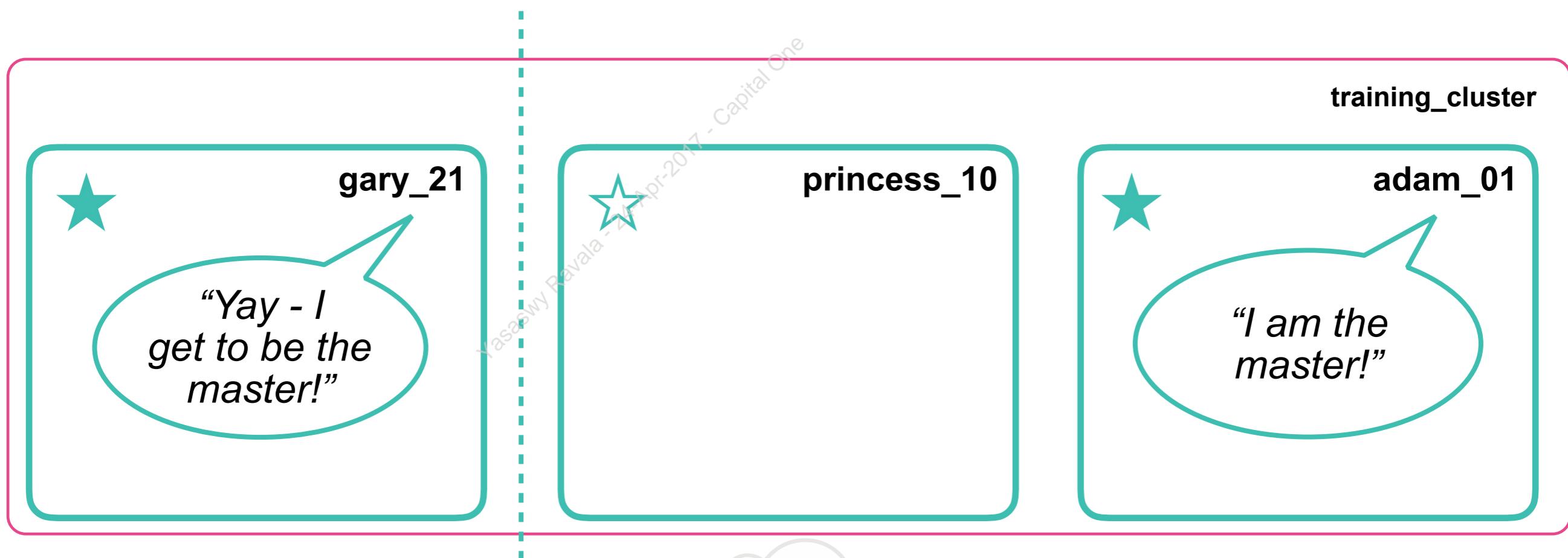


# Split Brain

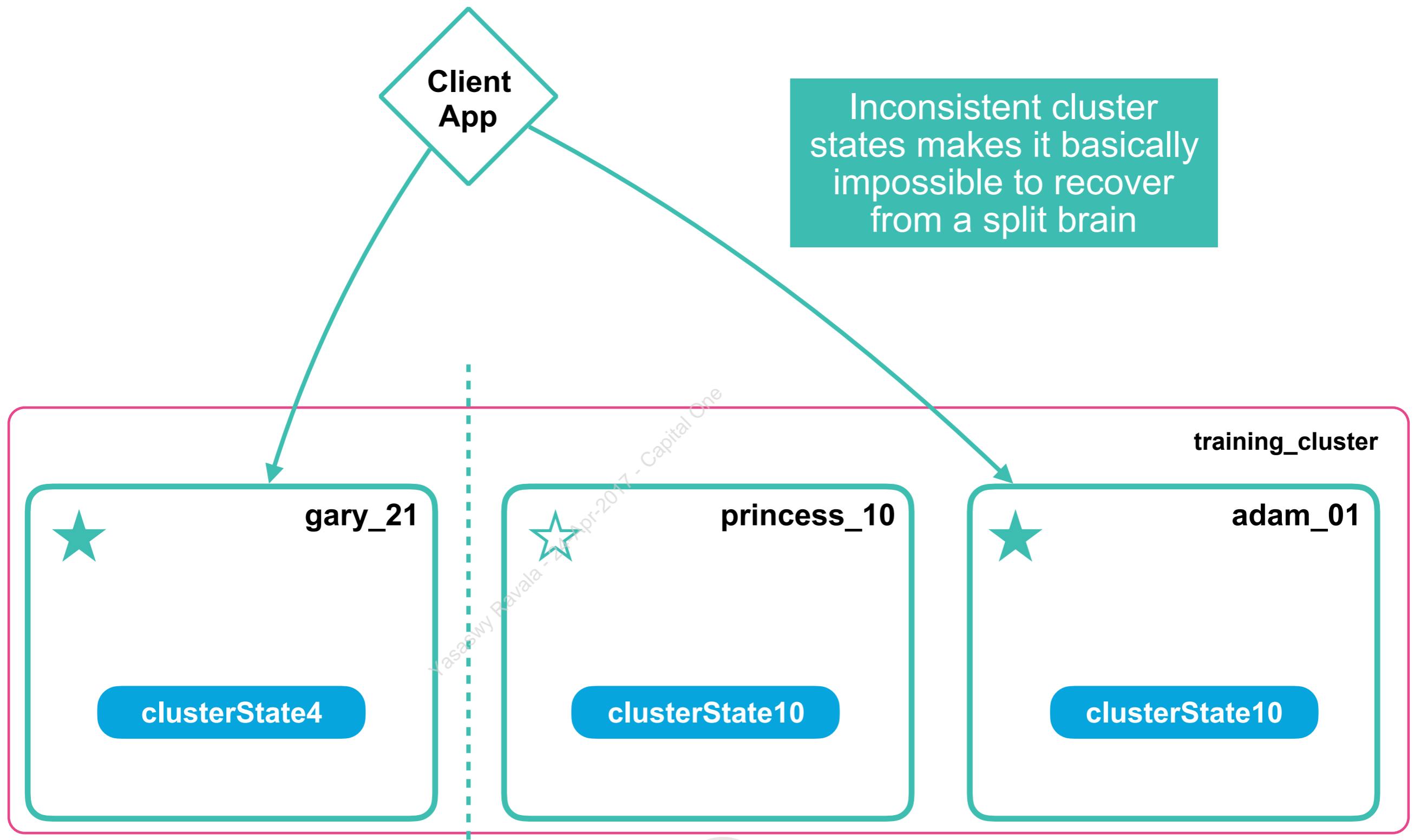
Yasaswy Raval - 24-Apr-2017 - Capital One

# Split Brain

- Even though the master node is the single source of truth when it comes to the cluster state, problems can still happen
  - In particular, a big concern is if the cluster gets **partitioned**
  - The cluster could inadvertently elect two masters, which is referred to as a “**split brain**”

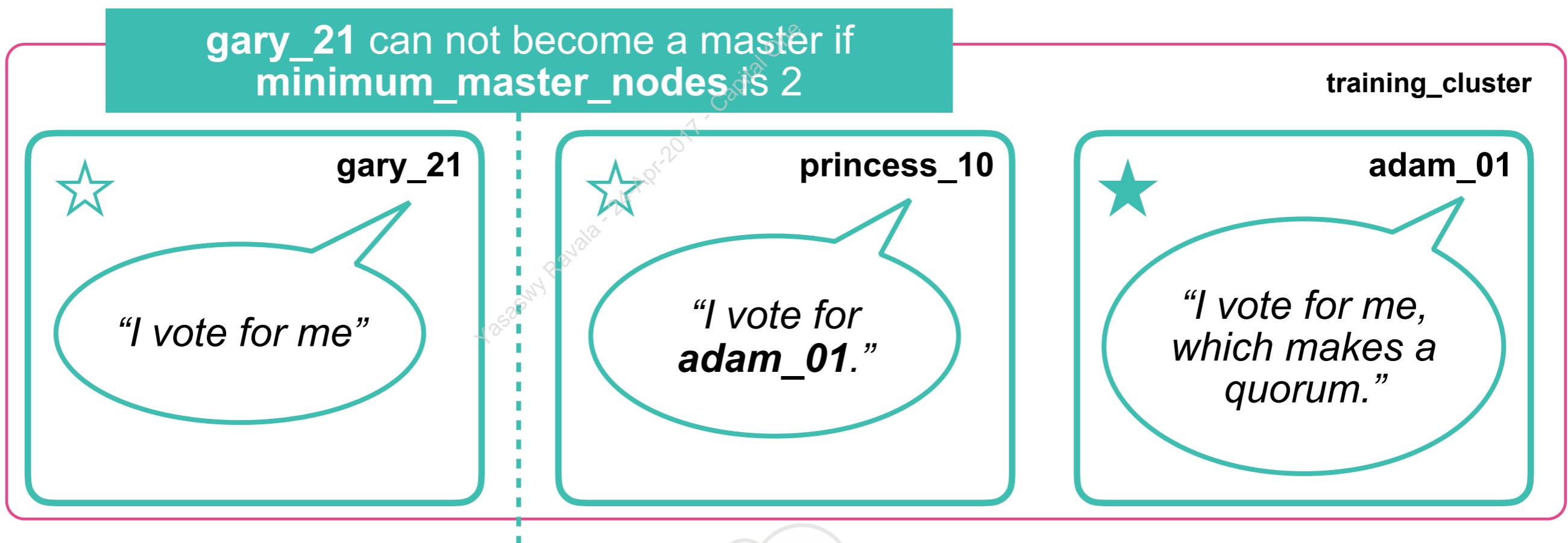


# Two masters can lead to inconsistency:



# Avoiding the Split Brain

- A master-eligible node needs at least **minimum\_master\_nodes** votes to win an election
  - Setting it to a quorum helps avoid the split brain scenario
- Our simple recommendation is for production systems to always have 3 dedicated master-eligible nodes
  - and set **minimum\_master\_nodes** to 2



# Chapter Review

Yasaswy Raval - 24-Apr-2017 - Capital One

# 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 document is routed to a shard using a formula that incorporates a hash value of the document id
- You can not change the number of primary shards of an index after it has been created
- After losing a primary, Elasticsearch will automatically promote a replica into a primary
- A master-eligible node needs at least **minimum\_master\_nodes** votes to win an election



# Quiz

1. How does Elasticsearch distribute your data within an index?
2. Suppose you have a cluster with 6 data nodes, you create a new index with “PUT hello”, then insert 10,000 documents. How many total shards will the **hello** index have?
3. **True or False:** When you add a new node to a cluster, you have to run the `_refresh` command to redistribute shards to the new node?
4. If **number\_of\_shards** for an index is 4, and **number\_of\_replicas** is 2, how many total shards will exist for this index?
5. If you have three master-eligible nodes in your cluster, what should you set **minimum\_master\_nodes** to?



# Lab 6

## The Distributed Model

Yasaswy Raval - 24-Apr-2017 - Capital One

## Chapter 7

# Working with Search Results

Yasaswy Raval - 24-Apr-2017 - Capital One

- 1 Introduction to Elasticsearch
- 2 The Search API
- 3 Text Analysis
- 4 Mappings
- 5 More Search Features
- 6 The Distributed Model
- 7 Working with Search Results
- 8 Aggregations
- 9 More Aggregations
- 10 Handling Relationships

# Topics covered:

- The Anatomy of a Search
- Boosting Relevance
- DFS Query-then-fetch
- Sorting Results
- Doc Values and Fielddata
- Pagination
- Scroll Searches
- Choosing a Search Type

Yasaswy Raveendran - 24-Apr-2017 - Capital One

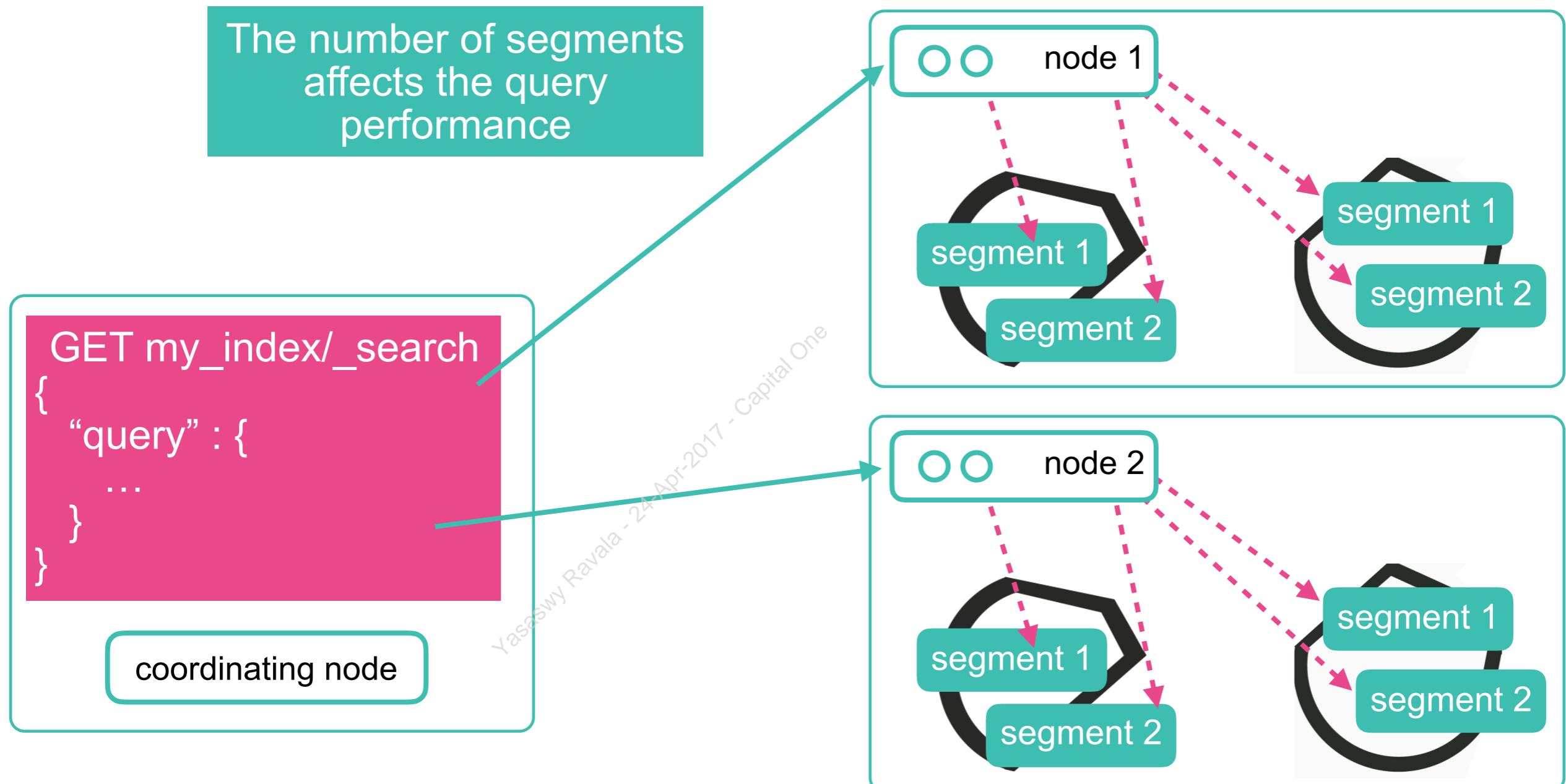


# The Anatomy of a Search

Yasaswy Raval - 24-Apr-2017 - Capital One

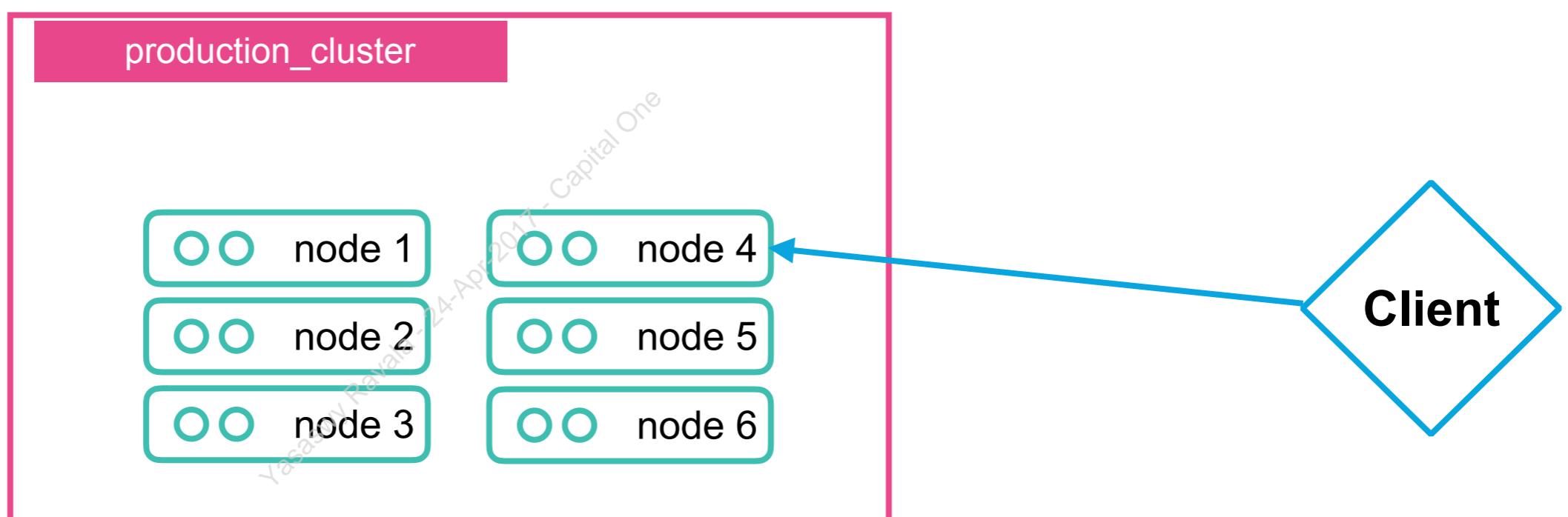
# Searches are performed on segments

- When you submit a query to Elasticsearch, Lucene runs the search on **every segment of the index**



# The Anatomy of a Search

- When a client makes a request, the node handling that request is referred to as the *coordinating node*
  - Any good client will “round robin” search requests to different coordinating nodes to avoid overloading a single node



# The Phases of a Search

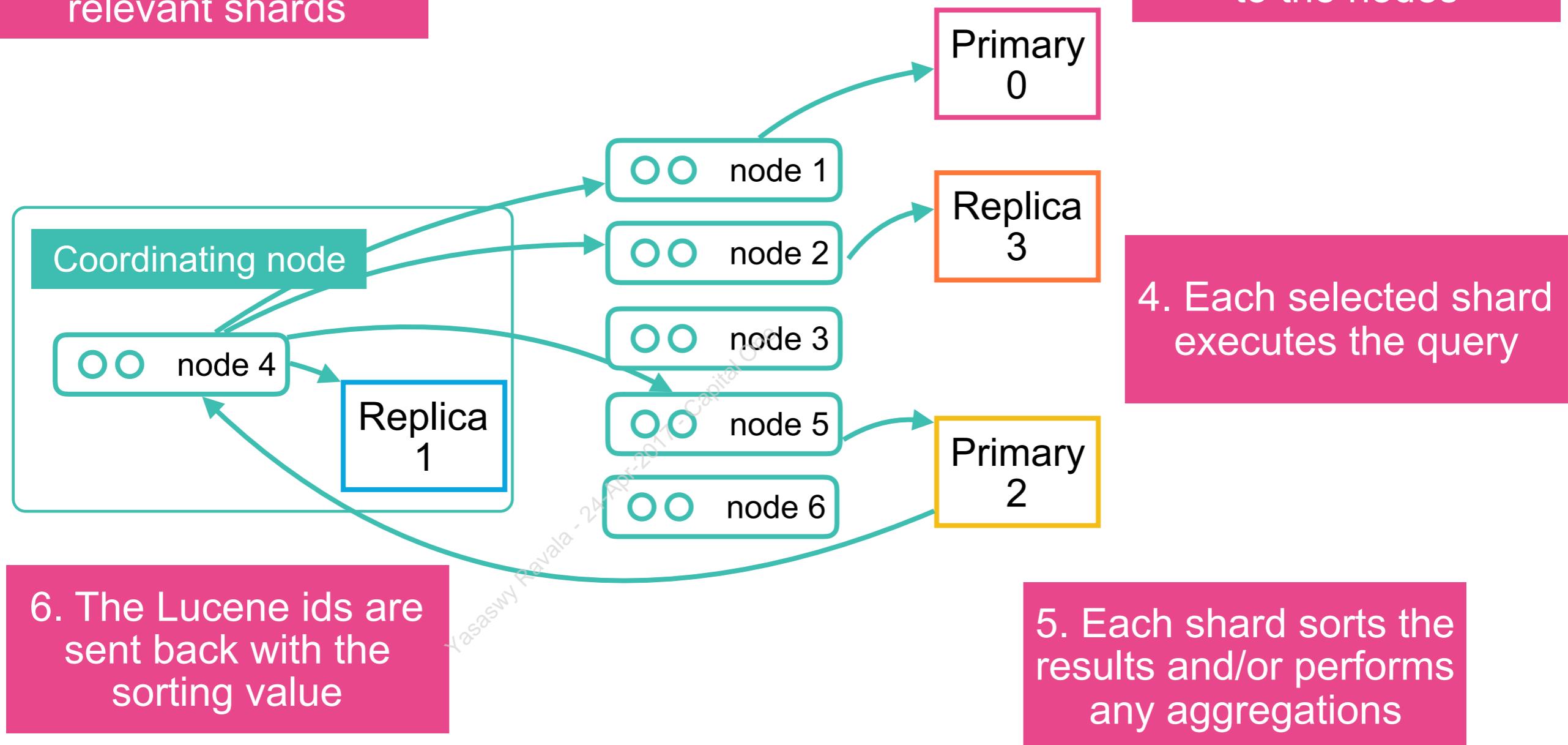
- The coordinating node is responsible for making sure a request is routed to the proper nodes
  - results from multiple shards must be combined into a single sorted list
  - then the coordinating node returns a “page” of results
- In a search request, the coordinating node also coordinates the two phases of the search:
  1. *Query phase*
  2. *Fetch phase*

# 1. Query Phase

1. Determine the relevant shards

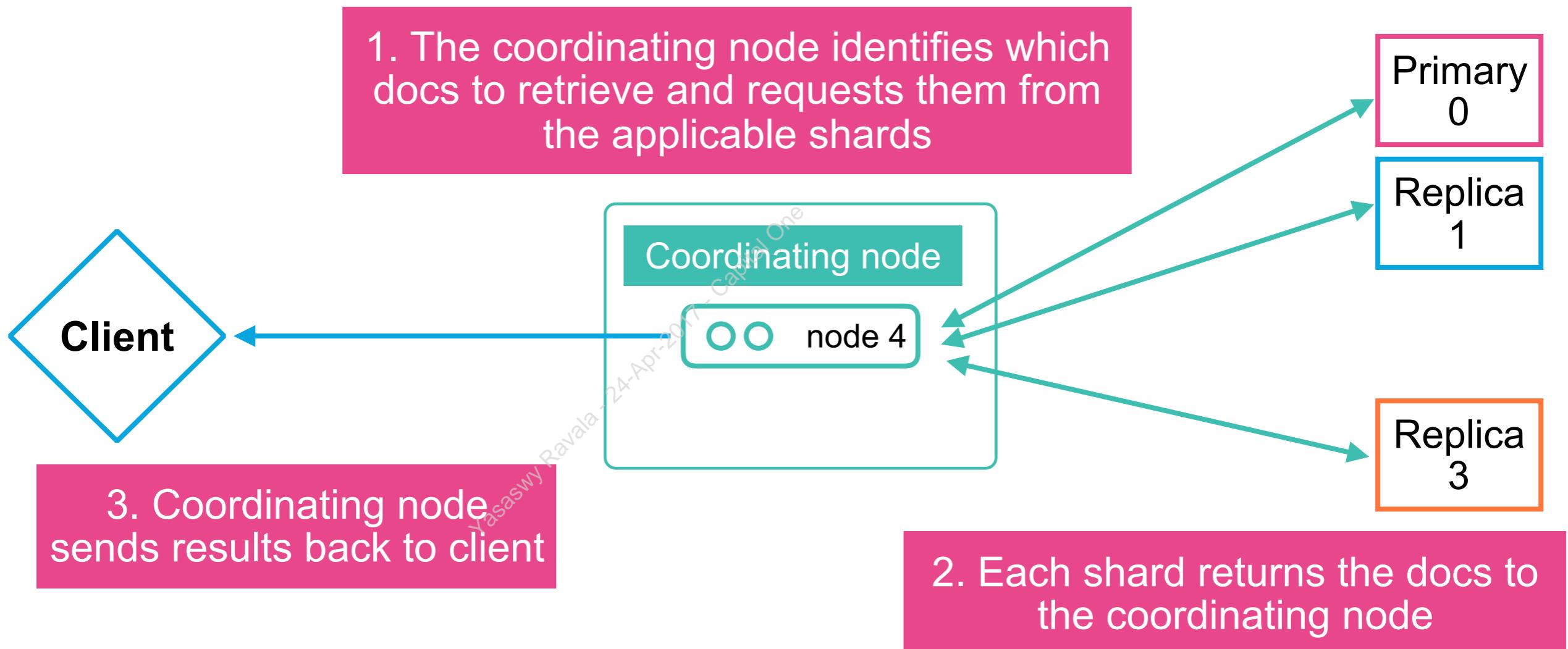
2. Choose shard copy (trying to load balance)

3. Forward the request to the nodes



## 2. The Fetch Phase

- The query phase identifies which documents satisfy the search request
- The *fetch phase* retrieves the documents:



# The Search Response

- The `_shards` section of a search response provides how many shards were searched
  - as well as a count of the successful/failed searched shards
  - if `successful < total` you likely received only partial results
  - your business logic and SLAs typically determine how to handle errors and/or incomplete results

```
{  
  "took": 11,  
  "timed_out": false,  
  "_shards": {  
    "total": 3,  
    "successful": 3,  
    "failed": 0  
  },  
  "hits": {
```

If “`successful`” is less than “`total`”, you likely have incomplete results

# Boosting Relevance

Yasaswy Raval - 24-Apr-2017 - Capital One

# Boosting Relevance

- A search for “**elastic**” will likely return a variety of different documents
  - From Elastic (the company) to stretchable fabric, rubber bands, kinetic energy, and even the name of a jazz album
- One way to control the relevance of a search is by ***boosting*** query clauses

Yasaswy Raval - 24-Apr-2017 - Capital One



# Query-time Boosting

- Some types of searches allow a “**boost**” parameter to increase (or decrease) the relative weight of a clause
- If **boost** > 1
  - the score is *increased*
- If 0 < **boost** < 1
  - the score is *decreased*
- The score is *not* multiplied by the boost
  - Instead, the new score is normalized after the boost is applied
  - Each type of query has its own normalization algorithm



# Query-time Boosting

- Add the “**boost**” property to a query clause to manipulate the score of matched document

```
GET my_index/my_type/_search
{
  "query" : {
    "bool": {
      "should": [
        {
          "match": {
            "comment": "star"
          }
        },
        {
          "match" : {
            "comment" : {
              "query": "house",
              "boost" : 2
            }
          }
        }
      ]
    }
  }
}
```

*I am  
searching for “star”  
and “house”, but with  
an emphasis on  
“house”*

Documents that match “house”  
will get a boost of factor 2

# Customizing Scoring

- Elasticsearch allows you to modify the score of documents using your own logic
  - customize the scoring in a Painless script (or other language)

```
GET messages/_search
{
  "query": {
    "function_score": {
      "functions": [
        {
          "script_score": {
            "script": {
              "lang" : "painless",
              "inline" : """
                _score * 0.05 * doc['details.plus_ones'].value + 1
              """
            }
          }
        }
      ]
    }
  }
}
```

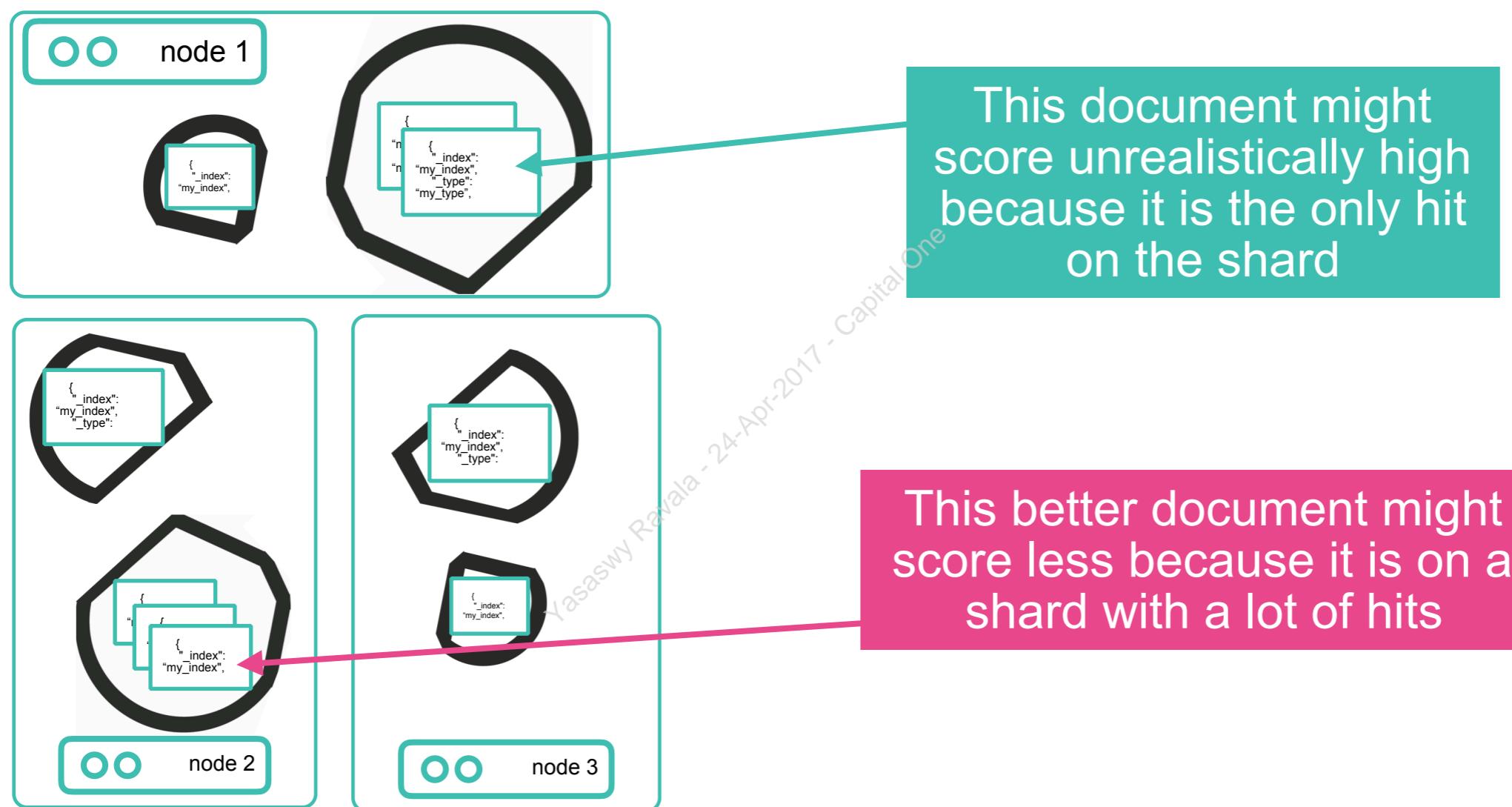
We discuss  
**function\_score** and  
scripting in our Advanced  
Developer course

# DFS Query-then-fetch

Yasaswy Raval - 24-Apr-2017 - Capital One

# Relevance and Sharding

- How is sharding related to the `_score` and search phases?
  - It is partially calculated at the shard level
  - For smaller or skewed datasets, this can drastically affect the scoring



# Query-then-fetch vs. DFS Query-then-fetch

- If the default **query-then-fetch** scoring is not working for you:
  - which uses local BM25 scores
- Then you can try executing a ***DFS query-then-fetch***
  - which computes a global BM25 score
  - more expensive, because it involves running a pre-query
- Tends to fix the issue of skewed datasets
  - For example, time-based indices may create skewed results. On the day a new iPhone is released, tweets containing #iphone6 will be skewed for that particular day
  - Smaller datasets are also more prone to being skewed



# Performing a DFS Query-then-fetch

- Set the `search_type` property to `dfs_query_then_fetch`
  - Defaults to `query_then_fetch`

```
GET my_index/_search
{
  "query" : {
    "match": {
      "comment": "star"
    }
  }
}
```

query\_then\_fetch

```
GET my_index/_search?search_type=dfs_query_then_fetch
{
  "query" : {
    "match": {
      "comment": "star"
    }
  }
}
```

dfs\_query\_then\_fetch

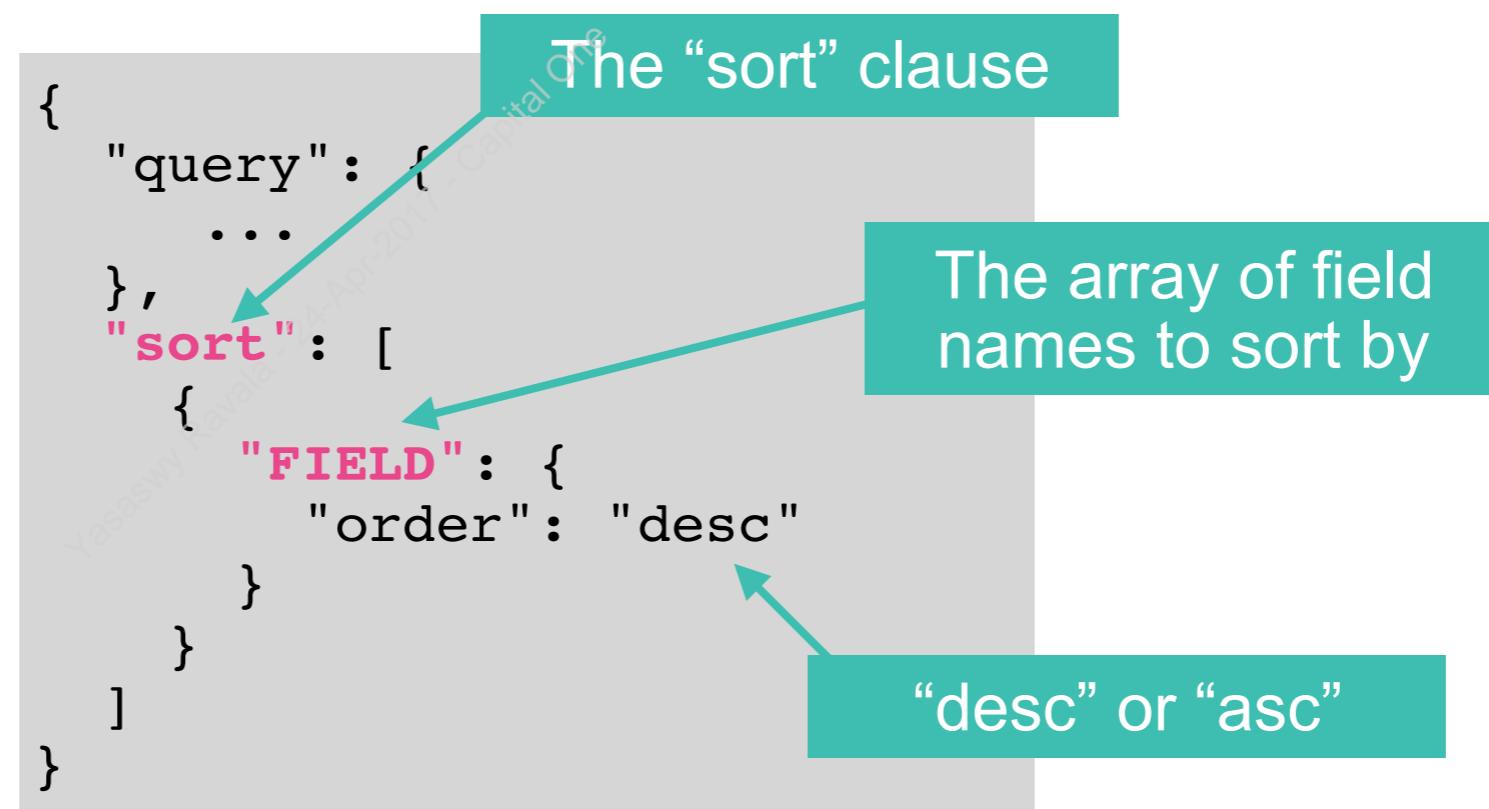


# Sorting Results

Yasaswy Raval - 24-Apr-2017 - Capital One

# Sorting

- Sorting the results of a query can be done three ways:
  - Sort by a field
  - Sort by `_score`
  - Sort by `_doc`
- The syntax looks like:



The diagram illustrates the syntax for the "sort" clause in an Elasticsearch query. It shows a JSON-like structure with the following components:

- The "sort" clause:** A green box labeled "The 'sort' clause" points to the `"sort": [` part of the code.
- The array of field names to sort by:** A green box labeled "The array of field names to sort by" points to the `[` and `]` brackets around the sorting fields.
- "desc" or "asc":** A green box labeled "'desc' or 'asc'" points to the `"order": "desc"` part of the sorting field definition.

```
{  
  "query": {  
    ...  
  },  
  "sort": [  
    {  
      "FIELD": {  
        "order": "desc"  
      }  
    }  
  ]  
}
```

A watermark with the text "Yasasvi Capital BV" is visible diagonally across the background of the code snippet.

# A sort Example

- The following query returns products with “chocolate” in the “ingredients” sorted by “total\_fat” descending:

*Which items with chocolate have the most fat?*

```
GET nutrition/_search
{
  "query": {
    "match": {
      "ingredients": "chocolate"
    }
  },
  "sort": [
    {
      "total_fat": {
        "order": "desc"
      }
    }
  ]
}
```

```
"hits": {
  "total": 15,
  "max_score": null,
  "hits": [
    {
      "_index": "nutrition",
      "_type": "nutrition_type",
      "_id": "AVhMPbOveR2NtF_alufM",
      "_score": null,
      "_source": {
        "total_fat": 18,
        "calories_from_fat": 170,
        "brand_name": "ProBar",
        "item_name": "Bar, Koka Moka",
        "calories": 360,
        "servings_per_container": 1,
        "saturated_fat": 5,
      }
    }
  ]
}
```

# Sort on Multiple Fields

```
GET nutrition/_search
{
  "query": {
    "match": {
      "ingredients": "chocolate"
    }
  },
  "sort": [
    {"servings_per_container": {"order": "desc"}},
    {"_score": {"order": "desc"}}
  ]
}
```

Sort by “servings\_per\_container”,  
then by “\_score”

Yasaswy Raval - 24-Apr-2017

```
"_score": 2.1980286,
"_source": {
  "total_fat": 7,
  "servings_per_container": 13.5,
  "brand_name": "Pepperidge Farm",
  "item_name": "Cookie Collection, Holiday Homestyle",
  "calories": 160,
  "serving_size_qty": 3
},
"sort": [
  13.5,
  3.3508143
],
},
```

Notice the values used for sorting  
are provided in the results

# Simpler Syntax for Sort

- If you want to sort on a value descending, you can simply put the value in the “sort” array
  - This query is identical to the one on the previous slide

```
GET nutrition/_search
{
  "query" : {
    "match": {
      "ingredients": "chocolate"
    }
  },
  "sort": [
    {"servings_per_container": {"order": "desc"}},
    "_score"
  ]
}
```

Shortcut for sorting by `_score` descending

# Sort by `_doc`

- Sorting by `_doc` provides the best performance and least overhead
  - for situations where you do not care about the sort order

```
GET nutrition/_search
{
  "query" : {
    "match": {
      "ingredients": "chocolate"
    }
  },
  "sort": [
    "_doc"
  ]
}
```

Documents are sorted in the order that Lucene has them indexed

# Sorting by Text

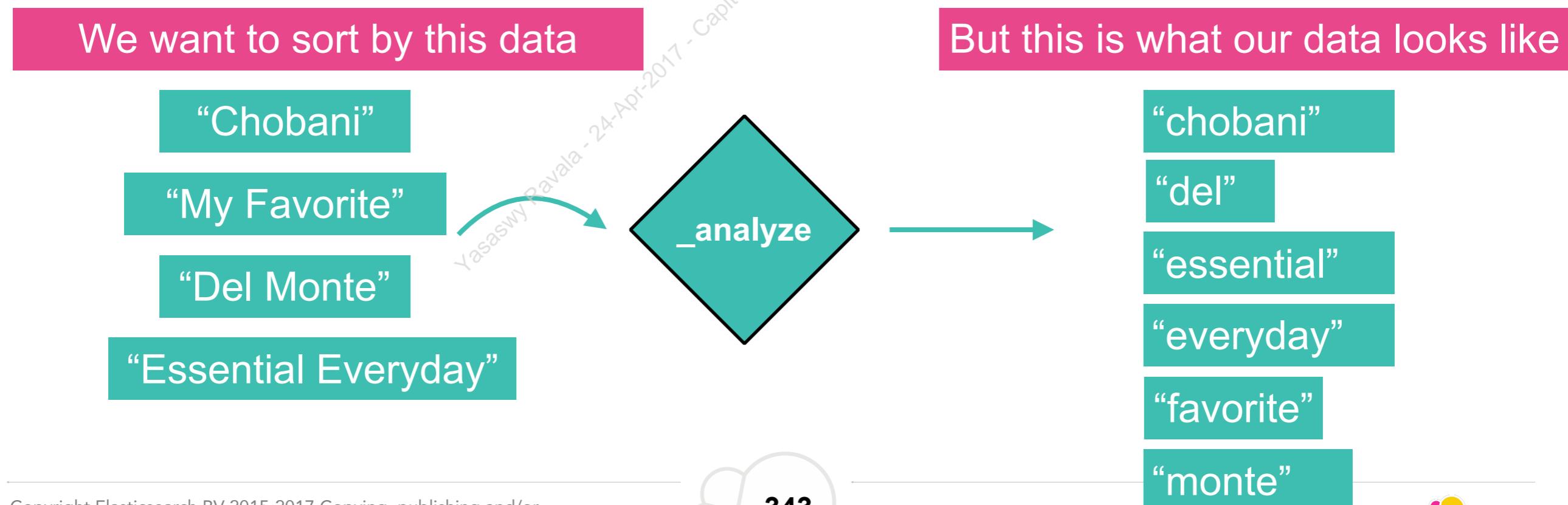
- Look at the following query. It seems very simple. What will the results look like?

```
GET nutrition/_search
{
  "query": {
    "match": {
      "item_name": "yogurt"
    }
  },
  "sort": [
    {"brand_name": {"order": "asc"}}
  ]
}
```

# Why can't we sort by text?

```
"type": "illegal_argument_exception",
"reason": "Fielddata is disabled on text fields by default. Set
fielddata=true on [brand_name] in order to load fielddata in memory
by uninverting the inverted index. Note that this can however use
significant memory."
```

- You **can** sort by text, but you need to understand the cost
  - The issue is that **text** data types are **analyzed**
  - The original value of the text has been changed for indexing



# Doc Values and Fielddata

Yasaswy Raval - 24-Apr-2017, Capital One

# Introducing Doc Values and Fielddata

- An inverted index is not ideal for sorting on field values
  - sorting (and aggregations) require us to *uninvert* the inverted index
- Two options in Elasticsearch for building this uninverted index:
  - **doc values**
  - **fielddata**
- **doc values** only work on exact value fields (so *NOT* on analyzed fields)
- We need to enable **fielddata** on a tokenized field if we want to sort by it



# Doc Values vs. Fielddata

- Which option is better? Well, it depends...

	Fielddata	Doc Values
When	Created on the fly when needed in a <code>_search</code>	Created at index time
Where	JVM heap	files on disk
Pros	faster indexing and avoids disk usage	avoids out-of-memory errors
Cons	large heap usage	extra disk usage, slightly slower indexing
Default	Elasticsearch 1.x and prior	Elasticsearch 2.x and later



# Why change the defaults?

- Save some disk space and slightly reduce index time:
  - disable `doc_values` in fields you **will not** sort or aggregate on
  - set **doc\_values: false** in the mappings of the field(s)
  - **Be careful:** if later on you want to sort or aggregate on that field you will need to update the mapping and re-index your documents
- You really need to sort or aggregate on a text field:
  - enable `fielddata` on that field (e.g. significant terms)
  - set **fielddata: true** in the mappings of the field(s)
  - **Be careful:** fielddata can cause out-of-memory errors



# Sorting by Strings

Yasaswy Raval - 24-Apr-2017 - Capital One

# So how do we sort by strings?

- One option is to use the “**keyword**” type:
  - in our case, “**brand\_name**” has a “**keyword**” multi-field

```
GET nutrition/_search
{
  "query": {
    "match": {
      "item_name": "yogurt"
    }
  },
  "sort": [
    {
      "brand_name.keyword": {
        "order": "asc"
      }
    }
  ]
}
```

Sorting is OK here because the “**keyword**” multi-field of “**brand\_name**” has doc values



“Chobani”
“Chobani Greek Yogurt”
“Crowley”
“Dannon”
...and so on

“**brand\_name.keyword**” works well here because all of our brand names start with an upper-case letter



# Configure a field just for sorting

- Another option is to add a field to the mapping that is used just for sorting
  - useful if our text starts with upper and lower case

Our sort field uses a custom analyzer that lowerscases the string

Add a **brand\_name** field just for sorting purposes

It is a “**text**” field, so we need to enable **fielddata** if we want to sort on it

```
PUT nutrition2
{
  "settings" : {
    "analysis" : {
      "analyzer" : {
        "my_sort_analyzer" : {
          "tokenizer" : "keyword",
          "filter" : [ "lowercase" ]
        }
      }
    }
  },
  "mappings" : {
    "nutrition_type" : {
      "properties" : {
        "brand_name" : {
          "type" : "text",
          "fields" : {
            "keyword" : {
              "type" : "keyword",
              "ignore_above" : 256
            }
          }
        }
      }
    }
  }
}

"my_sort" : {
  "type" : "text",
  "analyzer" : "my_sort_analyzer",
  "fielddata" : true
}
```

# Example using our sort field:

```
GET nutrition2/_search
{
  "query": {
    "match": {
      "item_name": "cheese"
    }
  },
  "sort": [
    {
      "brand_name.my_sort": {
        "order": "asc"
      }
    }
  ]
}
```

Sort by our added sort field

del Monte

Nabisco

Notice that sorting by **“brand\_name.keyword”** would have swapped the order of these two brand names.

# Pagination

Yasaswy Raval - 24-Apr-2017 - Capital One

# Pagination

- A very common use case of searching is ***pagination***
  - dividing search results into “pages”, or subsets
- A query clause uses the “**from**” and “**size**” parameters to implement pagination:

```
GET nutrition/_search
{
  "from": 0,
  "size": 10,
  "query": {
    "match": {
      "ingredients" : "onion"
    }
  },
  "sort" : [
    {"calories" : {"order" : "desc"}},
    "_score"
  ]
}
```

Return the first 10 hits from the given query



# Retrieving Subsequent Pages

- You have some work to do in your client application to determine which page to request
  - But that is common in any paging use case
  - Some clients will have an API to help simplify page requests

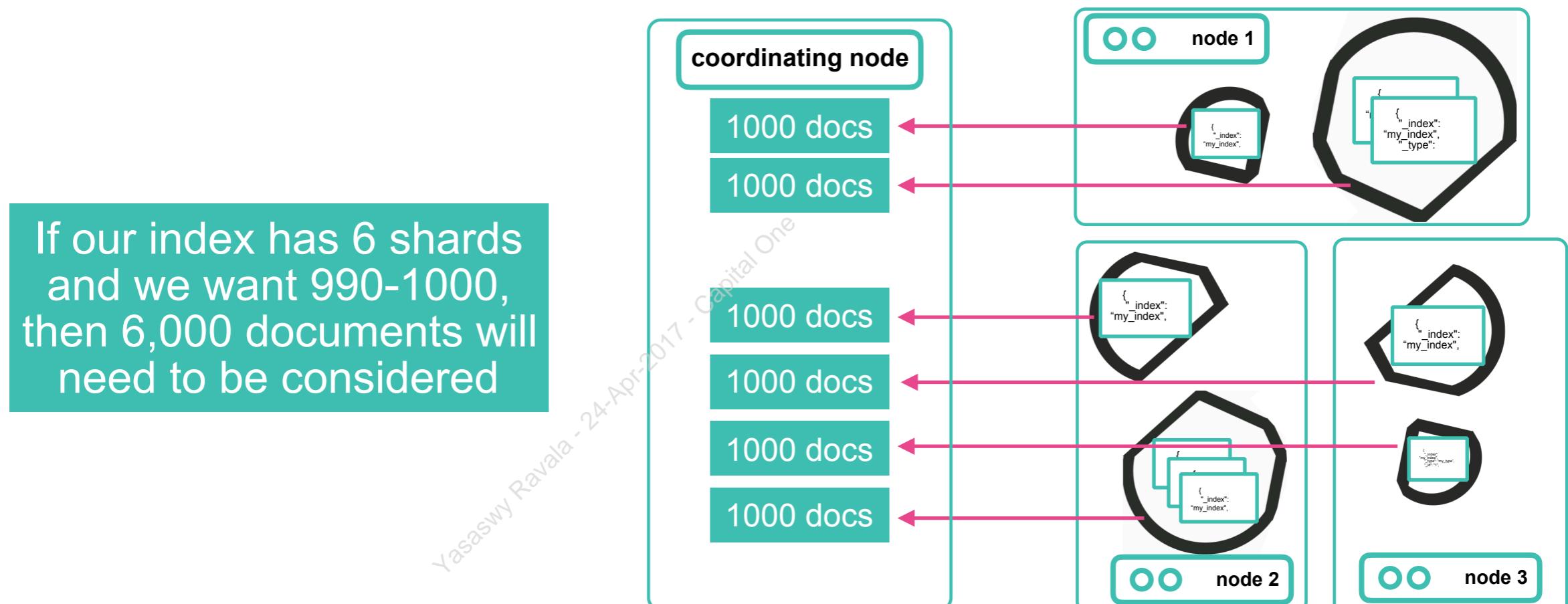
```
GET nutrition/_search
{
  "from": 10, // Circled in green
  "size": 10,
  "query": {
    "match": {
      "ingredients" : "onion"
    }
  },
  "sort" : [
    {"calories" : {"order" : "desc"}},
    "_score"
  ]
}
```

Returns the next 10 hits  
in the result set



# Be Careful with Deep Pagination

- Suppose you want hits from 990 to 1,000. That requires a 1,000 documents from each shard to be considered!
  - Deep pagination should be avoided*



**from + size** can not be more than **index.max\_result\_window**  
which defaults to 10,000

# The search\_after Parameter

- If you have a lot of pages, a better option for retrieving subsequent pages is to use the **search\_after** parameter:
  - You specify the last sort values from the previous result set:

```
GET nutrition/_search
{
  "size": 10,
  "query": {
    "match": {
      "ingredients" : "onion"
    }
  },
  "search_after" : [270, 0.93839806, "nutrition_type#AVhRCa"],
  "sort" : [
    {"calories" : {"order" : "desc"}},
    "_score",
    "_uid"
  ]
}
```

The previous hit was 270 calories  
and \_score 0.93839806



# Scroll Searches

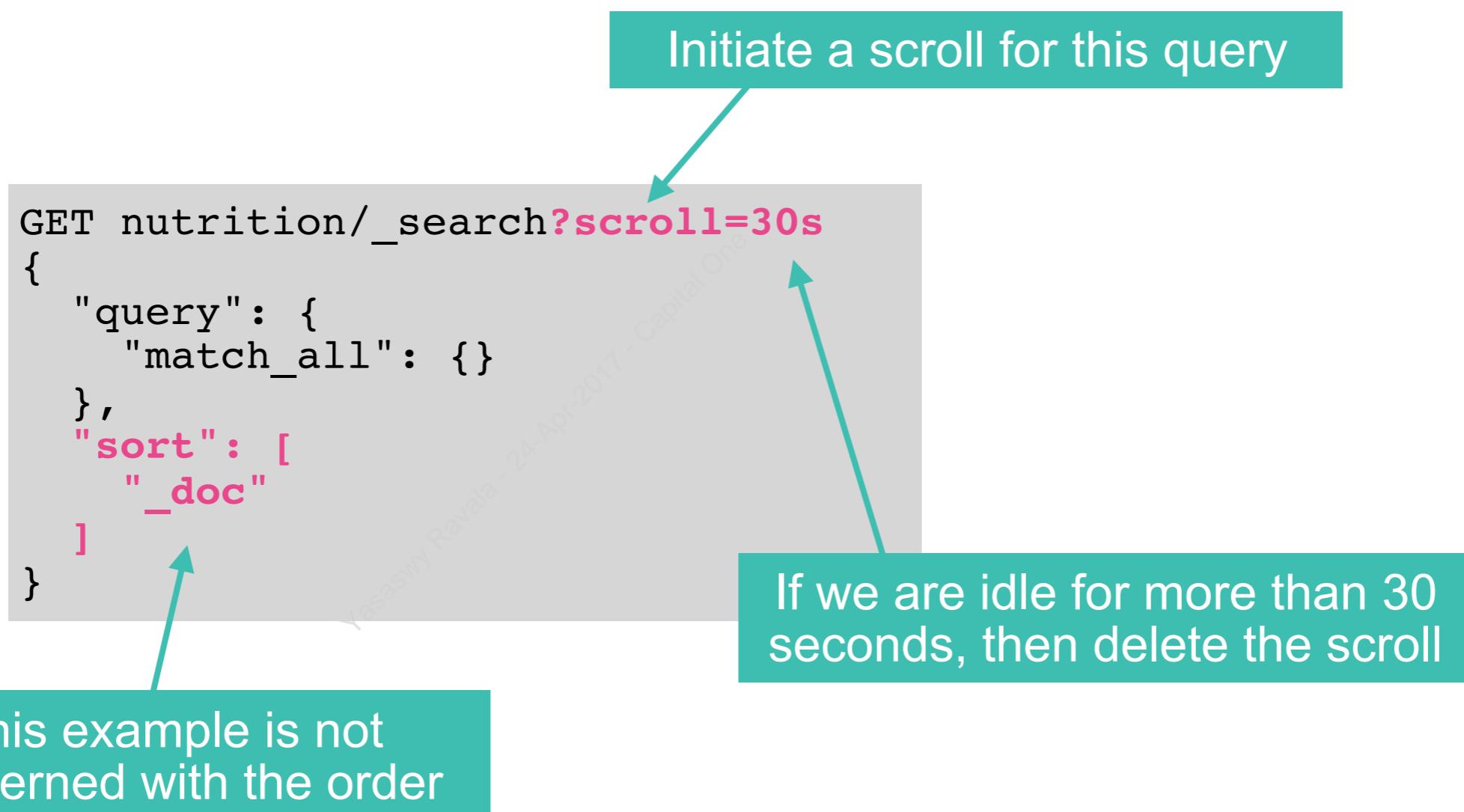
Yasaswy Raval - 24-Apr-2017 - Capital One

# Scroll Searches

- What if you want more than a “page” of results?
  - Large result sets can be expensive to both the cluster and the client
- The **scroll API** allows you to take a snapshot of a large number of results from a single search request
  - Notice the word “***snapshot***”
  - When you perform a scroll search, any changes made to documents after the scroll is initiated will not show up in your results
- Scroll searches are a 4-step process:

# 1. Initiate the scroll

- To initiate a scroll search, add the scroll parameter to your search query
  - Specifying how long Elasticsearch should keep the search context alive



## 2. Get the current \_scroll\_id

- The response will contain the first 100 hits, and
- a \_scroll\_id
  - Hang on to it! You need it for the next request

```
{  
  "_scroll_id":  
    "DnF1ZXJ5VGh1bkZldGNoCgAAAAAAAAAf1FnZVWmtsa2VKUdTVnJCRkFiZ3RjZ2  
    CAAAAAAAH5xZ2VVprbGt1SlFHU1ZyQkZBYmd0Y2dnAAAAAAAAB-  
    YWd1Vaa2xrZUpRR1NWckJGQWJndGNnZwAAAAAAAfoFnZVWmtsa2VKUdTVnJCR  
    kFiZ3RjZ2cAAAAAAAH6RZ2VVprbGt1SlFHU1ZyQkZBYmd0Y2dnAAAAAAAAB--0  
    Wd1Vaa2xrZUpRR1NWckJGQWJndGNnZwAAAAAAAfuFnZVWmtsa2VKUdTVnJCRk  
    FiZ3RjZ2c=",  
  "took": 2,  
  "timed_out": false,  
  "_shards": {  
    "total": 10,  
    "successful": 10,  
    "failed": 0  
  },  
  ...  
}
```

### 3. Retrieve more documents

- When you are ready for more documents, you just need to send a GET request, along with the prior `_scroll_id`

Do not specify an index, type or query here -  
just invoke **scroll**

```
GET _search/scroll
{
  "scroll" : "30s",
  "scroll_id" :
  "DnF1ZXJ5VGh1bkZldGNoCgAAAAAAAAlFnZVWmtsa2VKUUdTVnJCRkFiZ3RjZ2cAA
  AAAAAAH5xZ2VVprbGt1S1FHU1ZyQkZBYmd0Y2dnAAAAAAAAB-
  YWdlVaa2xrZUpRR1NWckJGQWJndGNnZwAAAAAAAfoFnZVWmtsa2VKUUdTVnJCRkFi
  Z3RjZ2cAAAAAAAHH6RZ2VVprbGt1S1FHU1ZyQkZBYmd0Y2dnAAAAAAAAB-
  oWdlVaa2xrZUpRR1NWckJGQWJndGNnZwAAAAAAAfrFnZVWmtsa2VKUUdTVnJCRkFi
  Z3RjZ2cAAAAAAAHH7BZ2VVprbGt1S1FHU1ZyQkZBYmd0Y2dnAAAAAAAAB-0WdlVaa2
  xrZUpRR1NWckJGQWJndGNnZwAAAAAAAfuFnZVWmtsa2VKUUdTVnJCRkFiZ3RjZ2c=
  "
}
```

Always use the prior `_scroll_id`

If the scroll is still alive, the next  
set of hits is returned

## 4. Clearing a scroll

- If you are done using a scroll and want to manually close it, use a **DELETE** request:

```
DELETE _search/scroll/DnF1ZXJ5VGh1bkZ1dGNoCgAAAAAAAf1FnZVWm  
tsa2VKUUDTVnJCRkFiZ3RjZ2cAAAAAAAH5xZ2VVprbGt1S1FHU1ZyQkZBYmd0Y2dnA  
AAAAAAABYwd1Vaa2xrZUpRR1NWckJGQWJndGNnZwAAAAAAfoFnZVWmtsa2VKUUDTV  
nJCRkFiZ3RjZ2cAAAAAAAH6RZ2VVprbGt1S1FHU1ZyQkZBYmd0Y2dnAAAAAAAB-  
oWd1Vaa2xrZUpRR1NWckJGQWJndGNnZwAAAAAAfrFnZVWmtsa2VKUUDTVnJCRkFiZ  
3RjZ2cAAAAAAAH7BZ2VVprbGt1S1FHU1ZyQkZBYmd0Y2dnAAAAAAAB-oWd1Vaa2xr  
ZUpRR1NWckJGQWJndGNnZwAAAAAAfuFnZVWmtsa2VKUUDTVnJCRkFiZ3RjZ2c=
```

# Choosing a Search Type

Yasaswy Raval - 24-Apr-2011, Capital One

# Choosing a Search Type

- We have seen several different types of searches:
  - **regular** searches: the typical “**query**” searches that we have been running throughout the course
  - **scroll** searches: a snapshot of a (possibly) large result set of documents
  - **pagination** searches: a small subset of a larger result set of documents

Yasaswy Raval - 24-Apr-2017 - Capital One

# Real-world Use Cases

- **Regular search:**
  - Suppose we are pulling back search results from an auction website like Ebay and sorting by the date the auction is ending. We want auctions that are ending the soonest at the top. We would need to run a new query each time.
- **Scroll search:**
  - Imagine you are implementing an “export your data” feature on a site like Facebook. A snapshot of the data works best for a large number of hits that we want to return in small batches.
- **Pagination search:**
  - If you're implementing classic pagination on a website, then **from** and **size** are likely your “go to” option
  - Use **search\_after** if you need deep pagination

# Chapter Review

Yasaswy Raval - 24-Apr-2017 - Capital One

# Summary

- Search queries typically have two phases: **query** and **fetch**
- **Relevance** refers to the **scoring of a document** based on how closely it matches the query
- Use the **sort** clause for sorting by a field, **\_score** or **\_doc**
- **Doc values** store your original fields' values on disk
- Use “**from**” and “**size**” parameters to implement pagination
- The **scroll API** allows you to take a snapshot of a large number of results from a single search request

# Quiz

1. Give two use cases for enabling **dfs\_query\_then\_fetch** on a query.
2. **True or False:** You can sort the results of a query by a field of type “**text**”.
3. **True or False:** You can sort the results of a query by a field of type “**keyword**”.
4. How are hits sorted by default?
5. What two parameters are used to implement paging of search results?
6. What is the difference between a **scroll search** and a **pagination search**?



# Lab 7

## Working with Search Results

Yasaswy Raval - 24-Apr-2017 - Capital One

# Chapter 8

# Aggregations

Yasaswy Raval - 24-Apr-2017 - Capital One

- 1 Introduction to Elasticsearch
- 2 The Search API
- 3 Text Analysis
- 4 Mappings
- 5 More Search Features
- 6 The Distributed Model
- 7 Working with Search Results
- 8 Aggregations**
- 9 More Aggregations
- 10 Handling Relationships

# Topics covered:

- What are Aggregations?
- Types of Aggregations
- Buckets and Metrics
- Common Metrics Aggregations
- The range Aggregation
- The date\_range Aggregation
- The terms Aggregation
- Nesting Buckets

# What are Aggregations?

Yasaswy Raval - 24-Apr-2017 Capital One

# What is an Aggregation?

- We have been focusing on search, but Elasticsearch has another powerful capability known as aggregations
- **Aggregations** are a way to perform analytics on your indexed data

*What movies have “star” in the title?*

*That  
is a search  
query...*

*What  
is the average ticket price  
of a movie in the U.S?*

*That  
is an aggregation...*



# You ask different questions with aggs:

- In **search**, we have been asking for a subset of the original dataset:
  - Results are **hits** that match our search parameters

“What are the ERRORS in our log file?”

- In **aggregations**, we analyze the data by asking questions about the dataset:
  - Results are (typically) **computed values**

“How many 404 errors occurred each day last month?”

“What is the average amount of time visitors spend on our home page?”

“What is the most visited page on our website?”



Search criteria

**Search**

Destination/Hotel Name:  
San Francisco

(Work) Leisure

Check-in  
Check-in Date

Check-out  
Check-out Date

Rooms: 1  
Adults: 2  
Children: 0

**Search**

Aggregations

Filter by:	
▼ Star Rating	
<input type="checkbox"/> No preference	454
<input type="checkbox"/> 1 star	16
<input type="checkbox"/> 2 stars	80
<input type="checkbox"/> 3 stars	72
<input type="checkbox"/> 4 stars	55
<input type="checkbox"/> 5 stars	9
<input type="checkbox"/> Unrated	222
▼ Accommodation Type	
<input type="checkbox"/> Apartments	186
<input type="checkbox"/> Hotels	175
<input type="checkbox"/> Motels	39
<input type="checkbox"/> Vacation Homes	21
<input type="checkbox"/> Hostels	18
<input type="checkbox"/> Inns	9
<input type="checkbox"/> Bed and Breakfasts	6

## San Francisco: 454 properties found

3 Reasons to Visit: Golden Gate Bridge, Fisherman's Wharf & cable cars

- [Check out the best of San Francisco before you find a place to stay...](#)
- Lock in a great price for your stay on these dates:
- |                 |                 |                 |                 |
|-----------------|-----------------|-----------------|-----------------|
| Jul 27 — Jul 28 | Jul 28 — Jul 29 | Jul 29 — Jul 30 | Jul 30 — Jul 31 |
|-----------------|-----------------|-----------------|-----------------|
- Our Top Picks First Stars Distance From Downtown Review Score
- 
**Hotel Nikko San Francisco** ★★★★ 
  
Union Square, San Francisco – Subway Access

Just a 5-minute walk from Union Square, Hotel Nikko San Francisco features an on-site restaurant. Pillow-top bedding and a flat-screen TV are provided in all rooms.

Booked 62 times today

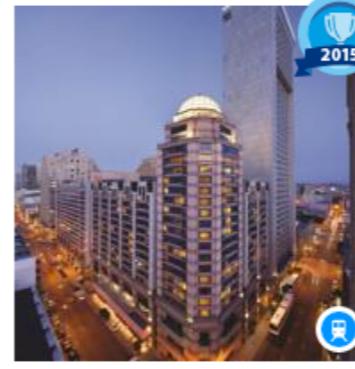
**Excellent 8.6**  
3,706 reviews

[Show prices](#)
- 
**Hotel Zephyr San Francisco** ★★★★ 
  
Fisherman's Wharf, San Francisco

Offering a fitness center, Hotel Zephyr San Francisco is located a short 330 yards from Pier 39 Fisherman's Wharf. Free WiFi access is available in this waterfront hotel.

Booked 65 times today

**Very good 8.3**  
WiFi 8.9  
2,747 reviews

[Show prices](#)
- 
**Hilton San Francisco Union Square** ★★★★ 
  
Union Square, San Francisco – Subway Access

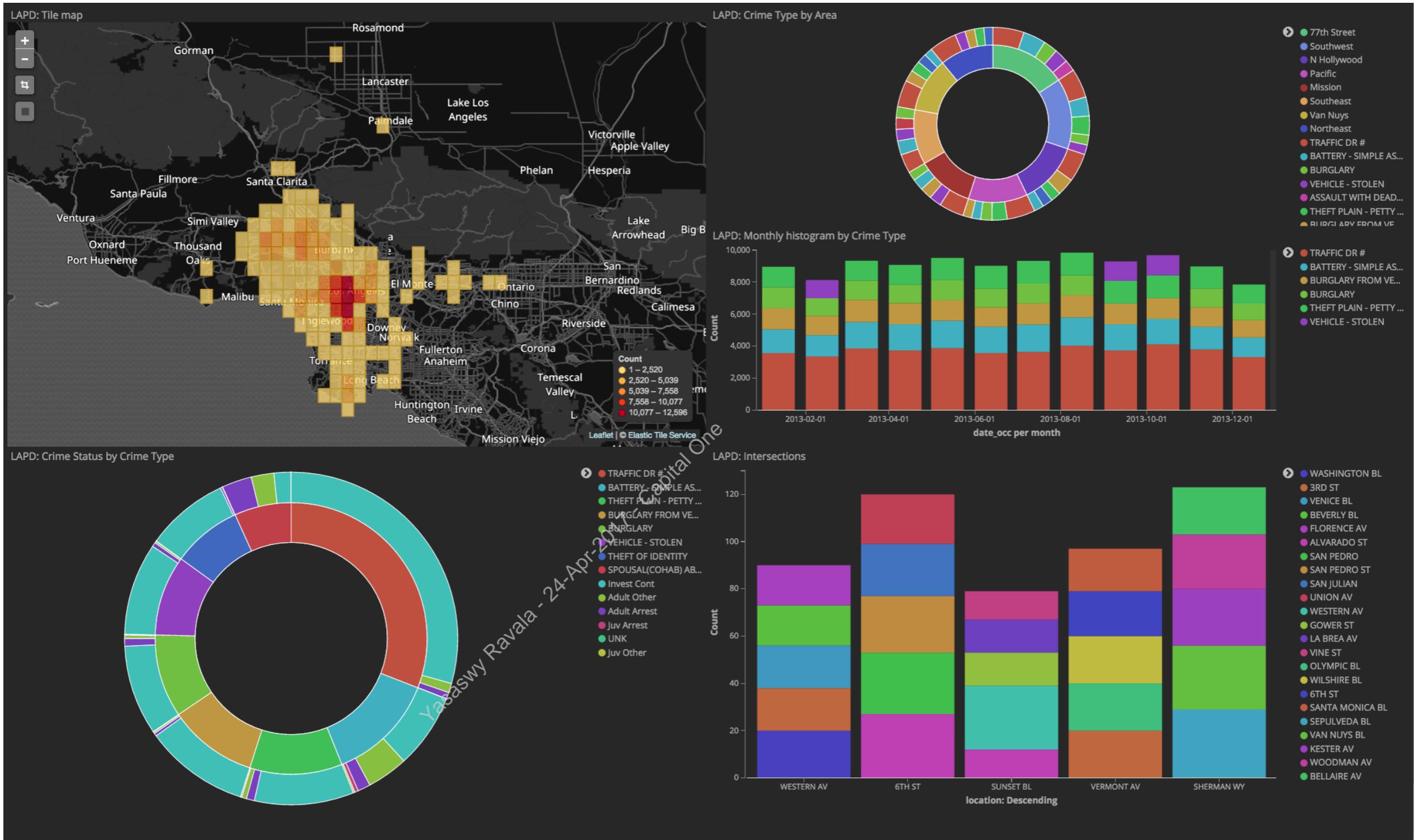
Located in the heart of downtown San Francisco and just a 5-minute walk from the Powell Street subway station, this Hilton features an on-site Herb n' Kitchen and a beautiful courtyard pool.

Booked 25 times today

**Good 7.6**  
3,817 reviews

[Show prices](#)

# Kibana Visualizations



# Aggregation Syntax

- An aggregation request is a part of the Search API
  - With or without a “query” clause

The “**aggs**” clause can be spelled out “**aggregations**”

```
GET my_index/_search
{
  "aggs": {
    "my_aggregation": {
      "AGG_TYPE": {
        ...
      }
    }
  }
}
```

The name you choose comes back in the results

Lots of different aggregation types



# Example of an Aggregation

*What  
is the average  
number of calories of items  
with “olive oil” in the  
ingredients?*

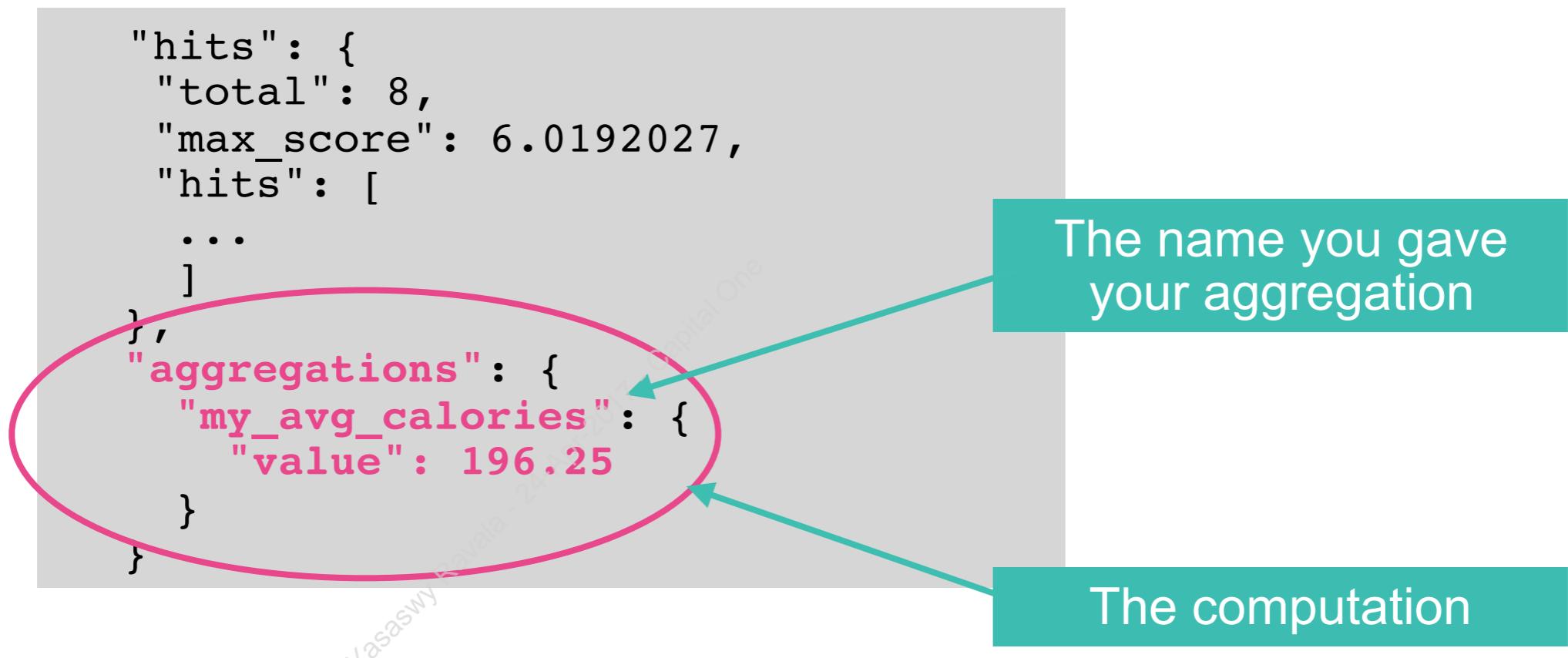
```
GET nutrition/_search
{
  "query" : {
    "match_phrase": {
      "ingredients": "olive oil"
    }
  },
  "aggs": {
    "my_avg_calories": {
      "avg": {
        "field": "calories"
      }
    }
  }
}
```

The “query” defines the  
scope of the aggregation

The “aggs” section returns an  
average

# Viewing the Results

- The results will have an “**aggregations**” section with the results of all the “**aggs**” in your search



# If you just want the aggregation value:

- If you are not interested in the hits and only want the values of the aggregations, then set “**size**” to 0
  - This will speed up the query since the “fetch” phase can be skipped

```
GET nutrition/_search
{
  "size": 0,
  "query" : {
    "match_phrase": {
      "ingredients": "olive oil"
    }
  },
  "aggs": {
    "my_avg_calories": {
      "avg": {
        "field": "calories"
      }
    }
  }
}
```



```
{
  "took": 25,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 8,
    "max_score": 0,
    "hits": []
  },
  "aggregations": {
    "my_avg_calories": {
      "value": 196.25
    }
  }
}
```



# Types of Aggregations

Yasaswy Raval - 24-Apr-2017 - Capital One

# Four Types of Aggregations

- **Bucket**: aggregations that combine documents that meet a given criteria into buckets
  - Bucket aggregations build buckets
- **Metric**: mathematical calculations performed on the fields of documents
  - Typically, metrics are calculated on buckets of data
- **Matrix**: aggregation that operates on multiple fields and produces a matrix result
  - **matrix\_stats** is currently the only matrix aggregation
- **Pipeline**: aggregate the output of other aggregations (instead of from documents)



# Buckets and Metrics

Yasaswy Raval - 24-Apr-2017 - Capital One

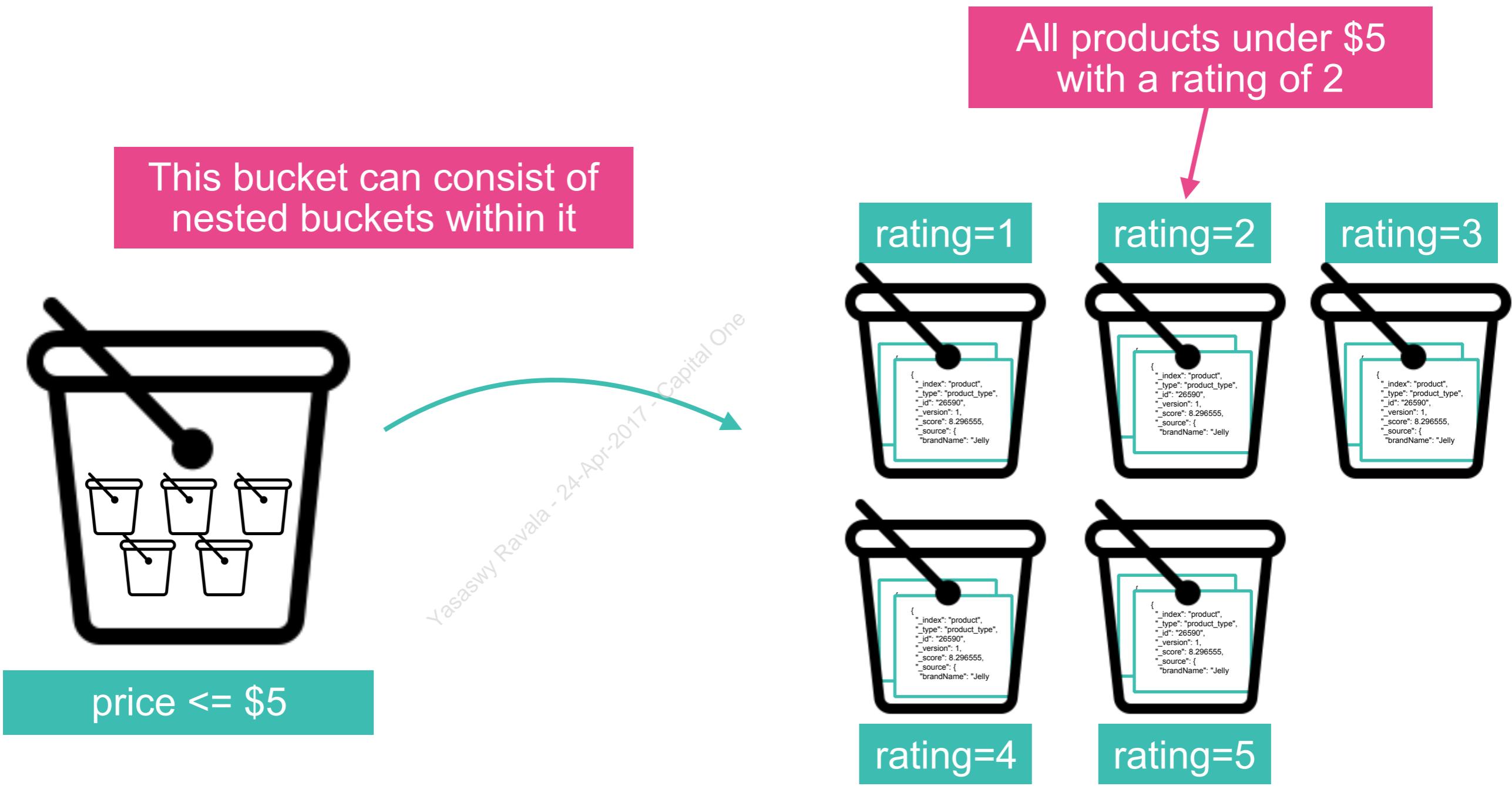
# Buckets

- A **bucket** is simply a collection of documents that meet a criterion
  - Buckets are a key element of aggregations
- For example, suppose we want to analyze products by their price range. We could put them into specific buckets:



# Buckets can be Nested

- Suppose we want to analyze products under \$5 and also by their customer rating



# Metrics

- **Metrics** compute numeric values based on your dataset
  - Either from fields
  - Or from values generated by custom scripts
- Most metrics are mathematical operations that output a single value:
  - **avg, sum, min, max, cardinality**
- Some metrics output multiple values:
  - **stats, percentiles, percentile\_ranks**

# Metrics + Buckets = Aggregations

- An *aggregation* can be a combination of *buckets* and *metrics*
  - This is why aggregations are so powerful - they can be combined in any number of combinations
  - For example, you could sum all the values of the “**quantitySold**” field, which would be one metric on one bucket
  - You could also compute the average “**quantitySold**” by price range, which would be one metric on multiple buckets:

What  
is the average  
quantity of items sold  
for products under \$5,  
sorted by customer  
rating.

That  
will require  
buckets and  
metrics



# Common Metrics Aggregations

Yasaswy Raval - 24-Apr-2017, Capital One

# The avg Metric

- The **avg** metric aggregation computes the *average* of a specified numeric field (or using a script)

```
GET nutrition/_search
{
  "size" : 0,
  "query" : {
    "match": {
      "ingredients": "sugar"
    }
  },
  "aggs" : {
    "my_sugar_calorie_average" : {
      "avg": {
        "field": "calories"
      }
    }
  }
}
```

*What is the average number of calories of products that have sugar in them?*

```
"aggregations": {
  "my_sugar_calorie_average": {
    "value": 155.44347826086957
  }
}
```

# min and max

- These two metrics are fairly straightforward:

```
GET nutrition/_search
{
  "size" : 0,
  "aggs" : {
    "my_max_calorie_item" : {
      "max": {
        "field": "calories"
      }
    },
    "my_min_calorie_item" : {
      "min": {
        "field": "calories"
      }
    }
  }
}
```

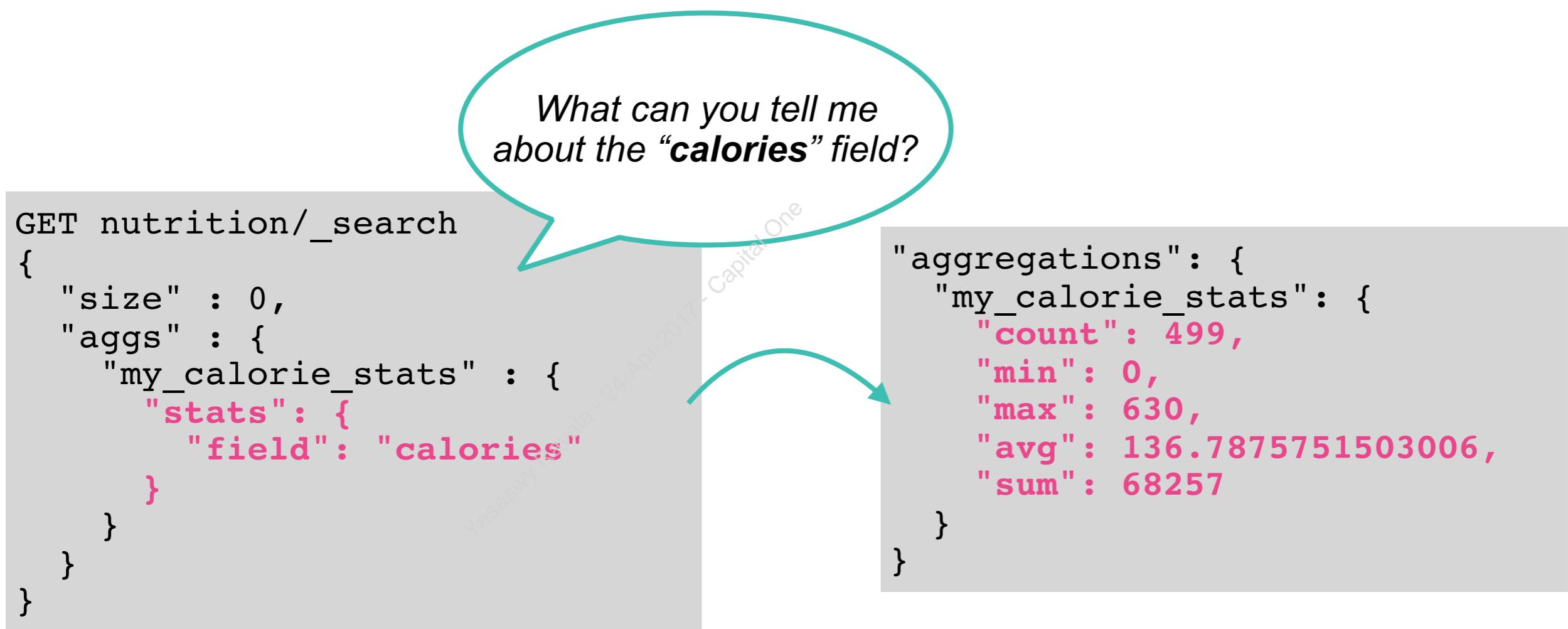
*What is the maximum and minimum number of calories of all products?*

```
"aggregations": {
  "my_min_calorie_item": {
    "value": 0
  },
  "my_max_calorie_item": {
    "value": 630
  }
}
```

Notice you can have multiple aggregations in a single “aggs” clause

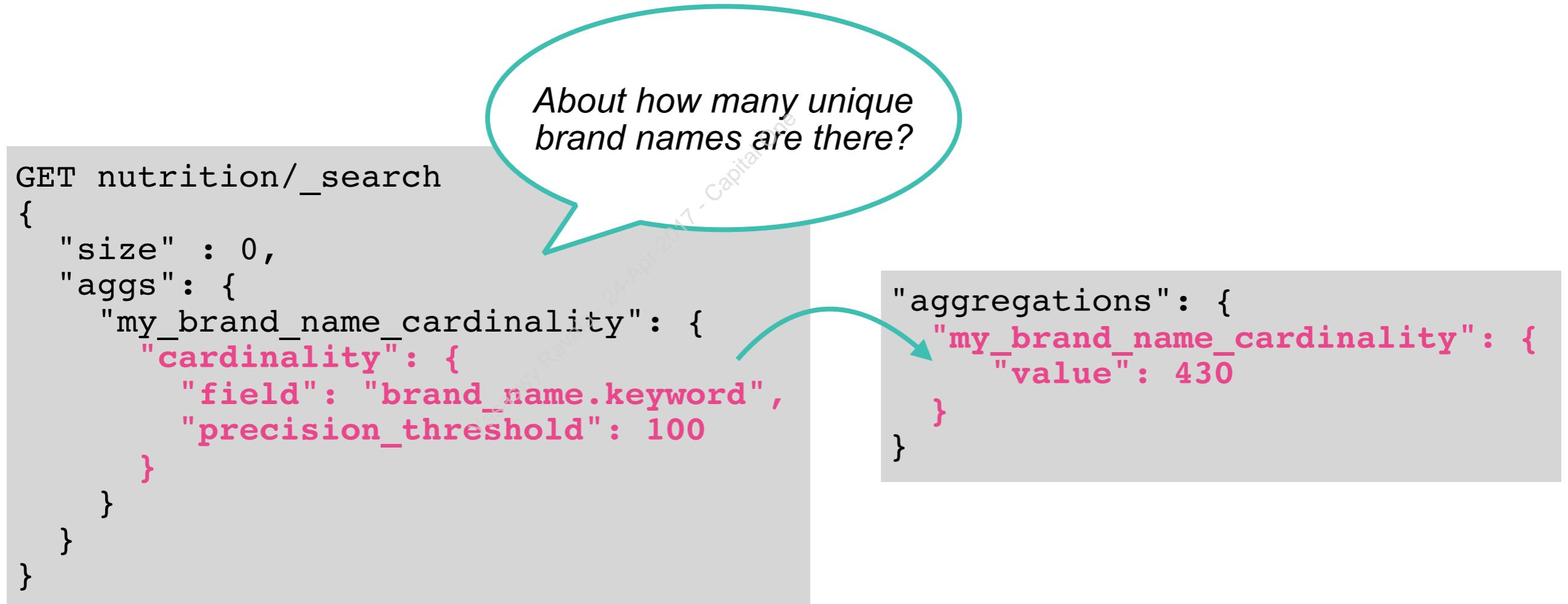
# The stats Aggregation

- The **stats** metric aggregation gives you common statistics of a field in a single aggregation request:
  - count, min, max, avg and sum



# The cardinality Aggregation

- The **cardinality** metric aggregation is an *approximation* of the number of distinct values of a field
  - based on the HyperLogLog++ algorithm
  - accurate up until **precision\_threshold** (max of 40,000)



# Common Bucket Aggregations

Yasaswy Raval - 24-Apr-2017, Capital One

# Common Bucket Aggregations

- Let's look at two useful bucket aggregations:
  - range
  - terms

Yasaswy Raval - 24-Apr-2017 - Capital One



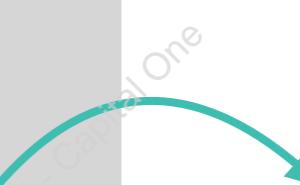
# The range Aggregation

Yasaswy Raval - 24-Apr-2017 - CapitalOne

# The range Aggregation

- The **range** bucket aggregation puts documents into buckets based on ranges of values on a specified field

```
GET nutrition/_search
{
  "size" : 0,
  "aggs" : {
    "my_calorie_range_buckets" : {
      "range": {
        "field": "calories",
        "ranges": [
          {
            "to": 100
          },
          {
            "from": 100,
            "to": 200
          },
          {
            "from" : 200
          }
        ]
      }
    }
  }
}
```



```
"aggregations": {
  "my_calorie_range_buckets": {
    "buckets": [
      {
        "key": "*-100.0",
        "to": 100,
        "doc_count": 178
      },
      {
        "key": "100.0-200.0",
        "from": 100,
        "to": 200,
        "doc_count": 207
      },
      {
        "key": "200.0-*",
        "from": 200,
        "doc_count": 114
      }
    ]
  }
}
```

# Labeling Ranges

- Add a “key” field to label a range using any string you like
  - Your “key” field gets returned in the results

```
"aggs" : {  
  "my_calorie_range_buckets" : {  
    "range": {  
      "field": "calories",  
      "ranges": [  
        {  
          "key" : "low_calorie",  
          "to": 100  
        },  
        {  
          "key" : "medium_calorie",  
          "from": 100,"to": 200  
        },  
        {  
          "key" : "high_calorie",  
          "from" : 200  
        }  
      ]  
    }  
  }  
}
```

```
"aggregations": {  
  "my_calorie_range_buckets": {  
    "buckets": [  
      {  
        "key": "low_calorie",  
        "to": 100,  
        "doc_count": 178  
      },  
      {  
        "key": "medium_calorie",  
        "from": 100,  
        "to": 200,  
        "doc_count": 207  
      },  
      {  
        "key": "high_calorie",  
        "from": 200,  
        "doc_count": 114  
      }  
    ]  
  }  
}
```

# Combining Buckets and Metrics

- The previous range example created three buckets
  - Let's perform a metrics aggregation on each bucket:

```
GET nutrition/_search
{
  "size" : 0,
  "ags" : {
    "my_calorie_range_buckets" : {
      "range": {
        "field": "calories",
        "ranges": [
          {"to": 100},
          {"from": 100,"to": 200},
          {"from" : 200}
        ]
      },
      "ags" : {
        "my_average_total_fat": {
          "avg": {
            "field": "total_fat"
          }
        }
      }
    }
  }
}
```

Add a nested aggregation to  
“my\_calorie\_range\_buckets”

The “avg” will be computed  
for each bucket

# The result of our nested aggregation:

```
"aggregations": {  
    "my_calorie_range_buckets": {  
        "buckets": [  
            {  
                "key": "*-100.0",  
                "to": 100,  
                "doc_count": 178,  
                "my_average_total_fat": {  
                    "value": 1.2522471911702933  
                }  
            },  
            {  
                "key": "100.0-200.0",  
                "from": 100,  
                "to": 200,  
                "doc_count": 207,  
                "my_average_total_fat": {  
                    "value": 4.618357487922705  
                }  
            },  
            {  
                "key": "200.0-*",  
                "from": 200,  
                "doc_count": 114,  
                "my_average_total_fat": {  
                    "value": 12.20438596501685  
                }  
            }  
        ]  
    }  
}
```

The average total fat for items with less than 100 calories

# The date\_range Aggregation

Yasaswy Raval - 24-Apr-2011, Capital One

# The Date Range Aggregation

- The `date_range` bucket aggregation is similar to `range` except it is specifically used for `date` fields
  - syntax looks like:

```
GET my_index/_search
{
  "aggs" : {
    "my_date_range_agg" : {
      "date_range": {
        "field": "FIELD",
        "ranges": [
          {
            "from": "FROM",
            "to": "TO"
          }
        ]
      }
    }
  }
}
```

You can use actual dates or date math expressions



# Example of date\_range

```
GET stocks/_search
{
  "size": 0,
  "aggs": {
    "my_week_comparison": {
      "date_range": {
        "field": "trade_date",
        "ranges": [
          {
            "from": "2010-01-01||-1w",
            "to": "2010-01-01"
          },
          {
            "from": "2010-01-01",
            "to": "2010-01-01||+1w"
          }
        ],
        "aggs": {
          "my_total_volume": {
            "sum": {
              "field": "volume"
            }
          }
        }
      }
    }
  }
}
```

*“Compare the week-to-week stock volume from January 1, 2010.”*

"my\_total\_volume": {  
 "value": 75871338  
}

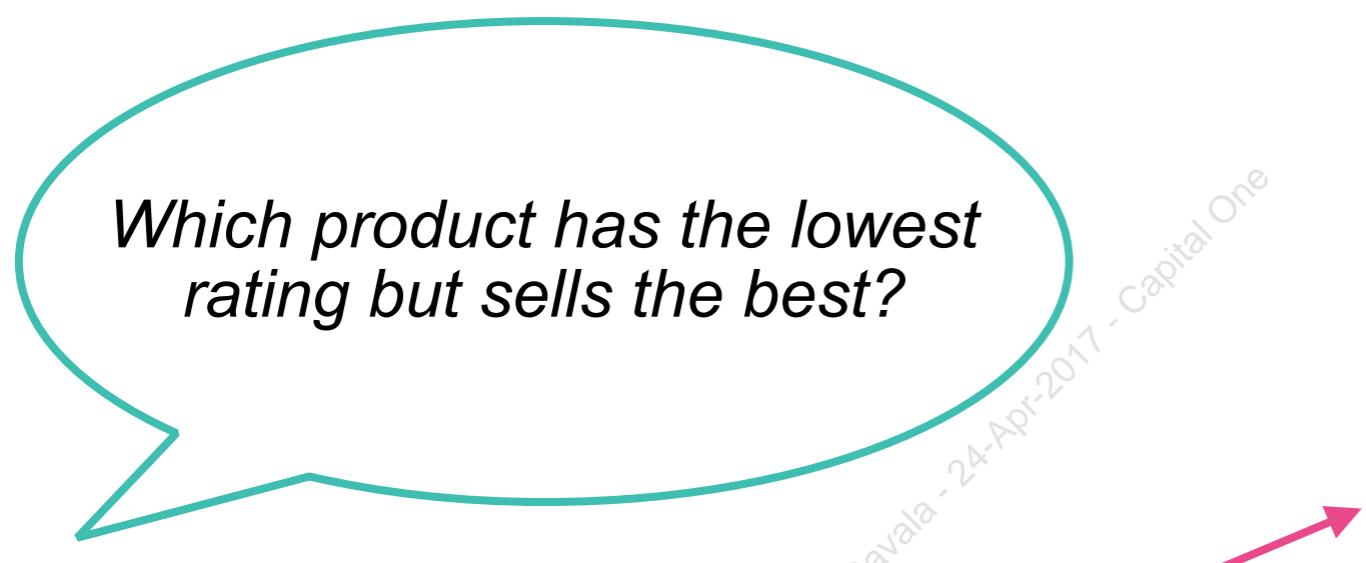
"my\_total\_volume": {  
 "value": 161270672  
}

# The terms Aggregation

Yasaswy Raval - 24-Apr-2017 - Capital One

# Terms Aggregation

- The **terms** aggregation will dynamically create a new bucket for every unique term it encounters of the specified field
  - In other words, each distinct value of the field will have its own bucket of documents



Start by using “**terms**” to put the products into distinct buckets by rating



# Simple Example of terms

- You just need to specify a “**field**” to bucket the terms by:

```
GET products/_search
{
  "size" : 0,
  "aggs" : {
    "my_customer_ratings" : {
      "terms": {
        "field": "customerRating",
        "size": 5
      }
    }
  }
}
```

size = number of buckets  
(defaults to 10)

# The Output of our terms Aggregation:

- Notice each bucket has a “key” that represents the distinct value of “field”,
- and “doc\_count” for the number of docs in the bucket

“2” is the most common rating

```
"aggregations": {  
  "my_customer_ratings": {  
    "doc_count_error_upper_bound": 0,  
    "sum_other_doc_count": 0,  
    "buckets": [  
      {  
        "key": 2,  
        "doc_count": 22190  
      },  
      {  
        "key": 3,  
        "doc_count": 22152  
      },  
      {  
        "key": 1,  
        "doc_count": 22100  
      },  
      {  
        "key": 4,  
        "doc_count": 22009  
      },  
      {  
        "key": 5,  
        "doc_count": 21984  
      } ] } }
```

# Understanding the Output of terms

- There are two values returned in a terms aggregation:
  - “**“doc\_count\_error\_upper\_bound”**”: the maximum number of missing documents that could potentially have appeared in a bucket
  - “**“sum\_other\_doc\_count”**”: the number of documents that do not appear in any of the buckets

```
GET nutrition/_search
{
  "size" : 0,
  "aggs" : {
    "my_brand_names" : {
      "terms" : {
        "field": "brand_name.keyword"
      }
    }
  }
}
```



```
"aggregations": {
  "my_brand_names": {
    "doc_count_error_upper_bound": 5,
    "sum_other_doc_count": 469,
    "buckets": [
      {
        "key": "Giant",
        "doc_count": 6
      },
      ...
    ]
  }
}
```

The value 6 may not be accurate

# Enabling show\_term\_doc\_count\_error

- Shows you the upper-bound error for each bucket:

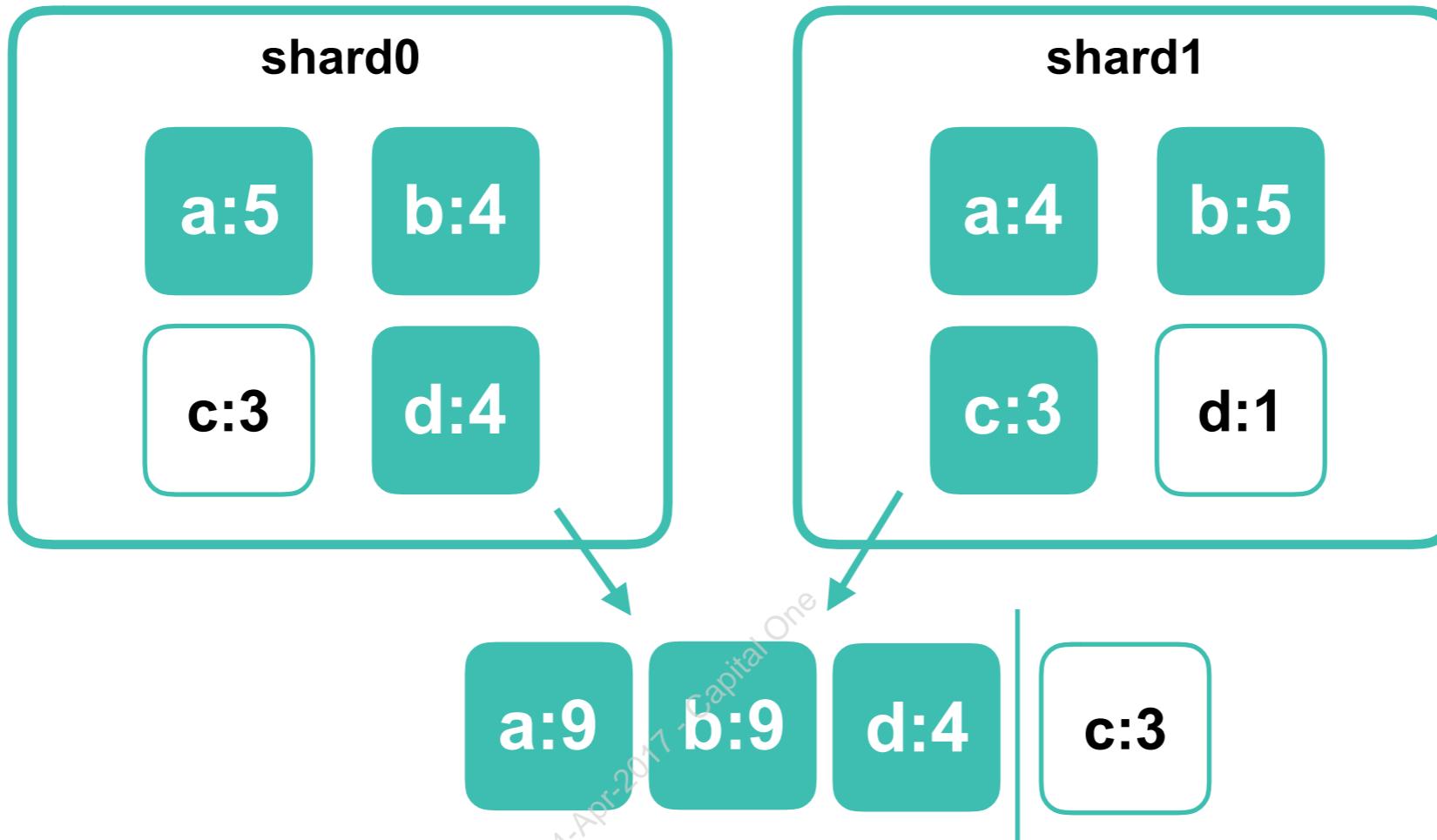
```
GET nutrition/_search
{
  "size" : 0,
  "aggs" : {
    "my_brand_names" : {
      "terms": {
        "field": "brand_name.keyword",
        "show_term_doc_count_error": true
      }
    }
  }
}
```

Both “Giant” and  
“Essential Everyday” can  
be at most 7



```
"my_brand_names": {
  "doc_count_error_upper_bound": 5,
  "sum_other_doc_count": 469,
  "buckets": [
    {
      "key": "Giant",
      "doc_count": 6,
      "doc_count_error_upper_bound": 1
    },
    {
      "key": "Essential Everyday",
      "doc_count": 4,
      "doc_count_error_upper_bound": 3
    }
  ]
}
```

# Why are terms aggs not always precise?



If **size=3**, we get an inaccurate result  
(c is 6 and should be in the top 3)

# The shard\_size Parameter

- **shard\_size** tells Elasticsearch to gather more top terms from each shard to improve accuracy
  - `shard_size = (size x 1.5) + 10` (by default)

Yasaswy Raval - 24-Apr-2017 - Capital One

# Increasing the shard\_size Parameter

- Let's try increasing `shard_size` to something larger than its default value:

```
GET nutrition/_search
{
  "size" : 0,
  "aggs" : {
    "my_brand_names" : {
      "terms": {
        "field": "brand_name.keyword",
        "show_term_doc_count_error": true,
        "size": 5,
        "shard_size": 110
      }
    }
  }
}
```

That lowered the error upper bound to 0 for the top hits

```
"my_brand_names": {
  "doc_count_error_upper_bound": 0,
  "sum_other_doc_count": 474,
  "buckets": [
    {
      "key": "Giant",
      "doc_count": 6,
      "doc_count_error_upper_bound": 0
    },
    "key": "Essential Everyday",
    "doc_count": 5,
    "doc_count_error_upper_bound": 0
  ],
}
```

# Nesting Buckets

Yasaswy Raval - 24-Apr-2017 - Capital One

# Nesting Buckets

- A bucket can be defined within a bucket:

```
GET nutrition/_search
{
  "size" : 0,
  "aggs" : {
    "my_brand_names" : {
      "terms": {
        "field": "brand_name.keyword",
        "size" : 40,
        "shard_size" : 110
      },
      "aggs": {
        "my_calories_buckets": {
          "range": {
            "field": "calories",
            "ranges": [
              {
                "from": 0,
                "to": 100
              },
              {
                "from" : 100
              }
            ]
          }
        }
      }
    }
  }
}
```

Put docs with the same  
“**brand\_name**” into a  
bucket

Put docs from each  
“**brand\_name**” into two  
buckets: “**calories**” < 100  
and “**calories**” >= 100



# Output of the example:

```
"aggregations": {  
    "my_brand_names": {  
        "doc_count_error_upper_bound": 0,  
        "sum_other_doc_count": 395,  
        "buckets": [  
            {  
                "key": "Giant",  
                "doc_count": 6,  
                "my_calories_buckets": {  
                    "buckets": [  
                        {  
                            "key": "0.0-100.0",  
                            "from": 0,  
                            "to": 100,  
                            "doc_count": 2  
                        },  
                        {  
                            "key": "100.0-*",  
                            "from": 100,  
                            "doc_count": 4  
                        }  
                    ]  
                }  
            },  
        ]  
    },  
},
```

There are six items from “Giant”, with 2 items below 100 calories and 4 items above 100 calories

# Chapter Review

Yasaswy Raval - 24-Apr-2017 - Capital One

# Summary

- You have two new tools in your Elasticsearch toolbox:
  - Metrics Aggregations like min, max, avg, stats
  - Bucket Aggregations like range and terms
- **Metrics** compute numeric values based on your dataset
- A **bucket** is a collection of documents that meet a criterion
- An **aggregation** can be thought of as a unit-of-work that builds analytic information over a set of documents
- Aggregations can be nested within other aggregations
- The **range** aggregation puts documents into buckets based on ranges of values on a specified field
- The **terms** aggregation dynamically creates buckets for every unique term it encounters of a specified field



# Quiz

1. **Query or Aggregation:** “What is the box office revenue of all movies directed by Steven Spielberg?”
2. **Query or Aggregation:** “What are the nearby restaurants that serve pizza?”
3. What are the four types of aggregations?
4. What aggregation would you use to put logging events into buckets by log type (“error”, “warn”, “info”, etc.)?
5. How can you increase the accuracy of a **terms** aggregation?
6. What aggregation(s) would you use to answer the question “How many unique visitors came to our website today?”

# Lab 8

## Working with Aggregations

Yasaswy Raval - 24-Apr-2017 - Capital One

# Chapter 9

# More

# Aggregations

Yasaswy Raval - 24-Apr-2017 - Capital One

- 1 Introduction to Elasticsearch
- 2 The Search API
- 3 Text Analysis
- 4 Mappings
- 5 More Search Features
- 6 The Distributed Model
- 7 Working with Search Results
- 8 Aggregations
- 9 More Aggregations
- 10 Handling Relationships

# Topics covered:

- Global Aggregation
- The missing Aggregation
- Histograms
- Date Histograms
- Percentiles
- Top Hits
- Significant Terms
- Sorting Buckets

Yasaswy Raval - 24-Apr-2017 - Capital One

# Global Aggregation

Yasaswy Raval - 24-Apr-2017 - CapitalOne

# The global Aggregation

- Use the **global** bucket aggregation to perform an aggregation over all documents
  - Even when a corresponding query is used
- The **scope** of a global aggregation is all documents

Yasaswy Raval - 24-Apr-2017 - Capital One

# Example of global Aggregation

```
GET nutrition/_search
{
  "size": 0,
  "query": {
    "term": {
      "calories": 120
    }
  },
  "aggs": {
    "max_total_fat_from_120": {
      "max": {
        "field": "total_fat"
      }
    },
    "all_of_my_items": {
      "global": {},
      "aggs": {
        "max_total_fat_from_all": {
          "max": {
            "field": "total_fat"
          }
        }
      }
    }
  }
}
```

*I want the max **total\_fat** of items with 120 calories, compared to the max **total\_fat** of all items.*

This **max** total fat of items with 120 calories

This **max** total fat of all documents

# Output from the global Aggregation:

```
"aggregations": {  
    "all_of_my_items": {  
        "doc_count": 499,  
        "max_total_fat_from_all": {  
            "value": 36  
        }  
    },  
    "max_total_fat_from_120": {  
        "value": 14  
    }  
}
```

# The missing Aggregation

Yasaswy Raval - 24-Apr-2011, Capital One

# The missing Aggregation

- The **missing** bucket aggregation creates a bucket of all documents that are missing a field value
  - Either the field is not defined or it contains a null value
  - Useful when you are creating buckets on a field value and you have documents missing that particular field
- The syntax looks like:

```
GET my_index/_search
{
  "aggs": {
    "my_bucket": {
      "missing": {
        "field": "FIELD_NAME"
      }
    }
  }
}
```

# Example of missing

```
GET nutrition/_search
{
  "size": 0,
  "aggs": {
    "my_missing_ingredients": {
      "missing": {
        "field": "ingredients"
      },
      "aggs": {
        "my_brands": {
          "terms": {
            "field": "brand_name.keyword"
          }
        }
      }
    }
  }
}
```

*“Show me the brand names with the most products which are missing their ingredients.”*

“Giant” has 5 items missing their ingredients

```
"my_missing_ingredients": {
  "doc_count": 250,
  "my_brands": {
    "doc_count_error_upper_bound": 2,
    "sum_other_doc_count": 222,
    "buckets": [
      {
        "key": "Giant",
        "doc_count": 5
      },
      {
        "key": "Unknown",
        "doc_count": 5
      },
      {
        "key": "Huggies",
        "doc_count": 5
      }
    ]
}
```

# Histograms

Yasaswy Raval - 24-Apr-2017 - Capital One

# The histogram Aggregation

- The *histogram* bucket aggregation builds a histogram on a given “field” using a specified “interval”:
  - A histogram is like a bar chart that shows how many values fit into various intervals

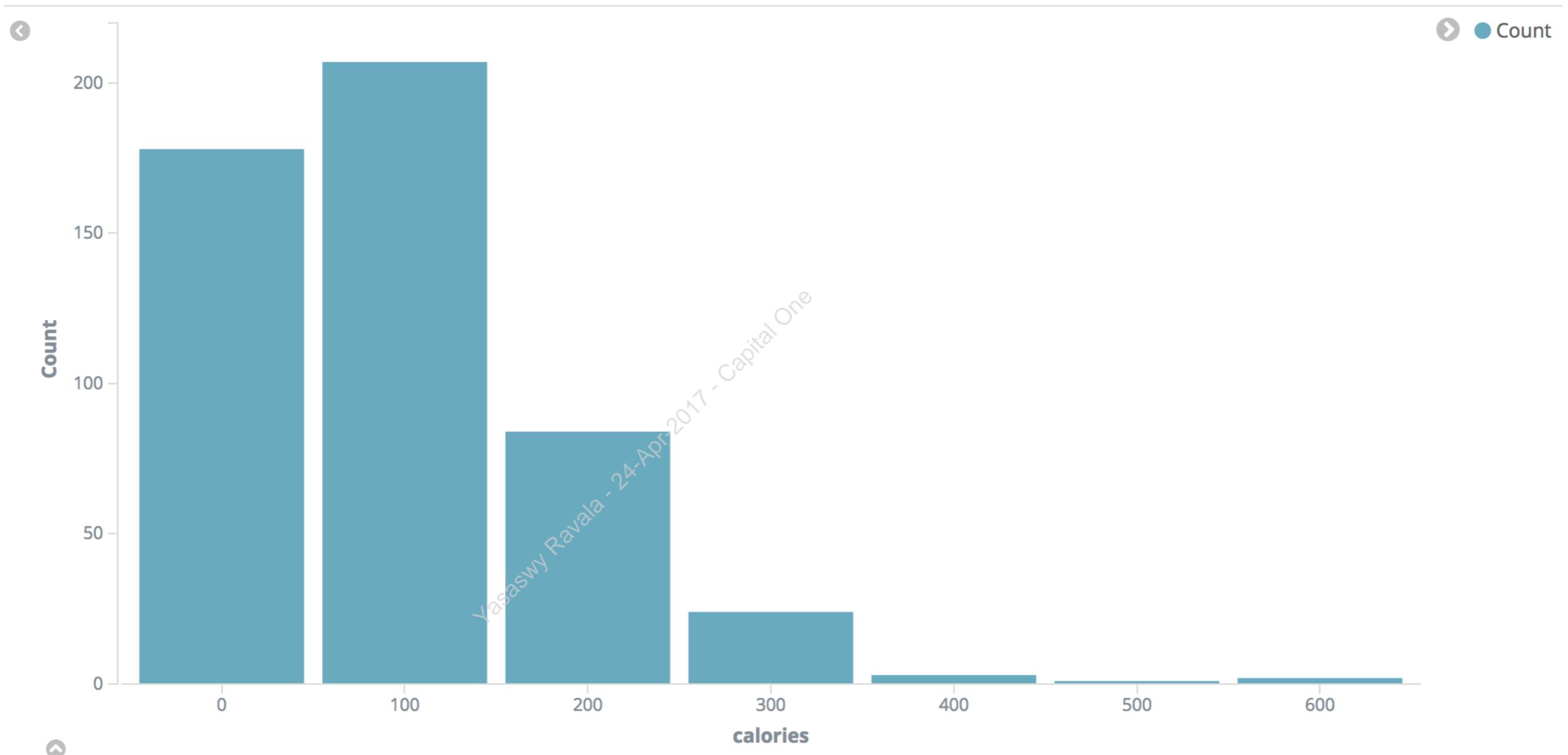
```
GET nutrition/_search
{
  "size": 0,
  "aggs": {
    "my_calories_histogram": {
      "histogram": {
        "field": "calories",
        "interval": 100
      }
    }
  }
}
```

A bucket is created for every 100 calories



# The output of histogram:

- It's a histogram!
  - (OK, we cheated and used a Kibana visualization)



# The API result:

- The number of buckets is dynamic and depends on the input values and the interval

24 items have calories between 300 inclusive and 400 exclusive

```
"aggregations": {  
    "my_calories_histogram": {  
        "buckets": [  
            {  
                "key": 0,  
                "doc_count": 178  
            },  
            {  
                "key": 100,  
                "doc_count": 207  
            },  
            {  
                "key": 200,  
                "doc_count": 84  
            },  
            {  
                "key": 300,  
                "doc_count": 24  
            },  
            {  
                "key": 400,  
                "doc_count": 3  
            },  
            {  
                "key": 500,  
                "doc_count": 1  
            },  
            {  
                "key": 600,  
                "doc_count": 2  
            }  
        ]  
    }  
}
```

# The min\_doc\_count Parameter

- You can use `min_doc_count` to raise the minimum count to throw out smaller buckets

```
GET nutrition/_search
{
  "size": 0,
  "aggs": {
    "my_calories_histogram": {
      "histogram": {
        "field": "calories",
        "interval": 100,
        "min_doc_count": 10
      }
    }
  }
}
```



```
"aggregations": {
  "my_calories_histogram": {
    "buckets": [
      {
        "key": 0,
        "doc_count": 178
      },
      {
        "key": 100,
        "doc_count": 207
      },
      {
        "key": 200,
        "doc_count": 84
      },
      {
        "key": 300,
        "doc_count": 24
      }
    ]
  }
}
```

# Metrics on a Histogram

- Just like any bucket, you can perform metrics on a histogram
  - Let's look at the **stats** of each bucket

```
GET nutrition/_search
{
  "size": 0,
  "aggs": {
    "my_calories_histogram": {
      "histogram": {
        "field": "calories",
        "interval": 100,
        "min_doc_count": 10
      },
      "aggs": {
        "my_bucket_stats": {
          "stats": {
            "field": "calories"
          }
        }
      }
    }
  }
}
```

metric within a bucket



# Output of stats:

There is at least one product with 0 calories

```
"aggregations": {
    "my_calories_histogram": {
        "buckets": [
            {
                "key": 0,
                "doc_count": 178,
                "bucket_stats": {
                    "count": 178,
                    "min": 0,
                    "max": 96,
                    "avg": 47.29775280898876,
                    "sum": 8419
                }
            },
            {
                "key": 100,
                "doc_count": 207,
                "bucket_stats": {
                    "count": 207,
                    "min": 100,
                    "max": 195,
                    "avg": 137.8985507246377,
                    "sum": 28545
                }
            },
            {
                "key": 200,
                "doc_count": 195,
                "bucket_stats": {
                    "count": 195,
                    "min": 200,
                    "max": 295,
                    "avg": 245.05263157894735,
                    "sum": 47955
                }
            },
            {
                "key": 300,
                "doc_count": 195,
                "bucket_stats": {
                    "count": 195,
                    "min": 300,
                    "max": 395,
                    "avg": 347.5263157894737,
                    "sum": 67825
                }
            },
            {
                "key": 400,
                "doc_count": 195,
                "bucket_stats": {
                    "count": 195,
                    "min": 400,
                    "max": 495,
                    "avg": 447.5263157894737,
                    "sum": 87825
                }
            },
            {
                "key": 500,
                "doc_count": 195,
                "bucket_stats": {
                    "count": 195,
                    "min": 500,
                    "max": 595,
                    "avg": 547.5263157894737,
                    "sum": 107825
                }
            },
            {
                "key": 600,
                "doc_count": 195,
                "bucket_stats": {
                    "count": 195,
                    "min": 600,
                    "max": 695,
                    "avg": 647.5263157894737,
                    "sum": 127825
                }
            },
            {
                "key": 700,
                "doc_count": 195,
                "bucket_stats": {
                    "count": 195,
                    "min": 700,
                    "max": 795,
                    "avg": 747.5263157894737,
                    "sum": 147825
                }
            },
            {
                "key": 800,
                "doc_count": 195,
                "bucket_stats": {
                    "count": 195,
                    "min": 800,
                    "max": 895,
                    "avg": 847.5263157894737,
                    "sum": 167825
                }
            },
            {
                "key": 900,
                "doc_count": 195,
                "bucket_stats": {
                    "count": 195,
                    "min": 900,
                    "max": 995,
                    "avg": 947.5263157894737,
                    "sum": 187825
                }
            }
        ]
    }
}
```

There are 178 products  
with less than 100  
calories

# Date Histograms

Yasaswy Raval - 24-Apr-2017 - Capital One

# Date Histograms

- The **date\_histogram** bucket aggregation creates buckets over a **date** field and specified **interval**
  - Histograms over date ranges are a powerful tool in data analytics
- The syntax looks like:

```
GET stocks/_search
{
  "aggs": {
    "my_date_histogram": {
      "date_histogram": {
        "field": "DATE_FIELD",
        "interval": "INTERVAL"
      }
    }
  }
}
```

must be a “**date**” field

the interval of each bucket

# Intervals

- Valid values for **interval** can either be one of the following strings:
  - **year, quarter, month, week, day, hour, minute, second**
- Or a **time unit**:
  - a number followed by a time unit
  - for example: “**2d**” is two days
- Time units are:
  - **d** = day, **h** = hour, **m** = minutes, **s** = seconds, **ms** = milliseconds

# Example of a Date Histogram

- Let's run a `date_histogram` aggregation on the stock prices:

```
GET stocks/_search
{
  "size": 0,
  "aggs": {
    "my_weekly_buckets": {
      "date_histogram": {
        "field": "trade_date",
        "interval": "1w"
      }
    }
  }
}
```

*Put the stock prices from each week into a bucket*

# The stocks in weekly buckets:

```
"aggregations": {  
    "my_weekly_buckets": {  
        "buckets": [  
            {  
                "key_as_string": "2009-08-20T00:00:00.000Z",  
                "key": 1250726400000,  
                "doc_count": 1996  
            },  
            {  
                "key_as_string": "2009-08-27T00:00:00.000Z",  
                "key": 1251331200000,  
                "doc_count": 2495  
            },  
            {  
                "key_as_string": "2009-09-03T00:00:00.000Z",  
                "key": 1251936000000,  
                "doc_count": 1497  
            },  
            {  
                "key_as_string": "2009-09-10T00:00:00.000Z",  
                "key": 1252540800000,  
                "doc_count": 2495  
            },  
            {  
                "key_as_string": "2009-09-17T00:00:00.000Z",  
                "key": 1253134400000,  
                "doc_count": 1497  
            }  
        ]  
    }  
}
```

Notice the buckets are  
7 days apart

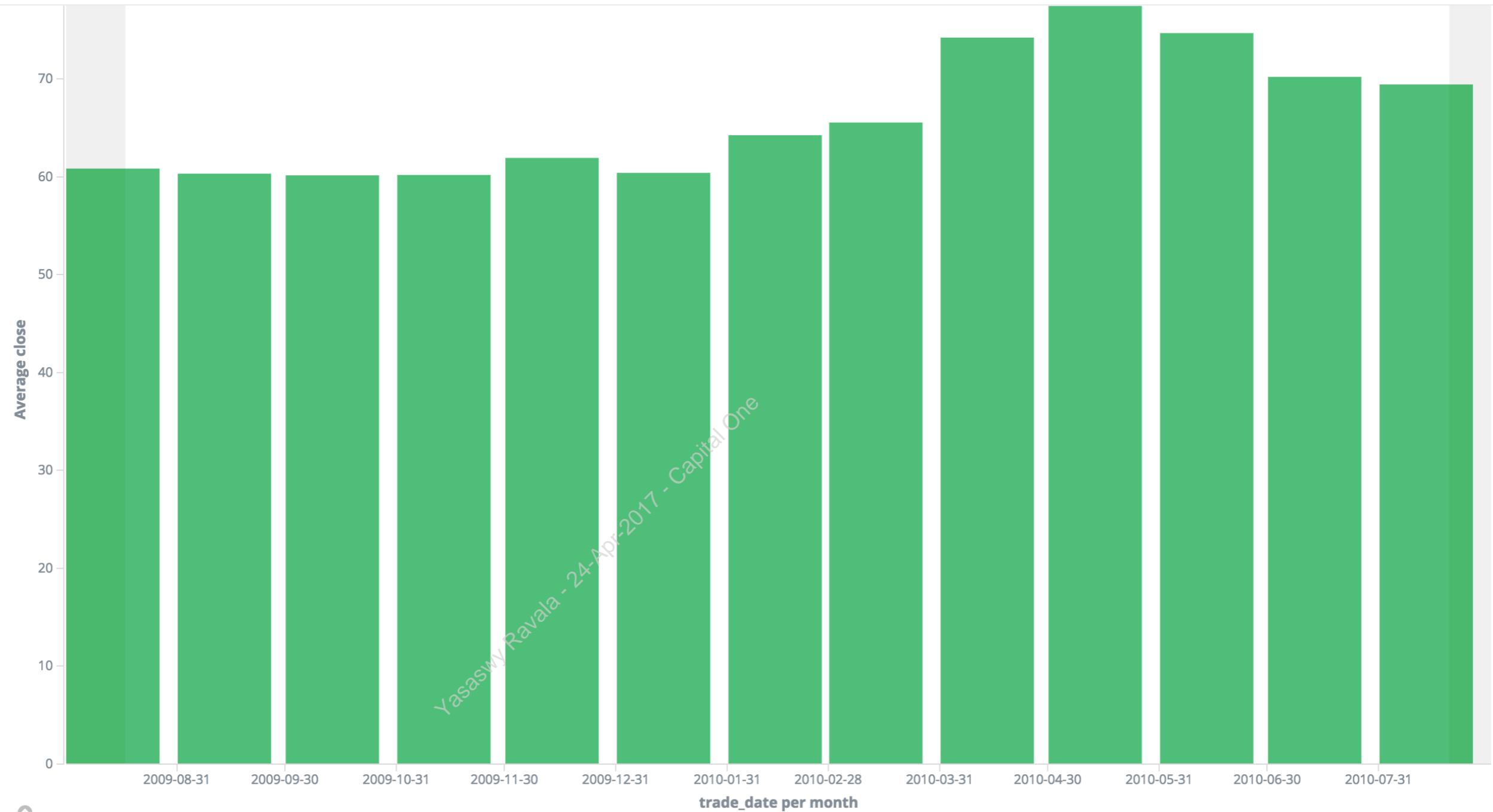
# Example of Metrics on a Date Histogram

```
GET stocks/_search
{
  "query": {
    "match": {
      "stock_symbol": "SHW"
    }
  },
  "size": 0,
  "aggs": {
    "my_stock_date_histogram": {
      "date_histogram": {
        "field": "trade_date",
        "interval": "month"
      },
      "aggs": {
        "my_avg_close": {
          "avg": {
            "field": "close"
          }
        }
      }
    }
  }
}
```

*What is the monthly average stock price of the “SHW” stock?*

2009-08-01 = 60.85857173374721  
2009-09-01 = 60.339000129699706  
2009-10-01 = 60.16849994659424  
2009-11-01 = 60.203529582304114  
2009-12-01 = 61.94954560019753  
2010-01-01 = 60.4199995743601  
2010-02-01 = 64.26526320608039  
2010-03-01 = 65.56347905034605  
2010-04-01 = 74.24333336239769  
2010-05-01 = 77.47699966430665  
2010-06-01 = 74.70545473965731  
2010-07-01 = 70.23285675048828  
2010-08-01 = 69.4557135445731

# Visualization of our date\_histogram

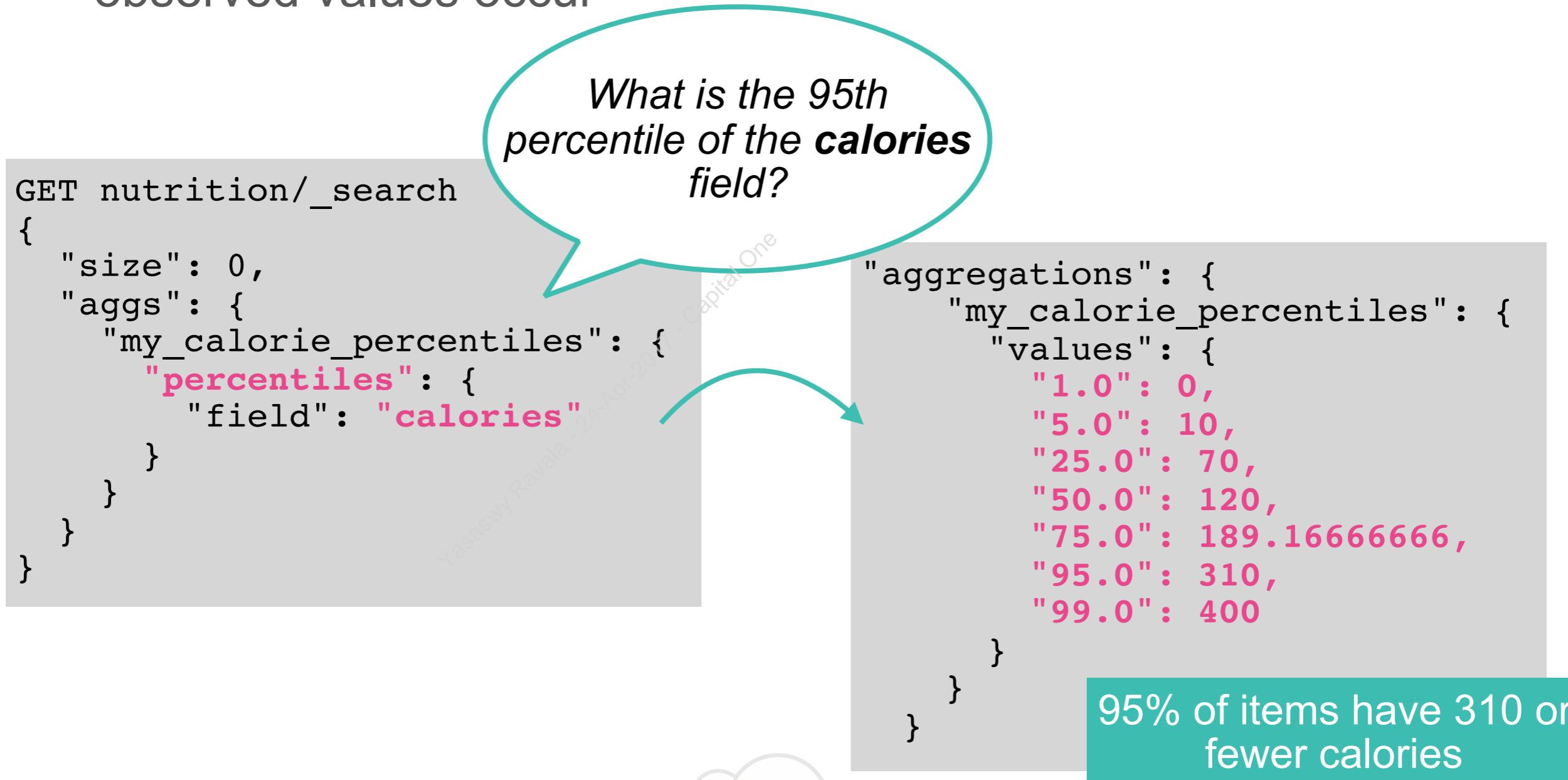


# Percentiles

Yasaswy Raval - 24-Apr-2017 - Capital One

# 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



# Percentiles Example

- Quintiles are a commonly-used percentile
  - 5 evenly-spaced percentages: 20% - 40% - 60% - 80% - 100%

```
GET nutrition/_search
{
  "size" : 0,
  "aggs": {
    "my_calorie_quintiles": {
      "percentiles": {
        "field": "calories",
        "percents": [
          20,
          40,
          60,
          80,
          100
        ]
      }
    }
  }
}
```

*What are the quintile values for calories?*

```
"aggregations": {
  "my_calorie_quintiles": {
    "values": {
      "20.0": 60,
      "40.0": 110,
      "60.0": 140,
      "80.0": 210,
      "100.0": 630
    }
  }
}
```

80% of the items have 210 or fewer calories

# 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**

What percentile is 150 calories?

```
GET nutrition/_search
{
  "size": 0,
  "aggs": {
    "my_calorie_percentiles": {
      "percentile_ranks": {
        "field": "calories",
        "values": [150, 300]
      }
    }
  }
}
```

You can pass in multiple values

65.3% of items have 150 calories or less, and 94.2% have less than 300

```
"aggregations": {
  "my_calorie_percentiles": {
    "values": {
      "150.0": 65.33066132264528,
      "300.0": 94.18837675350701
    }
  }
}
```

# Top Hits

Yasaswy Raval - 24-Apr-2017 - Capital One

# The Top Hits Aggregations

- **top\_hits** is a metrics aggregation that allows you to find the most relevant documents in a bucket
  - used typically as a sub-aggregation
- The syntax looks like:

```
"aggs": {  
  "my_top_hits": {  
    "top_hits": {  
      "size": SIZE,  
      "sort": [],  
      "from": OFFSET  
    }  
  }  
}
```

number of top hits to return per bucket

the offset from the first result you want to fetch

# Motivation for top\_hits

- Suppose we do a search for items with **sugar** in the **ingredients**, then bucket those by the number of **calories**:

```
GET nutrition/_search
{
  "size" : 0,
  "query": {
    "match": {
      "ingredients": "sugar"
    }
  },
  "aggs": {
    "my_calorie_bucket": {
      "terms": {
        "field": "calories"
      }
    }
  }
}
```

9 items with sugar have 140 calories, but what are the most relevant items in this bucket?

```
"buckets": [
  {
    "key": 140,
    "doc_count": 9
  },
  {
    "key": 120,
    "doc_count": 8
  },
  {
    "key": 270,
    "doc_count": 7
  },
  {
    "key": 150,
    "doc_count": 6
  },
  {
    "key": 180,
    "doc_count": 5
  },
  {
    "key": 210,
    "doc_count": 4
  },
  {
    "key": 240,
    "doc_count": 3
  },
  {
    "key": 270,
    "doc_count": 2
  },
  {
    "key": 300,
    "doc_count": 1
  }
]
```

# Example of top\_hits

```
GET nutrition/_search
{
  "size" : 0,
  "query": {
    "match": {
      "ingredients": "sugar"
    }
  },
  "aggs": {
    "my_calorie_bucket": {
      "terms": {
        "field": "calories"
      },
      "aggs": {
        "my_top_hits": {
          "top_hits": {
            "size": 5
          }
        }
      }
    }
  }
}
```

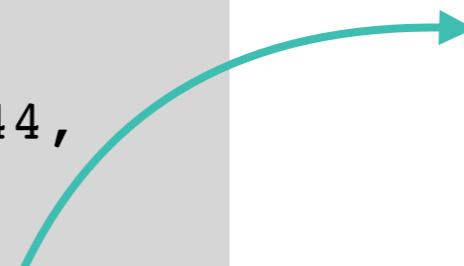
top\_hits is a sub-aggregation  
of the “calorie\_bucket”

Returns the top 5 hits in each  
bucket (based on its \_score of  
the “sugar” query)

# The output of top\_hits:

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

```
"aggregations": {  
    "my_calorie_bucket": {  
        "doc_count_error_upper_bound": 0,  
        "sum_other_doc_count": 57,  
        "buckets": [  
            {  
                "key": 140,  
                "doc_count": 9,  
                "my_top_hits": {  
                    "hits": {  
                        "total": 9,  
                        "max_score": 1.341344,  
                        "hits": [  
                            top 5 hits  
                        ]  
                    }  
                }  
            }  
        ]  
    }  
}
```



"Soda"
"Hot Cocoa Mix, Cappuccino Classics Suprema 1.16 Oz"
"Raisin Bran"
"Exotic Juice Drink, Guanabana"
"Simply Made Cookies, Butter"

# Significant Terms

Yasaswy Raval - 24-Apr-2017 - Capital One

# The Significant Terms Aggregation

- The **significant\_terms** bucket aggregation is used to find interesting or unusual occurrences of terms
- For example, suppose a bank needs to find fraudulent credit card transactions
  - A group of customers complains about fraudulent charges
  - A regular **terms** aggregation would reveal that a lot of those customers charged something recently at Amazon. This is “**commonly common**” - a lot of customers charge things at Amazon
  - Three of the complaining customers had charges at a local grocery store. This is “**commonly uncommon**” - only a few customers had this same transaction
  - **significant\_terms** helps you find the “**uncommonly common**” - the merchants with charges from all complaining customers, but not a lot of the other customers

# Let's Start with a Terms Agg

```
GET nutrition/_search
{
  "size":0,
  "aggs" : {
    "my_total_fat_histogram": {
      "histogram": {
        "field": "total_fat",
        "interval": 5,
        "min_doc_count": 5
      },
      "aggregations": {
        "my_top_words" : {
          "terms" : {
            "field" : "ingredients",
            "size":5
          }
        }
      }
    }
  }
}
```

*Let's look for a relationship between total fat and ingredients.*



# The Results of the Terms Arg

- These would be considered “commonly common” ingredients:

```
"key": 15,  
"doc_count": 31,  
"my_top_words": {  
    "buckets": [  
        {  
            "key": "salt",  
            "doc_count": 16  
        },  
        {  
            "key": "oil",  
            "doc_count": 14  
        },  
        {  
            "key": "and",  
            "doc_count": 11  
        },  
        {  
            "key": "corn",  
            "doc_count": 8  
        },  
        {  
            "key": "flour",  
            "doc_count": 9  
        } ] }},
```

The “commonly common” ingredients are “salt”, “oil”, “corn”, “flour” and the stopword “and”.



# Using significant\_terms

- Now let's try the same agg with **significant\_terms**:

```
GET nutrition/_search
{
  "size":0,
  "aggs" : {
    "my_total_fat_histogram": {
      "histogram": {
        "field": "total_fat",
        "interval": 5,
        "min_doc_count": 5
      },
      "aggregations":{
        "my_top_words" : {
          "significant_terms" : {
            "field" : "ingredients",
            "size":5
          }
        }
      }
    }
  }
}
```

*Let's look for the  
“uncommonly common”  
ingredients in relationship  
to total fat.*

# The Output of significant\_terms:

```
"key": 15,  
"doc_count": 31,  
"my_top_words": {  
    "doc_count": 31,  
    "buckets": [  
        {  
            "key": "almonds",  
            "doc_count": 5,  
            "score": 2.4349635796045783  
        },  
        {  
            "key": "cashews",  
            "doc_count": 4,  
            "score": 1.947970863683663  
        },  
        {  
            "key": "flax",  
            "doc_count": 4,  
            "score": 1.532570239334027,  
        },  
        {  
            "key": "brazil",  
            "doc_count": 3,  
            "score": 1.460978147762747,  
        },  
        {  
            "key": "dates",  
            "doc_count": 3,  
            "score": 1.460978147762747,  
        }  
    ]  
}
```

31 products with **total\_fat** between 15 and 20

The top 5 significant terms are “**almonds**”, “**cashews**”, “**flax**”, “**brazil**” and “**dates**”

# Sorting Buckets

Yasaswy Raval - 24-Apr-2017 - Capital One

# Default Bucket Sorting

- By default, the results within a bucket are sorted by **\_count** (the document count) in descending order
- You can specify the sorting of **terms** and the histograms using “**order**”:
  - **\_count**: this is the default; works with **terms**, **histogram** and **date\_histogram**
  - **\_term**: sort by the term alphabetically; only works with **terms**
  - **\_key**: Sort by the bucket’s key; works with **histogram** and **date\_histogram**
- You can also sort by a metric value in a nested aggregation

# Sort by \_key Example

```
GET stocks/_search
{
  "aggs": {
    "my_stock_date_histogram": {
      "date_histogram": {
        "field": "trade_date",
        "interval": "month",
        "order": {
          "_key": "desc"
        }
      },
      "aggs": {
        "my_avg_close": {
          "avg": {
            "field": "close"
          }
        }
      }
    }
  }
}
```

The default for **date\_histogram** is  
**\_key ascending**



# Sorting by a Metric

```
GET nutrition/_search
{
  "size" : 0,
  "aggs": {
    "my_brands_bucket": {
      "terms": {
        "field": "brand_name.keyword",
        "order": {
          "my_max_calories": "desc"
        }
      },
      "aggs": {
        "my_max_calories": {
          "max": {
            "field": "calories"
          }
        }
      }
    }
  }
}
```

*Find the brands with the highest-calorie products.*

This output will be sorted by the results of the nested metric

# Chapter Review

Yasaswy Raval - 24-Apr-2017 - Capital One

# Summary

- Use **global** to perform an aggregation over all documents
- The ***histogram*** bucket aggregation builds a histogram on a given “**field**” using a specified “**interval**”
- The ***date\_histogram*** bucket aggregation creates buckets over a **date** field and specified **interval**
- The ***percentiles*** metrics aggregation calculates percentiles over a numeric field
- ***top\_hits*** is a metrics aggregation that allows you to find the most relevant documents in another bucket
- The ***significant\_terms*** bucket aggregation is used to find interesting or unusual occurrences of terms

# Quiz

1. How many ERRORs occurred per day last month? Which aggregation would work well for this?
2. You want to recommend movies based on another movie that someone likes. Which aggregation might work well for finding recommendations that are out of the ordinary?
3. Which aggregations would you use to answer the question: “What are the top 3 posts from the top 10 users at StackOverflow?”
4. You want to compare the total volume of stocks from the tech industry to the overall market? What aggregation would you use?

# Lab 9

## More Aggregations

Yasaswy Raval - 24-Apr-2017 - Capital One

# Chapter 10

# Handling Relationships

Yasaswy Raval - 24-Apr-2017 - Capital One

- 1 Introduction to Elasticsearch
- 2 The Search API
- 3 Text Analysis
- 4 Mappings
- 5 More Search Features
- 6 The Distributed Model
- 7 Working with Search Results
- 8 Aggregations
- 9 More Aggregations
- 10 Handling Relationships

# Topics covered:

- The Need for Data Modeling
- Denormalization
- The Need for Nested Types
- Nested Types
- Querying a Nested Type
- Sorting on a Nested Type
- The Nested Aggregation
- Parent/Child Types
- The has\_child Query
- The has\_parent Query

# The Need for Data Modeling

Yasaswy Raval - 24-Apr-2017, Capital One

# 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**
  - **Parent/child relationships**
- We will discuss nested objects and parent/child relationships in detail in this chapter



# Denormalization

Yasaswy Raval - 24-Apr-2017 - Capital One

# Denormalization

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

Yasaswy Raval - 24-Apr-2017 - Capital One

# 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",  
  "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:

users

```
PUT users/doc/1
{
  "username" : "harrison",
  "city" : "Los Angeles",
  "state" : "California"
}
```

tweets

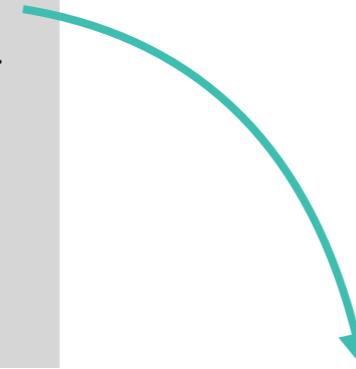
```
PUT tweets/doc/123
{
  "body" : "My favorite movie is Star Wars",
  "time" : "2017-01-24T02:32:27",
  "userid" : 1,
  "username" : "harrison"
}
```

```
PUT tweets/doc/456
{
  "body" : "Laugh it up, fuzzball.",
  "time" : "1977-05-25T00:00:00",
  "userid" : 1,
  "username" : "harrison"
}
```

# Example of Denormalization

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

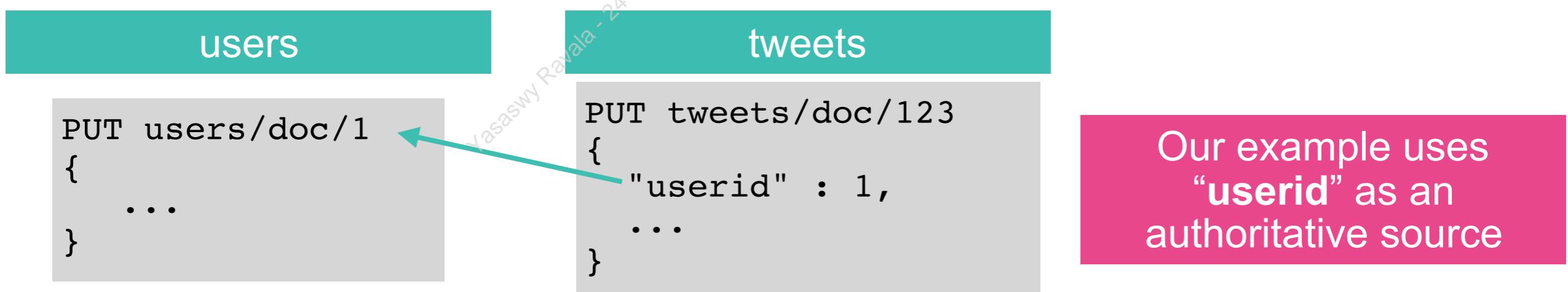
```
GET tweets/_search
{
  "query": {
    "bool": {
      "must": [
        {"match": {"body": "movie"}},
        {"match": {"username": "harrison"} }
      ]
    }
  }
}
```



```
"_score": 0.55510485,
"_source": {
  "body": "My favorite movie is Star Wars",
  "time": "2017-01-24T02:32:27",
  "userid": 1,
  "username": "harrison"
}
```

# 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

Yasaswy Raval - 24-Apr-2017 - Capital One

# Inner Objects

- There is an interesting (and confusing) scenario that can arise with JSON inner objects
- Suppose we index the following two documents into an index named “**companies**”:

```
PUT companies/doc/1
{
  "name" : "Stark Enterprises",
  "employees" : [
    {"first_name" : "Tony", "last_name" : "Stark"},
    {"first_name" : "Virginia", "last_name" : "Potts"}
  ]
}
```

```
PUT companies/doc/2
{
  "name" : "NBC Universal",
  "employees" : [
    {"first_name" : "Tony", "last_name" : "Potts"}
  ]
}
```



# Searching Inner Objects...

- What do you think the result is of the following query?

```
GET companies/_search
{
  "query": {
    "bool": {
      "must": [
        {"match": {"employees.first_name": "Tony"}},
        {"match": {"employees.last_name": "Potts"}}
      ]
    }
  }
}
```

?

# ...can have surprising results:

“Stark Enterprises” does not have an employee named “Tony Potts”

```
"hits": {
  "total": 2,
  "max_score": 0.5753642,
  "hits": [
    {
      "_index": "companies",
      "_type": "doc",
      "_id": "2",
      "_score": 0.5753642,
      "_": {
        "name": "NBC Universal",
        "employees": [
          {
            "first_name": "Tony",
            "last_name": "Potts"
          }
        ]
      }
    },
    {
      "_index": "companies",
      "_type": "doc",
      "_id": "1",
      "_score": 0.51623213,
      "_source": {
        "name": "Stark Enterprises",
        "employees": [
          {
            "first_name": "Tony",
            "last_name": "Stark"
          },
          {
            "first_name": "Virginia",
            "last_name": "Potts"
          }
        ]
      }
    }
  ]
}
```



# Why the confusing results?

- The “`first_name`” and “`last_name`” fields are JSON inner objects of the “`employees`” field
  - When the JSON object got flattened in the index, we lost the relationship between “**Tony**” and “**Potts**”

All first names and last names are put into respective arrays

```
{  
  "name" : "Stark Enterprises",  
  "employee.first_name" : [ "Tony", "Virginia" ],  
  "employee.last_name" : [ "Stark", "Potts" ]  
}
```

A query for “**Tony Potts**” is a hit  
(so is “**Virginia Stark**”)

# How do we fix this scenario?

- A hit on the inner object yields a hit on the root object
- The problem is with the mapping:
  - We need to use the “**nested**” data type for the JSON inner objects (in this particular scenario)

```
"companies": {  
    "mappings": {  
        "doc": {  
            "properties": {  
                "employees": {  
                    "properties": {  
                        "first_name": {  
                            "type": "text",  
                            "fields": {  
                                "keyword": {  
                                    "type": "keyword",  
                                    "ignore_above": 256  
                                }  
                            }  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

“**first\_name**” and “**last\_name**” are inner objects - but they should be “**nested**” for our use case



# Nested Types

Yasaswy Raval - 24-Apr-2017 - Capital One

# 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 independence of each object in the array
- The mapping syntax looks like:

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

# Example of nested

```
PUT companies
{
  "mappings": {
    "doc": {
      "properties": {
        "employees": {
          "type": "nested",
          "properties": {
            "first_name": {
              "type": "text",
              "fields": {
                "keyword": {
                  "type": "keyword",
                  "ignore_above": 256
                }
              }
            },
            "last_name": {
              "type": "text",
              "fields": {
                "keyword": {
                  "type": "keyword",
                  "ignore_above": 256
                }
              }
            }
          }
        }
      }
    }
  }
}
```

“`first_name`” and “`last_name`”  
are nested inside “`employees`”

# Nested Objects are Stored Separately

- The nested mapping stores the nested documents in a non-flattened way that maintains the original association:

```
{  
  "name" : "Stark Enterprises"  
}  
  
{  
  "employee.first_name" : "Tony",  
  "employee.last_name" : "Stark"  
}  
  
{  
  "employee.first_name" : "Virginia",  
  "employee.last_name" : "Potts"  
}
```



# Querying a Nested Type

Yasaswy Raval - 24-Apr-2011  
Capital One

# Querying a nested Type

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

```
GET companies/_search
{
  "query": {
    "nested": {
      "path": "employees",
      "query": {
        "bool": {
          "must": [
            {"match": {"employees.first_name": "Tony"}},
            {"match": {"employees.last_name": "Potts"}}
          ]
        }
      }
    }
  }
}
```

Specify the “path” to the nested object

We get 1 hit this time, as expected

# Sorting on a Nested Type

Yasaswy Raval - 24-Apr-2011  
Capital One

# Sorting Nested Queries

- It is possible to sort by the value of a **nested** field
  - even though the value exists in a separate nested document
- Sorting in nested queries is different than non-nested queries
  - we have to repeat the original query in the “**sort**” clause
  - otherwise the sorting happens on **all** documents in the index, not just the ones that hit the **nested** query



# Example of Nested Sorting

- Notice the “match” query is repeated in “sort” in a “nested\_filter” clause:

```
GET companies/_search
{
  "query": {
    "nested": {
      "path": "employees",
      "query": {
        "match": {"employees.last_name": "Potts"}
      }
    }
  },
  "sort": {
    "employees.first_name.keyword" : {
      "order" : "asc",
      "nested_path": "employees",
      "nested_filter" : {
        "match": {"employees.last_name": "Potts"}
      }
    }
  }
}
```



# Result of the Nested Sorting:

```
    "employees": [
      {
        "first_name": "Tony",
        "last_name": "Potts"
      }
    ],
    "sort": [
      "Tony"
    ],
    "employees": [
      {
        "first_name": "Tony",
        "last_name": "Stark"
      },
      {
        "first_name": "Virginia",
        "last_name": "Potts"
      }
    ],
    "sort": [
      "Virginia"
    ]
}
```

Notice the first names are sorted “asc”



# The Nested Aggregation

Yasaswy Raval - 24-Apr-2011, Capita One

# The nested Aggregation

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

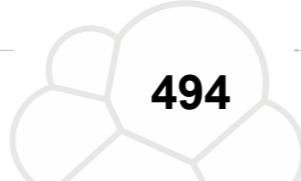
```
GET companies/_search
{
  "size": 0,
  "aggs": {
    "my_employees_bucket": {
      "nested": {
        "path": "employees"
      }
    }
  }
}
```

```
"aggregations": {
  "my_employees_bucket": {
    "doc_count": 3
  }
}
```

# Example of a nested Aggregation

- Suppose we want to put all employees with the same last name into the same bucket:

```
GET companies/_search
{
  "size": 0,
  "aggs": {
    "my_employees_bucket": {
      "nested": {
        "path": "employees"
      },
      "aggs": {
        "last_name": {
          "terms": {
            "field": "employees.last_name.keyword"
          }
        }
      }
    }
  }
}
```



# The result of the nested aggregation:

- We only have two unique last names in our small dataset:

```
"aggregations": {  
    "my_employees_bucket": {  
        "doc_count": 3,  
        "first_name": {  
            "doc_count_error_upper_bound": 0,  
            "sum_other_doc_count": 0,  
            "buckets": [  
                {  
                    "key": "Potts",  
                    "doc_count": 2  
                },  
                {  
                    "key": "Stark",  
                    "doc_count": 1  
                }  
            ]  
        }  
    }  
}
```



# Parent/Child Types

Yasaswy Raval - 24-Apr-2017 - Capital One

# The Need for Parent/Child Types

- Nested objects are great for many use cases
  - But nested objects all live within the same document
  - 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
  - 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 **\_parent** setting



# Parent/Child and Shards

- Behind the scenes, the parent and all its children must live on the same shard
  - This makes query-time joins very fast
- 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 mappings
  - Index some parent documents
  - Index some child documents
  - Query the documents

Yasaswy Raval - 24-Apr-2017 - Capital One

# 1. Define the mappings

- You have to define the parent mapping before any parent documents can be indexed
  - Use the `_parent` property to specify the parent type
  - The following “`employee`” type has “`company`” as its parent:

```
PUT companies
{
  "mappings": {
    "company" : {},
    "employee" : {
      "_parent": {
        "type": "company"
      }
    }
  }
}
```

## 2. Index some parent documents

- A parent is not aware of its children, so indexing a parent document is simply a matter of putting them into Elasticsearch
  - a child can actually be indexed prior to its parent, but we will just index the parents first

```
PUT companies/company/c1
{
  "name" : "Stark Enterprises"
}

PUT companies/company/c2
{
  "name" : "NBC Universal"
}
```

### 3. Index some child documents

- A child document needs to specify its parent when it is indexed
  - Use the “parent” parameter to specify the parent id

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

PUT companies/employee/emp2?parent=c1
{
  "first_name" : "Virginia",
  "last_name" : "Potts"
}

PUT companies/employee/emp3?parent=c2
{
  "first_name" : "Tony",
  "last_name" : "Potts"
}
```

The diagram illustrates the indexing of child documents. Three PUT requests are shown, each specifying a parent ID (c1 or c2). Arrows point from the parent IDs to their respective parent documents: 'Stark Enterprises' for c1 and 'NBC Universal' for c2.

# 4. Query the documents

- Notice that querying the index results in 5 separate documents:

```
GET companies/_search
```



Yashwantrao Ravala - 24-Apr-2017 - Capital One

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

- Typical parent/child queries involve **has\_child** and **has\_parent**

# The has\_child Query

Yasaswy Raval - 24-Apr-2017 - Capital One

# 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:

A diagram illustrating the Elasticsearch search query syntax for the `has_child` query. The query is shown in a grey box:

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

Two teal arrows point from labels to specific parts of the query:

- An arrow points from the label **the parent type** to the URL path segment `/parent`.
- An arrow points from the label **the child type** to the `type: "TYPE"` field.

# Example of has\_child

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

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

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

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

```
  "_source": {
    "name": "Stark Enterprises"
  },
  "inner_hits": {
    "employee": {
      "hits": {
        "total": 1,
        "max_score": 0.6931472,
        "hits": [
          {
            "_source": {
              "first_name": "Tony",
              "last_name": "Stark"
            }
          }
        ]
      }
    }
  }
}
```



# The has\_parent Query

Yasaswy Raval - 24-Apr-2017 - Capital One

# The has\_parent Query

- The **has\_parent** query is executed in the parent document space
  - Returns child documents which associated parents have matched
- The syntax looks like:

the child type

the parent type

```
GET my_index/child/_search
{
  "query": {
    "has_parentTYPE",
      "query": {}
    }
  }
}
```

A diagram illustrating the Elasticsearch search query structure. A grey box contains the query code. Two teal arrows point from teal boxes above and to the right towards specific parts of the code. One arrow points from the text "the child type" to the word "child" in the URL. Another arrow points from the text "the parent type" to the placeholder "TYPE" in the "parent\_type" field of the query block.



# Example of has\_parent

*I want all employees  
that work for “NBC”*

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

```
  "_source": {
    "first_name": "Tony",
    "last_name": "Potts"
  }
```

# Accessing a Child Document

- The Document APIs require the **parent** property for routing purposes
  - A child document is stored on the same shard as its parent
  - The APIs need to know the routing id of the parent to find the child

```
GET companies/employee/emp1
```

```
{  
  "error": {  
    "root_cause": [  
      {  
        "type":  
"routing_missing_exception",  
        "reason": "routing is required for  
[companies]/[employee]/[emp1]",  
        "index_uuid": "_na_",  
        "index": "companies"  
      }  
    ],  
    "type": "routing_missing_exception",  
  }
```

```
GET companies/employee/emp1?parent=c1
```

```
{  
  "_index": "companies",  
  "_type": "employee",  
  "_id": "emp1",  
  "_version": 2,  
  "_routing": "c1",  
  "_parent": "c1",  
  "found": true,  
  "_source": {  
    "first_name": "Tony",  
    "last_name": "Stark"  
  }  
}
```

# Updating Child Documents

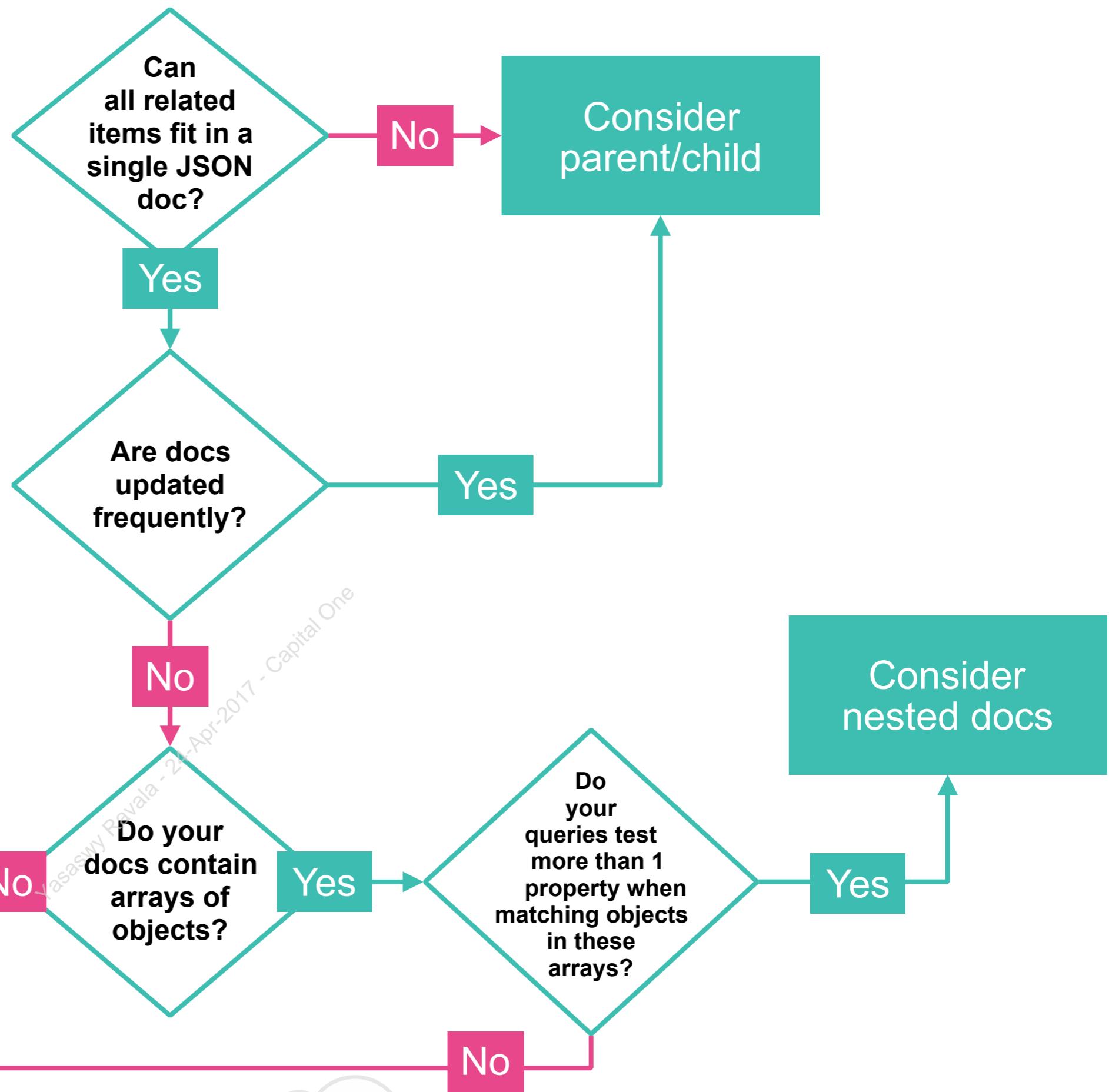
- One of the key benefits of a parent/child relationship is the ability to modify a child object independent of the parent
  - For example, changing an employee has no effect on its parent (or any of its siblings either)

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

Change “Tony” to “Anthony”

# Choosing a Technique

Yasaswy Raval - 24-Apr-2017 - Capital One



# Chapter Review

Yasaswy Raval - 24-Apr-2017 - Capital One

# Summary

- The ***nested*** type allows arrays of objects to be indexed and queried independently of each other
- Sorting by a nested field requires you to repeat the original query in the “**sort**” clause
- The ***nested*** bucket aggregation puts nested objects into a bucket
- 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

# Quiz

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



# Lab 10

## Handling Relationships

Yasaswy Raval - 24-Apr-2017 - Capital One

# Conclusions

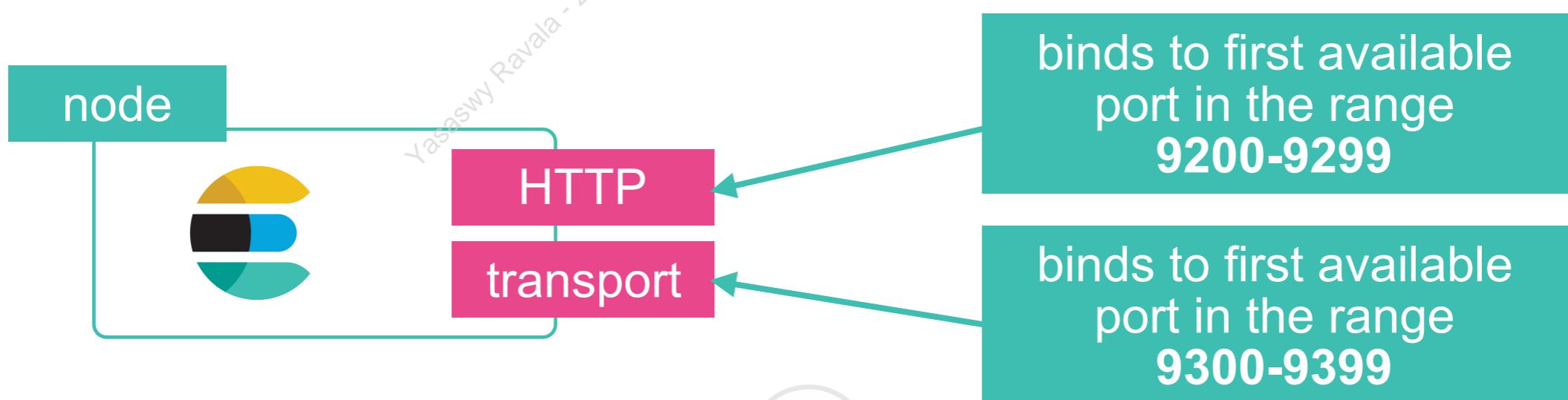
Yasaswy Raval - 24-Apr-2017 - Capital One

# Development vs. Production Mode

Yasaswy Raval - 24-Apr-2017 Capital One

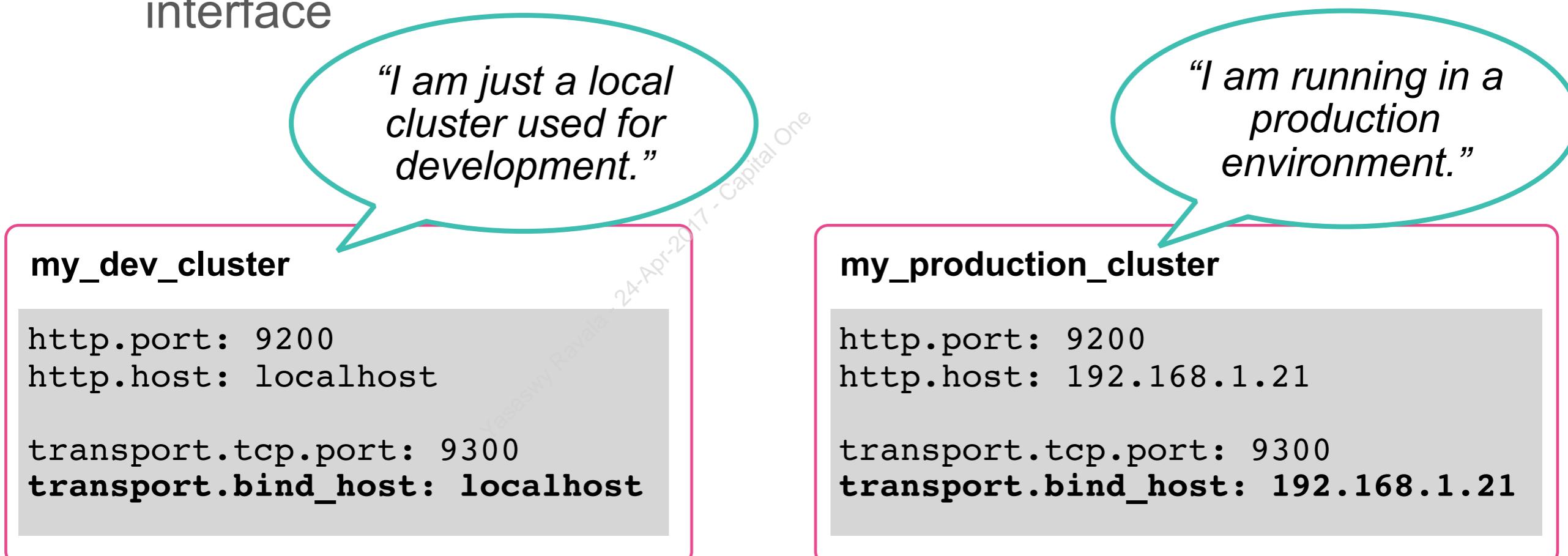
# HTTP vs. Transport

- 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 5.x instance 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>

# Documentation and Help

- Discussion forums: <http://discuss.elastic.co/>
- Meetups: <http://elasticsearch.meetup.com>
- Docs: <https://elastic.co/docs>
- Community: <https://elastic.co/community>
- More resources: <https://elastic.co/learn>

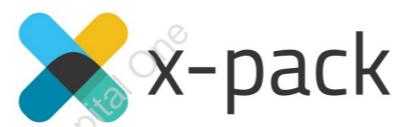
# The Elastic Journey

 beats  
 logstash    kibana  
 elasticsearch

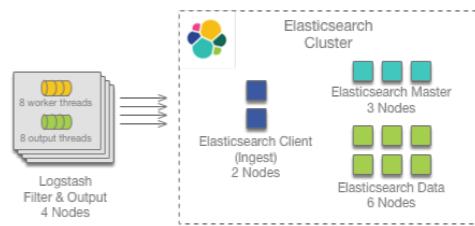
Engineer discovers the power of the Elastic Stack via the open-source community



Engineer engages **TRAINING** to gain individual depth and expertise



Proof-of-concept that delivers real business value with the help of **DEV SUPPORT**



**CONSULTING SERVICES** assists with design and deployment of the production platform



# Quiz Answers

Yasaswy Raval - 24-Apr-2017 - Capital One

# Chapter 1 Quiz Answers

1. True
2. As a **noun**: an index is what Elasticsearch creates for your documents. As a **verb**: we “index” documents into an index
3. True
4. True - much more efficient than separate requests
5. Elasticsearch will define a new index for you
6. No. If you want Elasticsearch to generate an ID, you have to use POST

# Chapter 2 Quiz Answers

1. Multiple **match** terms use “and” or “or”, while multiple terms in **match\_phrase** are “and” and position matters
2. slop
3. now-12h
4. 70% of 5 = 3.5 and it rounds down, so 3 need to match
5. Only the **\_source** of the **my\_index** document, and only fields whose names end in “e”



# Chapter 3 Quiz Answers

1. text analysis
2. Exact values are not analyzed, while full text is analyzed
3. character filter, tokenizer, token filter
4. “happy” is analyzed to “happi”
5. “love” is a synonym for “like”, but “like” is not a synonym for “love”
6. False. you “can” call **\_refresh**, but it is certainly not required.

Yasaswy Raval - 24-Apr-2017 Capital One

# Chapter 4 Quiz Answers

1. “value” would be a “long”
2. The document is not indexed and an error occurs
3. True
4. False: you can not change the data type of any field in a mapping
5. Combines all fields into one big string. You can use `_all` in a query, but keep in mind `_all` will soon be removed from Elasticsearch



# Chapter 5 Quiz Answers

1. 1
2. True: the field being used for suggestions must be of type “completion”
3. A **multi\_match** query
4. We are searching for “java” in both the body and subject. **most\_fields** gives a higher score to documents with “java” in both fields, **best\_fields** returns the score of the better field

Yasaswy Raval - 24-Apr-2017 - Capital One



# Chapter 6 Quiz Answers

1. Document based, using `hash(_id) % num_shards` to determine which shard a doc is routed to
2. Since no settings are defined, the number of shards and replicas default to 5 and 1, respectively, which means 10 shards
3. False - there is nothing to do when adding a new node, except to wait
4.  $12 = 4 \text{ primary shards} + 8 \text{ replicas}$
5. 2

Yasaswy Raval - 24-Apr-2017 Capital One

# Chapter 7 Quiz Answers

1. Skewed data, like what can occur with time-series data; and small datasets that are more likely to have skew
2. True, but only if fielddata is enabled.
3. True
4. `_score`
5. “from” and “size”
6. `scroll` is a snapshot, pagination is “dynamic”

Yasaswy Raval - 24-Apr-2017 Capital One

# Chapter 8 Quiz Answers

1. actually both query and aggregation
2. Query
3. Metrics, bucket, matrix, pipeline
4. terms
5. Increase `shard_size`
6. cardinality

Yasaswy Raval - 24-Apr-2017 - Capital One

# Chapter 9 Quiz Answers

1. Date histogram + terms/filters is probably the simplest solution
2. Significant terms aggregation
3. Use **terms** aggregation for the username buckets, then a **top\_hits** sub-aggregation
4. You could do an aggregation of the tech industry stocks, along with a **global** agg over all stocks

Yasaswy Raval - 24-Apr-2017 - Capital One

# Chapter 10 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 - using the **parent** parameter
5. False
6. True

Yasaswy Raval - 24-Apr-2017 - Capital One

# Thank you!

Please complete the online survey

Yasaswy Raval - 24-Apr-2017 - Capital One

Course: Core Elasticsearch for Developers

Version 5.3.1

© 2015-2017 Elasticsearch BV. All rights reserved. Decompiling, copying, publishing and/or distribution without written consent of Elasticsearch BV is strictly prohibited.

Yasaswy Raval - 24-Apr-2017 - Capital One