elasticsearch

elasticsearch
getting set up

Install Virtualbox

Install Ubuntu

Install Elasticsearch

elasticsearch
system requirements

enable virtualization

Virtualization must be enabled in your BIOS settings. If you have “Hyper-V” virtualization as an option, turn it off.

beware avast

Avast anti-virus is known to conflict with Virtualbox.

let’s do this.

elasticsearch basics.

logical concepts of elasticsearch

documents

Documents are the things you're searching for. They can be more than text – any structured JSON data works. Every document has a unique ID, and a type.

types

A type defines the schema and mapping shared by documents that represent the same sort of thing. (A log entry, an encyclopedia article, etc.)

indices

An index powers search into all documents within a collection of types. They contain inverted indices that let you search across everything within them at once.

what is an inverted index

Document 1:
Space: The final frontier. These are the voyages…

Document 2:
He’s bad, he’s number one. He’s the space cowboy with the laser gun!

Inverted index

space:	1, 2
the:	1, 2
final:	1
frontier:	1
he:	2
bad:	2
…	

# of course it's not quite that simple.

TF-IDF means Term Frequency * Inverse Document Frequency

Term Frequency is how often a term appears in a given document

Document Frequency is how often a term appears in all documents

Term Frequency / Document Frequency measures the relevance of a term in a document

using indices

RESTful API

Elasticsearch fundamenatally works via HTTP requests and JSON data. Any language or tool that can handle HTTP can use Elasticsearch.

client API's

Most languages have specialized Elasticsearch libraries to make it even easier.

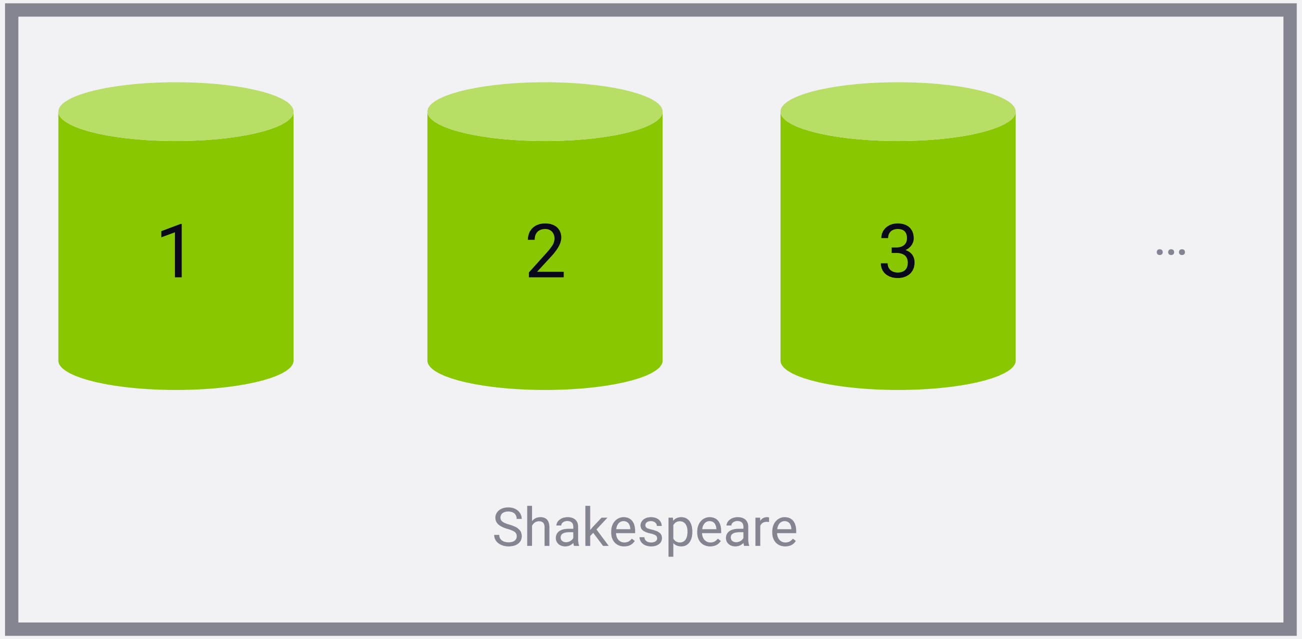
analytic tools

Web-based graphical UI's such as Kibana let you interact with your indices and explore them without writing code.

how elasticsearch scales

an index is split into shards.

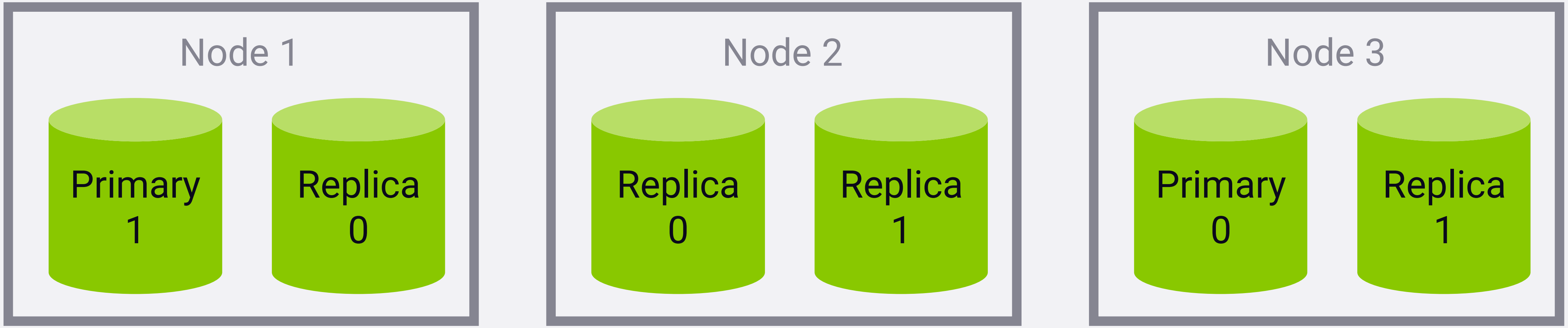
Documents are hashed to a particular shard.



Each shard may be on a different node in a cluster.
Every shard is a self-contained Lucene index of its own.

primary and replica shards

This index has two primary shards and two replicas.
Your application should round-robin requests amongst nodes.



Write requests are routed to the primary shard, then replicated
Read requests are routed to the primary or any replica

The number of primary shards cannot be changed later.

Not as bad as it sounds – you can add more replica shards for more read throughput.

Worst case you can re-index your data.

The number of shards can be set up front via a PUT command via REST / HTTP

```
PUT /testindex
{
 "settings": {
   "number_of_shards": 3
   , "number_of_replicas": 1
 }
}
```

quiz time

The schema for your documents are defined by…

- The index
- The type
- The document itself

01

The schema for your documents are defined by…

- The index
- The type
- The document itself

01

02

What purpose do inverted indices serve?

- They allow you search phrases in reverse order
- They quickly map search terms to documents
- They load balance search requests across your cluster

What purpose do inverted indices serve?

- They allow you search phrases in reverse order
- They quickly map search terms to documents
- They load balance search requests across your cluster

02

03

- 8
- 15
- 20

An index configured for 5 primary shards and 3 replicas would have how many shards in total?

03

- 8
- 15
- 20

An index configured for 5 primary shards and 3 replicas would have how many shards in total?

04

- true
- false

Elasticsearch is built only for full-text search of documents.

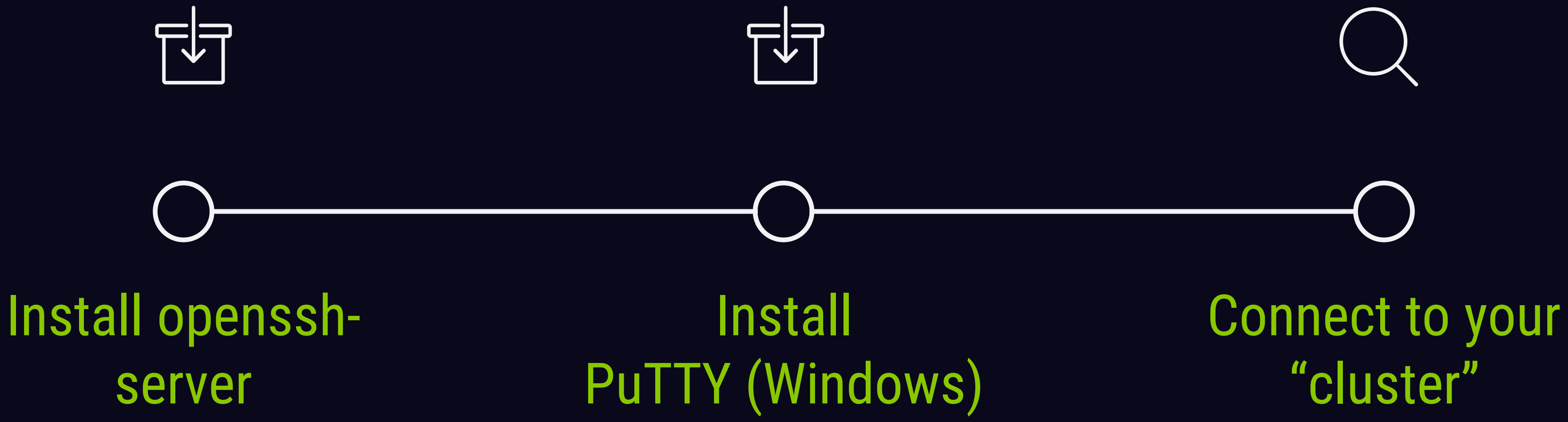
04

- true
- false

Elasticsearch is built only for full-text search of documents.

connecting to your cluster_

elasticsearch
more setup



Install openssh-server

Install PuTTY (Windows)

Connect to your “cluster”

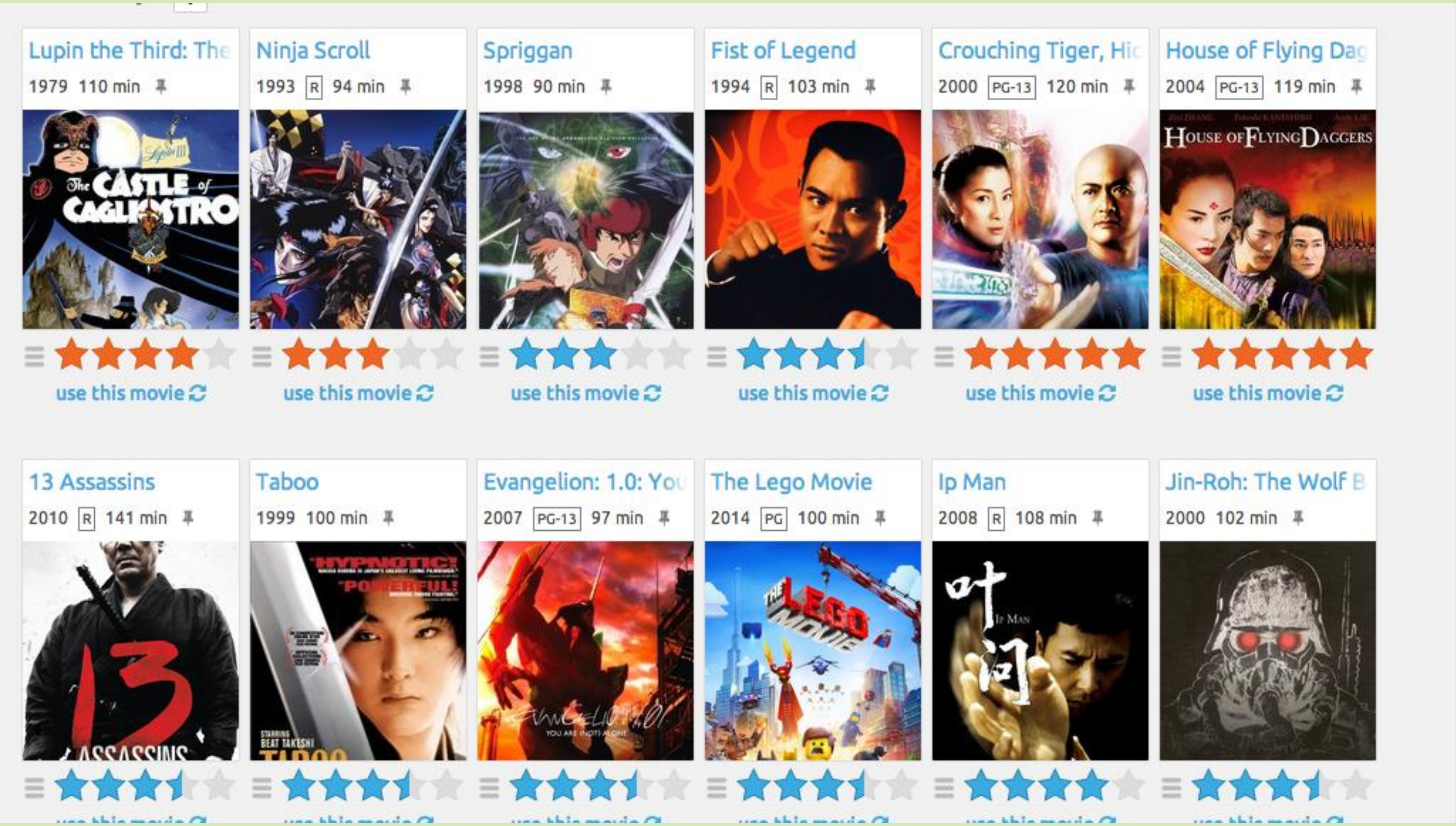
examining movielens

movielens

movielens is a free dataset of movie ratings gathered from movielens.org.

It contains user ratings, movie metadata, and user metadata.

Let’s download and examine the data files from movielens.org



creating mappings

what is a mapping?

a mapping is a schema definition.
elasticsearch has reasonable defaults, but sometimes you need to customize them.

```
curl -XPUT 127.0.0.1:9200/movies -d '
{
        "mappings": {
                "movie": {
                        "_all": {"enabled": false},
                        "properties" : {
                                "year" : {“type": "date"}
                        }
                }
        }
}'
```

common mappings

field types

string, byte, short, integer, long, float, double, boolean, date

```
“properties”: {
   “user_id” : {
       “type”: “long”
    }
}
```

field index

do you want this field indexed for full-text search? analyzed / not_analyzed / no

```
“properties”: {
   “genre” : {
      “index”: “not_analyzed”
    }
}
```

field analyzer

define your tokenizer and token filter. standard / whitespace / simple / english etc.

```
“properties”: {
   “description” : {
      “analyzer”: “english”
    }
}
```

more about analyzers

character filters

remove HTML encoding, convert & to and

tokenizer

split strings on whitespace / punctuation / non-letters

token filter

lowercasing, stemming, synonyms, stopwords

choices for analyzers

standard

splits on word boundaries, removes punctuation, lowercases. good choice if language is unknown

simple

splits on anything that isn't a letter, and lowercases

whitespace

splits on whitespace but doesn't lowercase

language (i.e. english)

accounts for language-specific stopwords and stemming

import
one document

insert

```
curl -XPUT
127.0.0.1:9200/movies/movie/109487 -d '
{
"genre" : ["IMAX","Sci-Fi"],
"title" : "Interstellar",
"year" : 2014
}'
```

A FILM BY CHRISTOPHER NOLAN
INTERSTELLAR
THE END OF EARTH WILL NOT BE THE END OF US.
WARNER BROS. PICTURES AND PARAMOUNT PICTURES PRESENT
IN ASSOCIATION WITH LEGENDARY PICTURES A SYNCOPY/LYNDA OBST PRODUCTIONS PRODUCTION A FILM BY CHRISTOPHER NOLAN "INTERSTELLAR"

import many documents

json bulk import

```
curl -XPUT 127.0.0.1:9200/_bulk -d ‘
```

```
{ "create" : { "_index" : "movies", "_type" : "movie", "_id" : "135569" } }
{ "id": "135569", "title" : "Star Trek Beyond", "year":2016 , "genre":["Action", "Adventure", "Sci-Fi"] }
{ "create" : { "_index" : "movies", "_type" : "movie", "_id" : "122886" } }
{ "id": "122886", "title" : "Star Wars: Episode VII - The Force Awakens", "year":2015 , "genre":["Action", "Adventure", "Fantasy", "Sci-Fi", "IMAX"] }
{ "create" : { "_index" : "movies", "_type" : "movie", "_id" : "109487" } }
{ "id": "109487", "title" : "Interstellar", "year":2014 , "genre":["Sci-Fi", "IMAX"] }
{ "create" : { "_index" : "movies", "_type" : "movie", "_id" : "58559" } }
{ "id": "58559", "title" : "Dark Knight, The", "year":2008 , "genre":["Action", "Crime", "Drama", "IMAX"] }
{ "create" : { "_index" : "movies", "_type" : "movie", "_id" : "1924" } }
{ "id": "1924", "title" : "Plan 9 from Outer Space", "year":1959 , "genre":["Horror", "Sci-Fi"] }  `
```

updating documents

versions

Every document has a _version field
Elasticsearch documents are immutable.
When you update an existing document:

- a new document is created with an incremented _version
- the old document is marked for deletion

partial update api

```
curl -XPOST 127.0.0.1:9200/movies/movie/109487/_update -d '
{
    "doc": {
           "title": "Interstellar"
    }
}'
```

deleting documents

it couldn't be easier.

Just use the DELETE method:

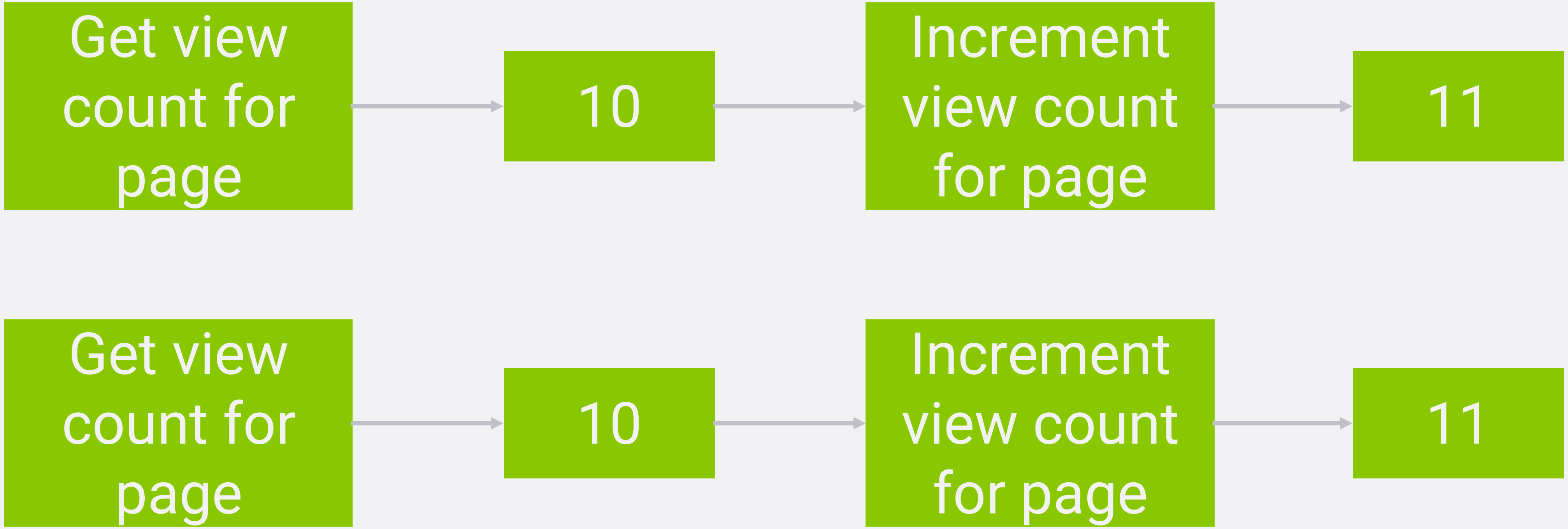
```
curl -XDELETE 127.0.0.1:9200/movies/movie/58559
```

exercise

insert, update, and then **delete** a movie of your choice into the movies index!

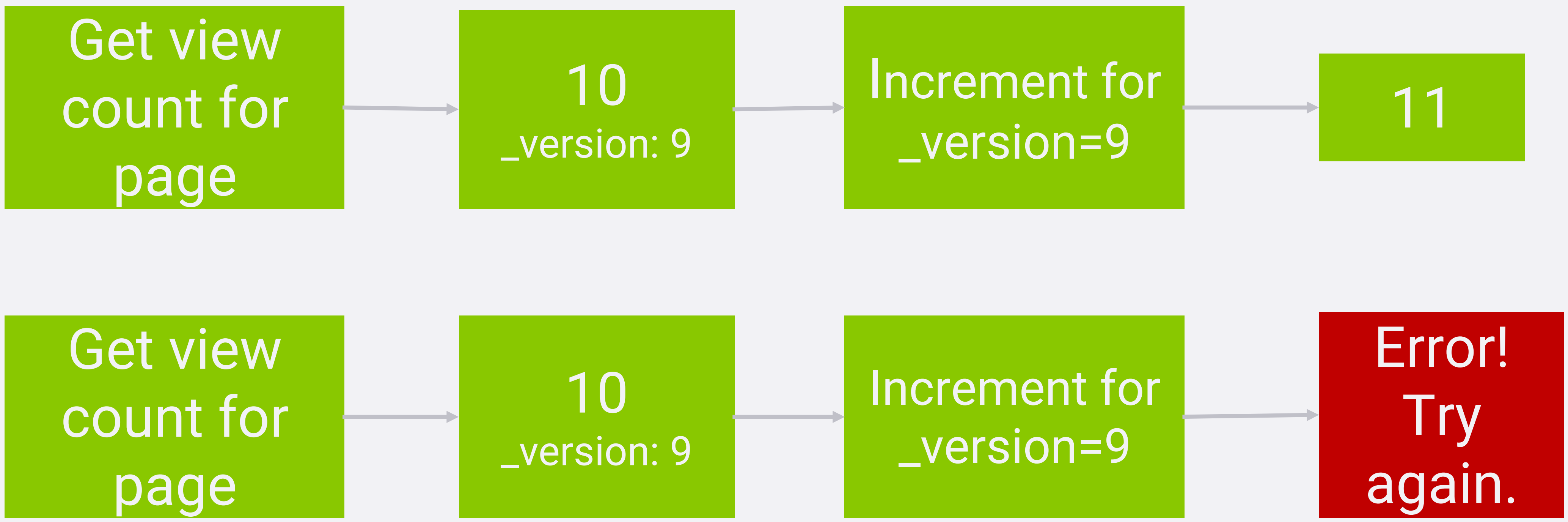
dealing with concurrency

the problem



But it should be 12!

optimistic concurrency control



Get view count for page → 10 _version: 9 → Increment for _version=9 → 11

Get view count for page → 10 _version: 9 → Increment for _version=9 → Error! Try again.

Use retry_on_conflicts=N to automatically retry.

controlling full-text search

using analyzers

sometimes text fields should be exact-match

- use no_analyzer mapping

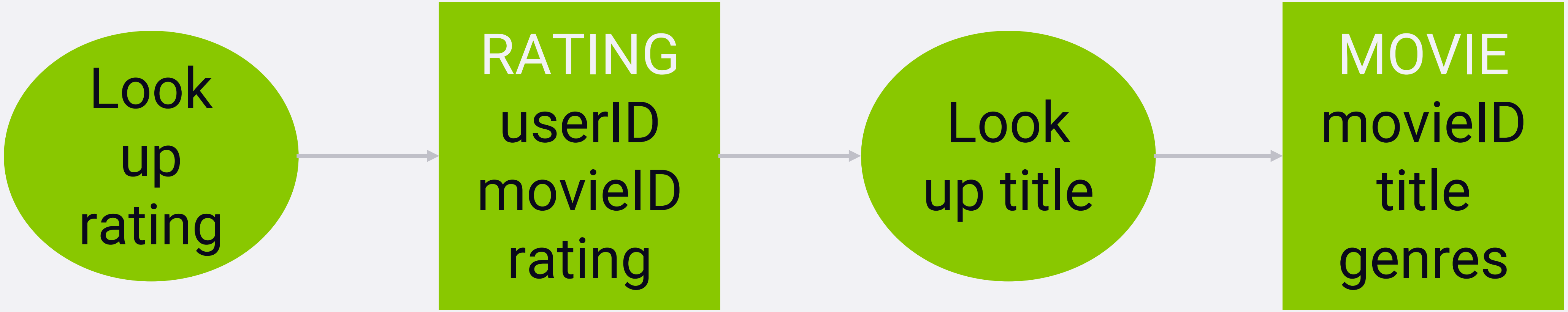
search on analyzed fields will return anything remotely relevant

- depending on the analyzer, results will be case-insensitive, stemmed, stopwords removed, synonyms applied, etc.
- searches with multiple terms need not match them all

data modeling

strategies for relational data

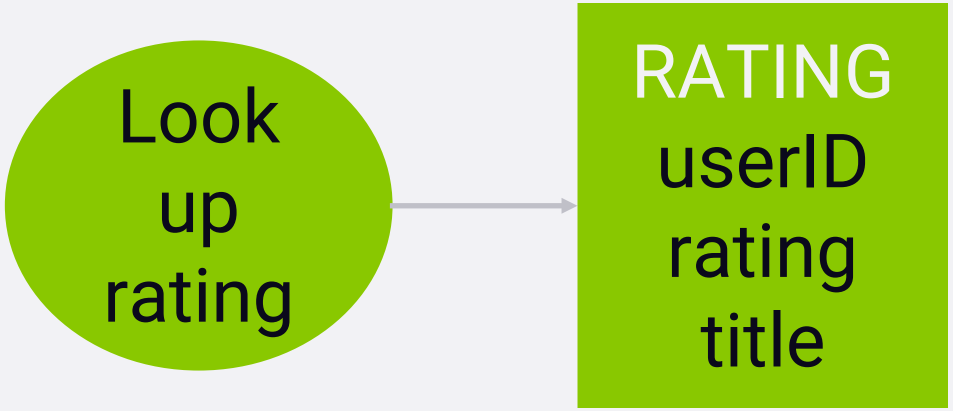
normalized data



Minimizes storage space, makes it easy to change titles
But requires two queries, and storage is cheap!

strategies for relational data

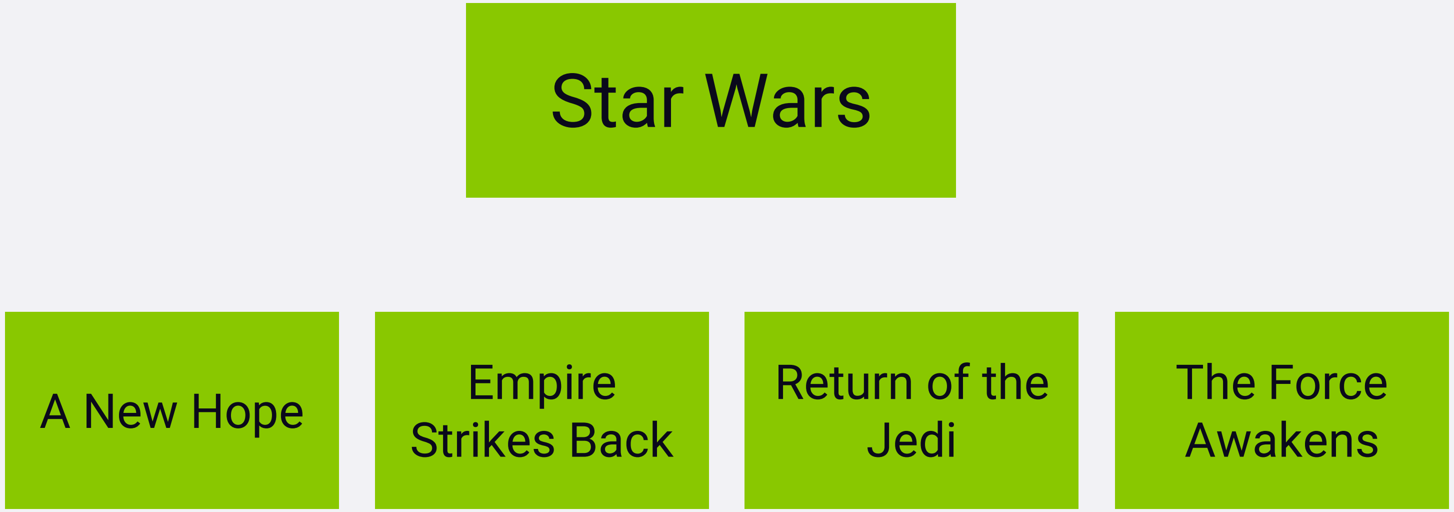
denormalized data



titles are duplicated, but only one query

strategies for relational data

Parent / Child Relationship



Star Wars

A New Hope

Empire Strikes Back

Return of the Jedi

The Force Awakens

query-line search

“query lite”

/movies/movie/_search?q=title:star

/movies/movie/_search?q=+year:>2010+title:trek

it’s not always simpler.

spaces etc. need to be URL encoded.

```
/movies/movie/_search?q=+year:>2010+title:trek
```

↓

```
/movies/movie/_search?q=%2Byear%3A%3E2010+%2Btitle%3Atrek
```

and it can be dangerous.

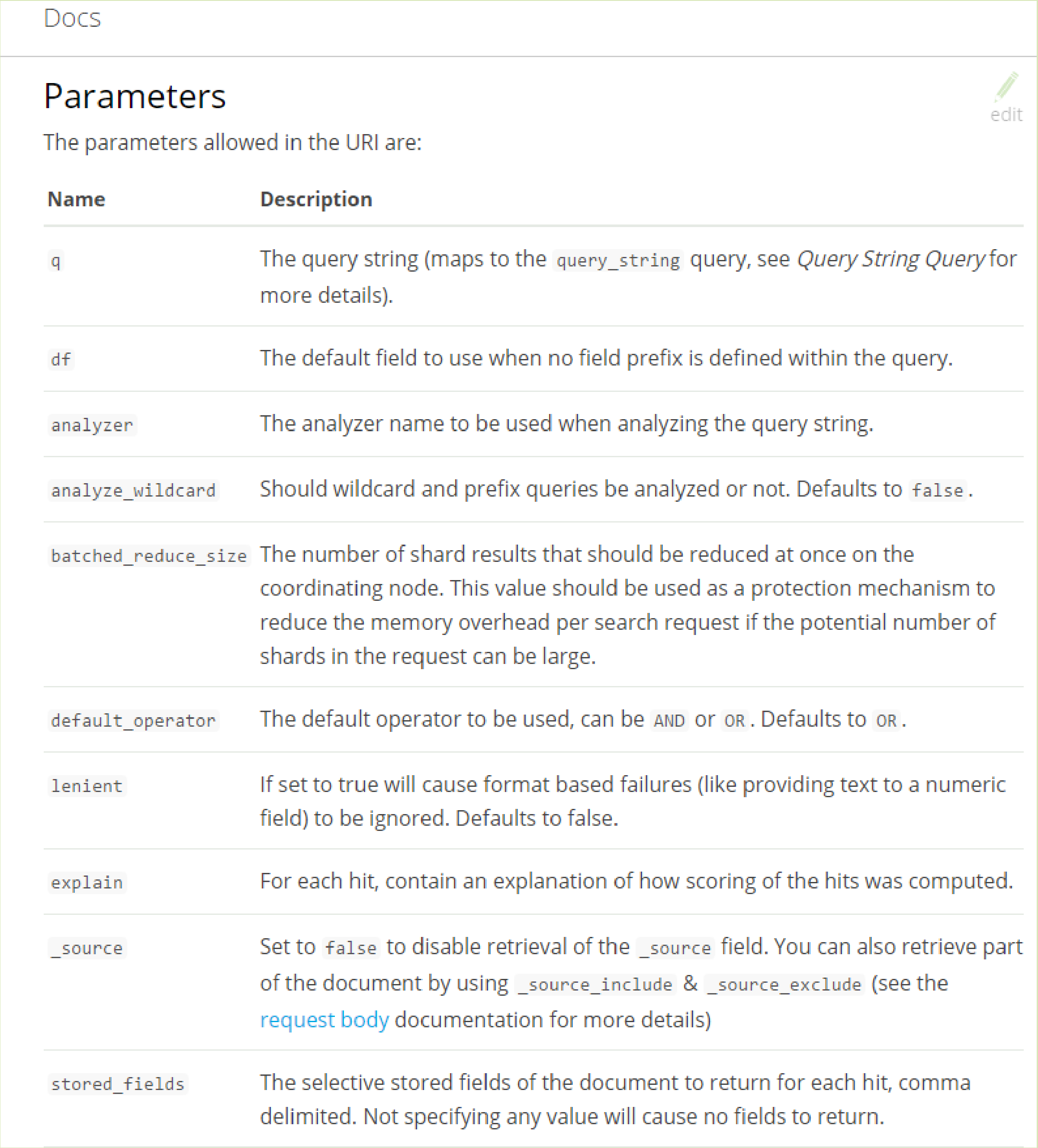
- cryptic and tough to debug
- can be a security issue if exposed to end users
- fragile – one wrong character and you're hosed.

But it's handy for quick experimenting.

learn more.

this is formally called “URI Search”. Search for that on the Elasticsearch documentation.

it’s really quite powerful, but again is only appropriate for quick “curl tests”.



Parameters

The parameters allowed in the URI are:

Name	Description
`q`	The query string (maps to the `query_string` query, see *Query String Query* for more details).
`df`	The default field to use when no field prefix is defined within the query.
`analyzer`	The analyzer name to be used when analyzing the query string.
`analyze_wildcard`	Should wildcard and prefix queries be analyzed or not. Defaults to `false`.
`batched_reduce_size`	The number of shard results that should be reduced at once on the coordinating node. This value should be used as a protection mechanism to reduce the memory overhead per search request if the potential number of shards in the request can be large.
`default_operator`	The default operator to be used, can be `AND` or `OR`. Defaults to `OR`.
`lenient`	If set to true will cause format based failures (like providing text to a numeric field) to be ignored. Defaults to false.
`explain`	For each hit, contain an explanation of how scoring of the hits was computed.
`_source`	Set to `false` to disable retrieval of the `_source` field. You can also retrieve part of the document by using `_source_include` & `_source_exclude` (see the request body documentation for more details)
`stored_fields`	The selective stored fields of the document to return for each hit, comma delimited. Not specifying any value will cause no fields to return.

request body search

request body search

how you’re supposed to do it

query DSL is in the request body as JSON

(yes, a GET request can have a body!)

```
curl -XGET 127.0.0.1:9200/movies/movie/_search?pretty -d '
{
        "query": {
                "match": {
                        "title": "star"
                }
        }
}'
```

queries and filters

filters ask a yes/no question of your data
queries return data in terms of relevance

use filters when you can – they are faster and cacheable.

example: boolean query with a filter

```
curl -XGET 127.0.0.1:9200/movies/movie/_search?pretty -d'
{
        "query":{
                "bool": {
                        "must": {"term": {"title": "trek"}},
                        "filter": {"range": {"year": {"gte": 2010}}}
                }
        }
}'
```

some types of filters

term: filter by exact values

```
{“term”: {“year”: 2014}}
```

terms: match if any exact values in a list match

```
{“terms”: {“genre”: [“Sci-Fi”, “Adventure”] } }
```

range: Find numbers or dates in a given range (gt, gte, lt, lte)

```
{“range”: {“year”: {“gte”: 2010}}}
```

exists: Find documents where a field exists

```
{“exists”: {“field”: “tags”}}
```

missing: Find documents where a field is missing

```
{“missing”: {“field”: “tags”}}
```

bool: Combine filters with Boolean logic (must, must_not, should)

some types of queries

match_all: returns all documents and is the default. Normally used with a filter.

```
        {“match_all”: {}}
```

match: searches analyzed results, such as full text search.

```
        {“match”: {“title”: “star”}}
```

multi_match: run the same query on multiple fields.

```
        {“multi_match”: {“query”: “star”, “fields”: [“title”, “synopsis” ] } }
```

bool: Works like a bool filter, but results are scored by relevance.

syntax reminder

queries are wrapped in a “query”: { } block,
filters are wrapped in a “filter”: { } block.

you can combine filters inside queries, or queries inside filters too.

```
curl -XGET 127.0.0.1:9200/movies/movie/_search?pretty -d'
{
        "query":{
                "bool": {
                        "must": {"term": {"title": "trek"}},
                        "filter": {"range": {"year": {"gte": 2010}}}
                }
        }
}'
```

phrase search

phrase matching

must find all terms, in the right order.

```
curl -XGET 127.0.0.1:9200/movies/movie/_search?pretty -d '
{
        "query": {
                "match_phrase": {
                        "title": "star wars"
                }
        }
}'
```

slop

order matters, but you're OK with some words being in between the terms:

```
curl -XGET 127.0.0.1:9200/movies/movie/_search?pretty -d '
{
        "query": {
                "match_phrase": {
                "title": {"query": "star beyond", "slop": 1}
                }
        }
}'
```

the slop represents how far you're willing to let a term move to satisfy a phrase (in either direction!)

another example: "quick brown fox" would match "quick fox" with a slop of 1.

proximity queries

remember this is a query – results are sorted by relevance.

just use a really high slop if you want to get any documents that contain the words in your phrase, but want documents that have the words closer together scored higher.

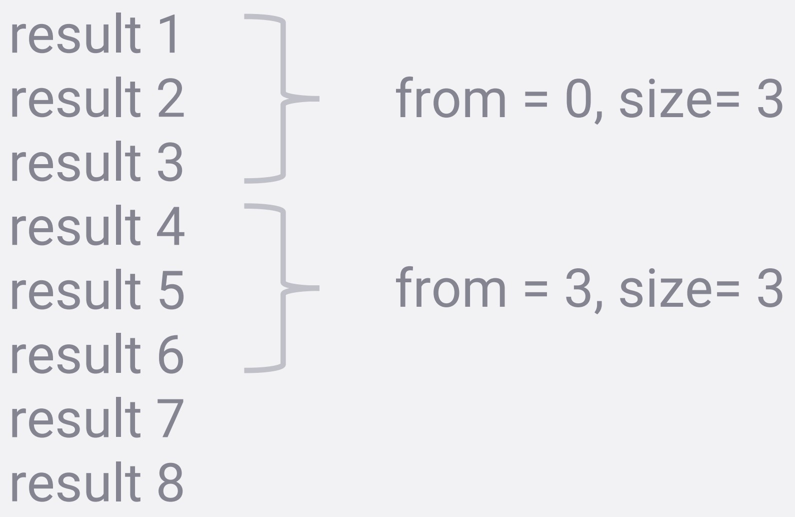
```
curl -XGET 127.0.0.1:9200/movies/movie/_search?pretty -d '
{
        "query": {
                "match_phrase": {
                "title": {"query": "star beyond", "slop": 100}
                }
        }
}'
```

exercise

search for "Star Wars" movies released after 1980, using both a **URI search** and a **request body search.**

pagination

specify "from" and "size"



result 1
result 2
result 3
result 4
result 5
result 6
result 7
result 8

from = 0, size= 3

from = 3, size= 3

pagination syntax

```
curl -XGET '127.0.0.1:9200/movies/movie/_search?size=2&from=2&pretty'

curl -XGET 127.0.0.1:9200/movies/movie/_search?pretty -d'
{
        "from": 2,
        "size": 2,
        "query": {"match": {"genre": "Sci-Fi"}}
}'
```

beware

deep pagination can kill performance.

every result must be retrieved, collected, and sorted.

enforce an upper bound on how many results you'll return to users.

sorting

sorting your results is usually quite simple.

```
curl -XGET '127.0.0.1:9200/movies/movie/_search?sort=year&pretty'
```

unless you're dealing with strings.

A string field that is analyzed for full-text search can't be used to sort documents

This is because it exists in the inverted index as individual terms, not as the entire string.

If you need to sort on an analyzed field, map a not_analyzed copy.

```
curl -XPUT 127.0.0.1:9200/movies/ -d '
{
   "mappings": {
            "movie": {
                "_all": {"enabled": false},
                "properties" : {
                   "title": {
                       "type" : "string",
                       "fields": {
                                 "raw": {
                                       "type": "string",
                                       "index": "not_analyzed"
                                   }
                          }
                   }
                }
            }
        }
   }'
```

Now you can sort on the not_analyzed “raw” field.

```
curl -XGET '127.0.0.1:9200/movies/movie/_search?sort=title.raw&pretty'
```

sadly, you cannot change the mapping on an existing index.

you’d have to delete it, set up a new mapping, and re-index it.

like the number of shards, this is something you should think about before importing data into your index.

more with filters

another filtered query

```
curl -XGET 127.0.0.1:9200/movies/_search?pretty -d'
{
        "query":{
                "bool": {
                        "must": {"match": {"genre": "Sci-Fi"}},
                        "must_not": {"match": {"title": "trek"}},
                        "filter": {"range": {"year": {"gte": 2010, "lt": 2015}}}
                }
        }
}'
```

exercise

search for science fiction movies before 1960, sorted by title.

fuzziness

fuzzy matches

a way to account for typos and misspellings

the levenshtein edit distance accounts for:

- substitutions of characters (interstellar -> intersteller)
- insertions of characters (interstellar -> insterstellar)
- deletion of characters (interstellar -> interstelar)

all of the above have an edit distance of 1.

the fuzziness parameter

```
curl -XGET 127.0.0.1:9200/movies/movie/_search?pretty -d '
{
        "query": {
                "fuzzy": {
                        "title": {"value": "intrsteller", "fuzziness": 2}
                }
        }
}'
```

AUTO fuzziness

fuzziness: AUTO

- 0 for 1-2 character strings
- 1 for 3-5 character strings
- 2 for anything else

partial matching

prefix queries on strings

If we remapped year to be a string…

```
curl -XGET '127.0.0.1:9200/movies/movie/_search?pretty' -d '
{
        "query": {
                "prefix": {
                        "year": "201"
                }
        }
}'
```

wildcard queries

```
curl -XGET '127.0.0.1:9200/movies/movie/_search?pretty' -d '
{
	"query": {
		"wildcard": {
			"year": "1*"
		}
	}
}'
```

“regexp” queries also exist.

search as you type

query-time search-as-you-type

abusing sloppiness…

```
curl -XGET '127.0.0.1:9200/movies/movie/_search?pretty' -d '
{
        "query": {
                "match_phrase_prefix": {
                        "title": {
                                "query": "star trek",
                                "slop": 10
                        }
                }
        }
}'
```

index-time with N-grams

“star”:

unigram:	[s, t, a, r]
bigram:	[st, ta, ar]
trigram:	[sta, tar]
4-gram:	[star]

edge n-grams are built only on the beginning of each term.

indexing n-grams

1. Create an “autocomplete” analyzer

```
curl -XPUT '127.0.0.1:9200/movies?pretty' -d '
{
        "settings": {
                "analysis": {
                        "filter": {
                                "autocomplete_filter": {
                                        "type": "edge_ngram",
                                        "min_gram": 1,
                                        "max_gram": 20
                                }
                        },
                        "analyzer": {
                                "autocomplete": {
                                        "type": "custom",
                                        "tokenizer": "standard",
                                        "filter": [
                                                "lowercase",
                                                "autocomplete_filter"
                                        ]
                                }
                        }
                }
        }
}'
```

now map your field with it

```
curl -XPUT '127.0.0.1:9200/movies/_mapping/movie?pretty' -d '
{
       "movie": {
                 "properties" : {
                          "title": {
                                   "type" : "string",
                                    "analyzer": "autocomplete"
                           }
                 }
         }
  }'
```

but only use n-grams on the index side!

```
curl -XGET 127.0.0.1:9200/movies/movie/_search?pretty -d '
{
        "query": {
                "match": {
                        "title": {
                                "query": "sta",
                                "analyzer": "standard"
                        }
                }
        }
}'
```

otherwise our query will also get split into n-grams, and we’ll get results for everything that matches ‘s’, ‘t’, ‘a’, ‘st’, etc.

completion suggesters

You can also upload a list of all possible completions ahead of time using completion suggesters.

importing data

you can import from just about anything

stand-alone scripts can submit bulk documents via REST API

logstash and beats can stream data from logs, S3, databases, and more

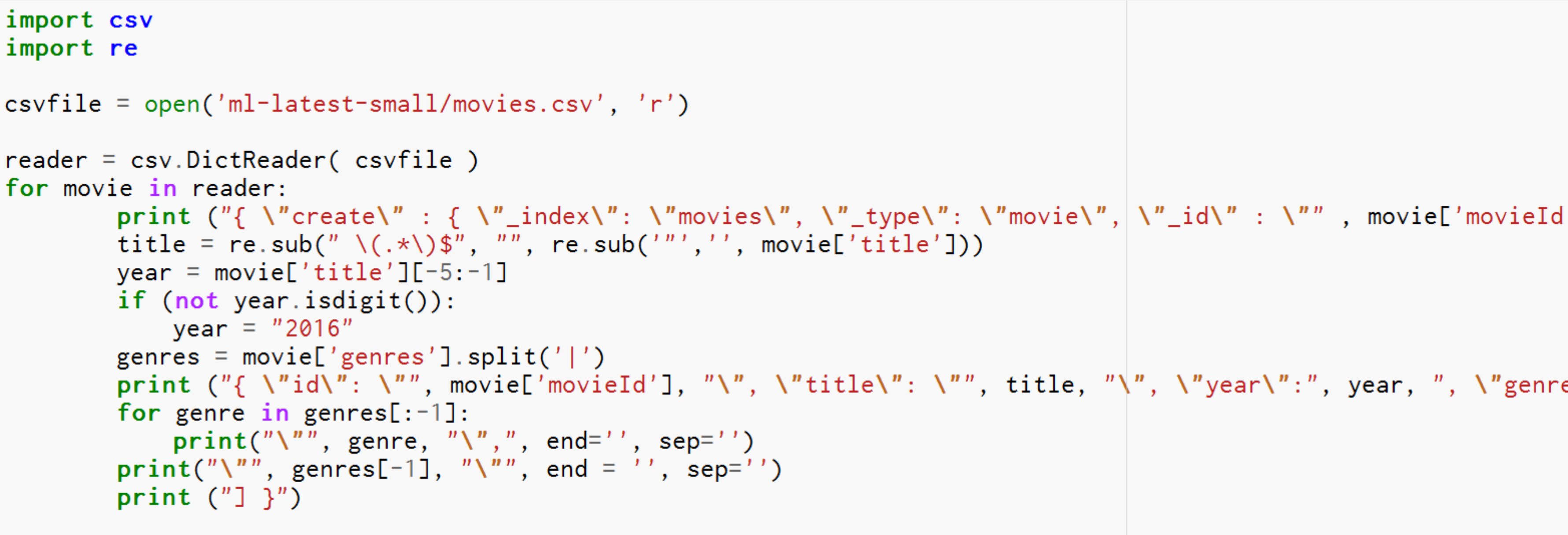
AWS systems can stream in data via lambda or kinesis firehose

kafka, spark, and more have Elasticsearch integration add-ons

importing via script / json

hack together a script

- read in data from some distributed filesystem
- transform it into JSON bulk inserts
- submit via HTTP / REST to your elasticsearch cluster



```
import csv
import re

csvfile = open('ml-latest-small/movies.csv', 'r')

reader = csv.DictReader( csvfile )
for movie in reader:
        print ("{ \"create\" : { \"_index\": \"movies\", \"_type\": \"movie\", \"_id\" : \"" , movie['movieId
        title = re.sub(" \(.*\)$", "", re.sub('"','', movie['title']))
        year = movie['title'][-5:-1]
        if (not year.isdigit()):
            year = "2016"
        genres = movie['genres'].split('|')
        print ("{ \"id\": \"", movie['movieId'], "\", \"title\": \"", title, "\", \"year\":", year, ", \"genre
        for genre in genres[:-1]:
            print("\"", genre, "\",", end='', sep='')
        print("\"", genres[-1], "\"", end = '', sep='')
        print ("] }")
```

for completeness:

```
import csv
import re

csvfile = open('ml-latest-small/movies.csv', 'r')

reader = csv.DictReader( csvfile )
for movie in reader:
        print ("{ \"create\" : { \"_index\": \"movies\", \"_type\": \"movie\", \"_id\" : \"" , movie['movieId'], "\" } }", sep='')
        title = re.sub(" \(.*\)$", "", re.sub('"','', movie['title']))
        year = movie['title'][-5:-1]
        if (not year.isdigit()):
            year = "2016"
        genres = movie['genres'].split('|')
        print ("{ \"id\": \"", movie['movieId'], "\", \"title\": \"", title, "\", \"year\":", year, ", \"genre\":[", end='', sep='')
        for genre in genres[:-1]:
            print("\"", genre, "\",", end='', sep='')
        print("\"", genres[-1], "\"", end = '', sep='')
        print ("] }")
```

importing
via client api's

a less hacky script.

free elasticsearch client libraries are available for pretty much any language.

- java has a client maintained by elastic.co
- python has an elasticsearch package
- elasticsearch-ruby
- several choices for scala
- elasticsearch.pm module for perl

You don’t have to wrangle JSON.

```
es = elasticsearch.Elasticsearch()

es.indices.delete(index="ratings",ignore=404)
deque(helpers.parallel_bulk(es,readRatings(),index="ratings",doc_ty
es.indices.refresh()
```

for completeness:

```
import csv
from collections import deque
import elasticsearch
from elasticsearch import helpers

def readMovies():
  csvfile = open('ml-latest-small/movies.csv', 'r')

  reader = csv.DictReader( csvfile )

  titleLookup = {}

  for movie in reader:
      titleLookup[movie['movieId']] = movie['title']

  return titleLookup

def readRatings():
  csvfile = open('ml-latest-small/ratings.csv', 'r')

  titleLookup = readMovies()

  reader = csv.DictReader( csvfile )
  for line in reader:
    rating = {}
    rating['user_id'] = int(line['userId'])
    rating['movie_id'] = int(line['movieId'])
    rating['title'] = titleLookup[line['movieId']]
    rating['rating'] = float(line['rating'])
    rating['timestamp'] = int(line['timestamp'])
    yield rating


es = elasticsearch.Elasticsearch()

es.indices.delete(index="ratings",ignore=404)
deque(helpers.parallel_bulk(es,readRatings(),index="ratings",doc_type="rating"), maxlen=0)
es.indices.refresh()
```

exercise

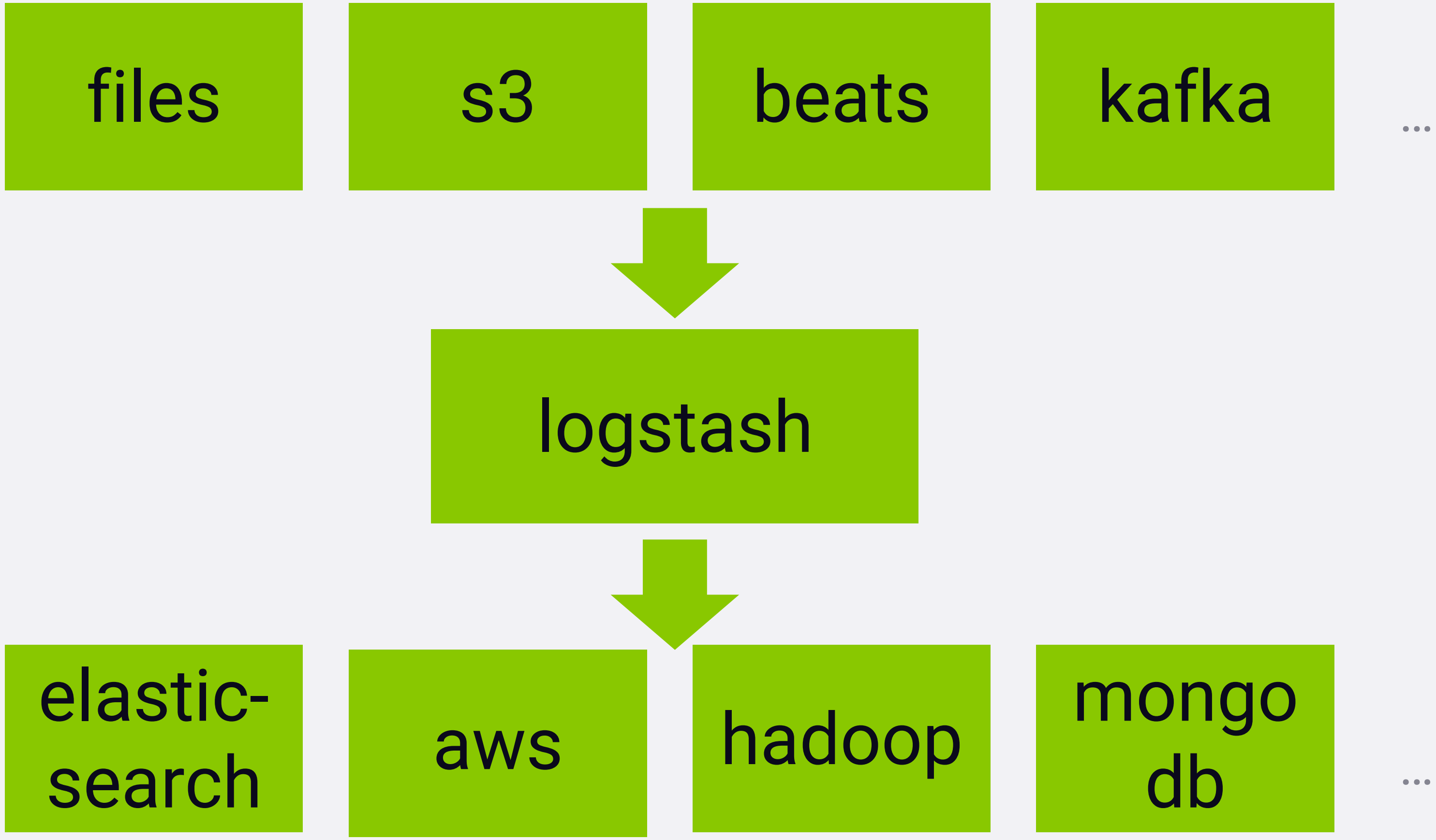
write a script to import the tags.csv data from ml-latest-small into a new "tags" index.

my solution

```
import csv
from collections import deque
import elasticsearch
from elasticsearch import helpers

def readMovies():
    csvfile = open('ml-latest-small/movies.csv', 'r')

    reader = csv.DictReader( csvfile )

    titleLookup = {}

    for movie in reader:
            titleLookup[movie['movieId']] = movie['title']

    return titleLookup

def readTags():
    csvfile = open('ml-latest-small/tags.csv', 'r')

    titleLookup = readMovies()

    reader = csv.DictReader( csvfile )
    for line in reader:
        tag = {}
        tag['user_id'] = int(line['userId'])
        tag['movie_id'] = int(line['movieId'])
        tag['title'] = titleLookup[line['movieId']]
        tag['tag'] = line['tag']
        tag['timestamp'] = int(line['timestamp'])
        yield tag


es = elasticsearch.Elasticsearch()

es.indices.delete(index="tags",ignore=404)
deque(helpers.parallel_bulk(es,readTags(),index="tags",doc_type="tag"), maxlen=0)
es.indices.refresh()
```

introducing logstash

what logstash is for



files

s3

beats

kafka

...

logstash

elastic-search

aws

hadoop

mongo db

...

it’s more than plumbing

- logstash parses, transforms, and filters data as it passes through.
- it can derive structure from unstructured data
- it can anonymize personal data or exclude it entirely
- it can do geo-location lookups
- it can scale across many nodes
- it guarantees at-least-once delivery
- it absorbs throughput from load spikes

See https://www.elastic.co/guide/en/logstash/current/filter-plugins.html for the huge list of filter plugins.

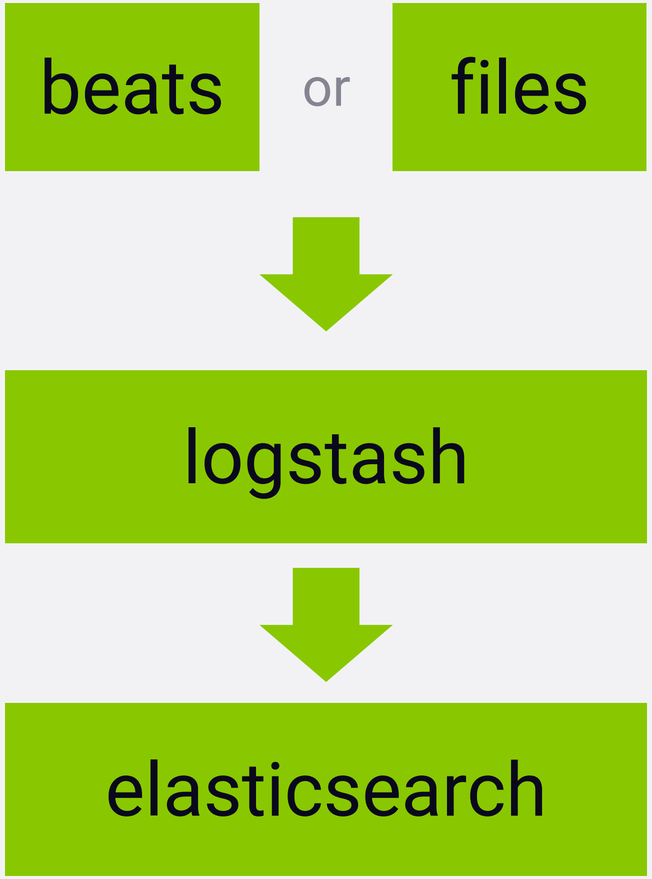
huge variety of input source events

elastic beats – cloudwatch – couchdb – drupal – elasticsearch – windows event log – shell output – local files – ganglia – gelf – gemfire – random generator – github – google pubsub – graphite – heartbeats – heroku – http – imap – irc – jdbc – jmx – kafka – lumberjack – meetup – command pipes – puppet – rabbitmq – rackspace cloud queue – redis – relp – rss – s3 – salesforce – snmp – sqlite – sqs – stdin – stomp – syslog – tcp – twitter – udp – unix sockets – varnish log – websocket – wmi – xmpp – zenoss – zeromq

huge variety of output "stash" destinations

boundary – circonus – cloudwatch – csv – datadoghq – elasticsearch – email – exec – local file – ganglia – gelf – bigquery – google cloud storage – graphite – graphtastic – hipchat – http – influxdb – irc – jira – juggernaut – kafka – librato – loggly – lumberjack – metriccatcher – mongodb – nagios – new relic insights – opentsdb – pagerduty – pipe to stdin – rabbitmq – rackspace cloud queue – redis – redmine – riak – riemann – s3 – sns – solr – sqs – statsd – stdout – stomp – syslog – tcp – udp – webhdfs – websocket – xmpp – zabbix - zeromq

typical usage



Web logs

Parse into structured fields, geolocate

installing logstash

installing logstash

```
sudo apt-get update
sudo apt-get install logstash
```

configuring logstash

```
sudo vi /etc/logstash/conf.d/logstash.conf
```

```
input {
    file {
        path => "/home/fkane/access_log“
        start_position => "beginning"
        ignore_older => 0
    }
}

filter {
    grok {
        match => { "message" => "%{COMBINEDAPACHELOG}" }
    }
    date {
        match => [ "timestamp", "dd/MMM/yyyy:HH:mm:ss Z" ]
    }
}

output {
    elasticsearch {
        hosts => ["localhost:9200"]
    }
    stdout {
        codec => rubydebug
    }
}
```

running logstash

```
cd /usr/share/logstash/

sudo bin/logstash -f /etc/logstash/conf.d/logstash.conf
```

logstash
with mysql

install a jdbc driver

get a mysql connector from https://dev.mysql.com/downloads/connector/j/

```
wget https://dev.mysql.com/get/Downloads/Connector-J/mysql-connector-java-5.1.42.zip

unzip mysql-connector-java-5.1.42.zip
```

configure logstash

```
input {
    jdbc {
        jdbc_connection_string => "jdbc:mysql://localhost:3306/movielens"
        jdbc_user => "root"
        jdbc_password => “password"
        jdbc_driver_library => "/home/fkane/mysql-connector-java-5.1.42/mysql-connector-java-5.1.42-bin.jar"
        jdbc_driver_class => "com.mysql.jdbc.Driver"
        statement => "SELECT * FROM movies"
    }
}
```

logstash with s3

what is s3

amazon web services’ simple storage service

cloud-based distributed storage system

integration is easy-peasy.

```
input {
        s3 {
                bucket => "sundog-es"
                access_key_id => "AKIAIS****C26Y***Q"
                secret_access_key => "d*****FENOXcCuNC4iTbSLbibA*****eyn****"
        }
}
```

logstash with kafka

what is kafka

- apache kafka
- open-source stream processing platform
- high throughput, low latency
- publish/subscribe
- process streams
- store streams

has a lot in common with logstash, really.

integration is easy-peasy.

```
input {
	kafka {
		bootstrap_servers => "localhost:9092"
		topics => ["kafka-logs"]
	}
}
```

elasticsearch with spark

what is apache spark

- “a fast and general engine for large-scale data processing”
- a faster alternative to mapreduce
- spark applications are written in java, scala, python, or r
- supports sql, streaming, machine learning, and graph processing

flink is nipping at spark’s heels, and can also integrate with elasticsearch.

integration with elasticsearch-spark

```
./spark-2.1.1-bin-hadoop2.7/bin/spark-shell --packages org.elasticsearch:elasticsearch-spark-20_2.11:5.4.3

import org.elasticsearch.spark.sql._

case class Person(ID:Int, name:String, age:Int, numFriends:Int)

def mapper(line:String): Person = {
        val fields = line.split(',')
        val person:Person = Person(fields(0).toInt, fields(1), fields(2).toInt, fields(3).toInt)
        return person
}

import spark.implicits._
val lines = spark.sparkContext.textFile("fakefriends.csv")
val people = lines.map(mapper).toDF()

people.saveToEs("spark/people")
```

elasticsearch

exercise

write spark code that imports movie ratings from ml-latest-small into a “spark” index with a “ratings” type.

integration with elasticsearch-spark

```
./spark-2.1.1-bin-hadoop2.7/bin/spark-shell --packages org.elasticsearch:elasticsearch-spark-20_2.11:5.4.3

import org.elasticsearch.spark.sql._

case class Person(ID:Int, name:String, age:Int, numFriends:Int)

def mapper(line:String): Person = {
	val fields = line.split(',')
	val person:Person = Person(fields(0).toInt, fields(1), fields(2).toInt, fields(3).toInt)
	return person
}

import spark.implicits._
val lines = spark.sparkContext.textFile("fakefriends.csv")
val people = lines.map(mapper).toDF()

people.saveToEs("spark/people")
```

dealing with the header line

```
val header = lines.first()
val data = lines.filter(row => row != header)
```

my solution

```
import org.elasticsearch.spark.sql._

case class Rating(userID:Int, movieID:Int, rating:Float, timestamp:Int)

def mapper(line:String): Rating= {
        val fields = line.split(',')
        val rating:Rating = Rating(fields(0).toInt, fields(1).toInt, fields(2).toFloat, fields(3).toInt)
        return rating
}

import spark.implicits._
val lines = spark.sparkContext.textFile("ml-latest-small/ratings.csv")
val header = lines.first()
val data = lines.filter(row => row != header)
val ratings= data.map(mapper).toDF()

ratings.saveToEs("spark/ratings")
```

aggregations

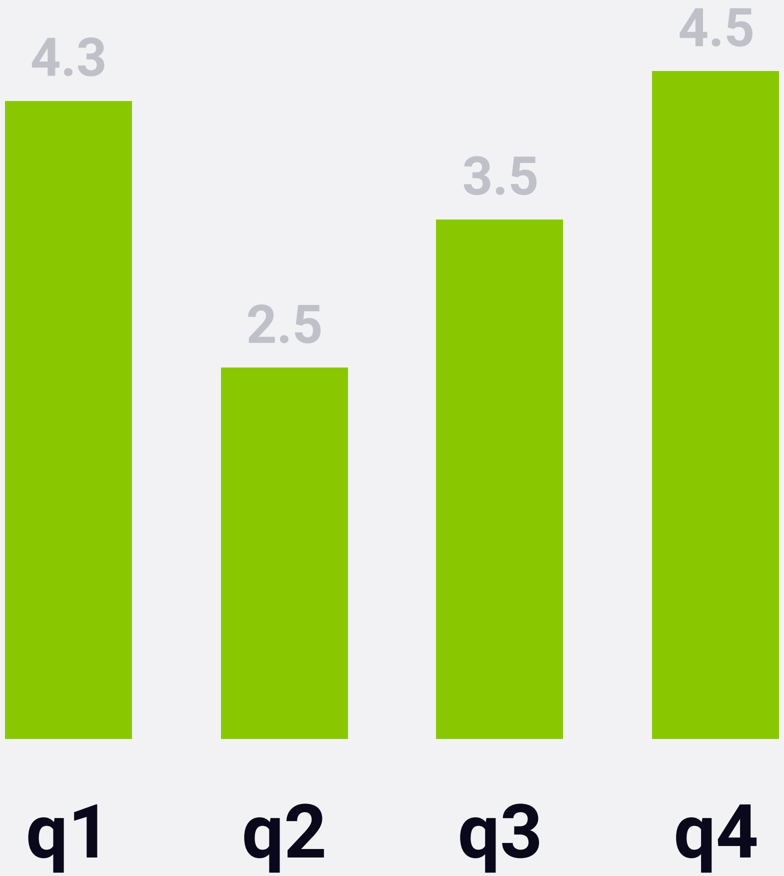
it's not just for search anymore

metrics

average, stats, min/max, percentiles, etc.

buckets

histograms, ranges, distances, significant terms, etc.



pipelines

moving average, average bucket, cumulative sum, etc.

matrix

matrix stats

aggregations
are amazing

elasticsearch aggregations can sometimes take the place of hadoop / spark / etc – and return results instantly!

it gets better

you can even nest aggregations together!

let’s learn
by example

bucket by rating value:

```
curl -XGET
'127.0.0.1:9200/ratings/rating/_search?size=0&pretty' -d ‘
{
        "aggs": {
                "ratings": {
                        "terms": {
                                "field": "rating"
                        }
                }
        }
}'
```

let’s learn
by example

count only 5-star ratings:

```
curl -XGET
'127.0.0.1:9200/ratings/rating/_search?size=0&pretty' -d ‘
{
        "query": {
                "match": {
                        "rating": 5.0
                }
        },
        "aggs" : {
                "ratings": {
                        "terms": {
                                "field" : "rating"
                        }
                }
        }
}'
```

let’s learn
by example

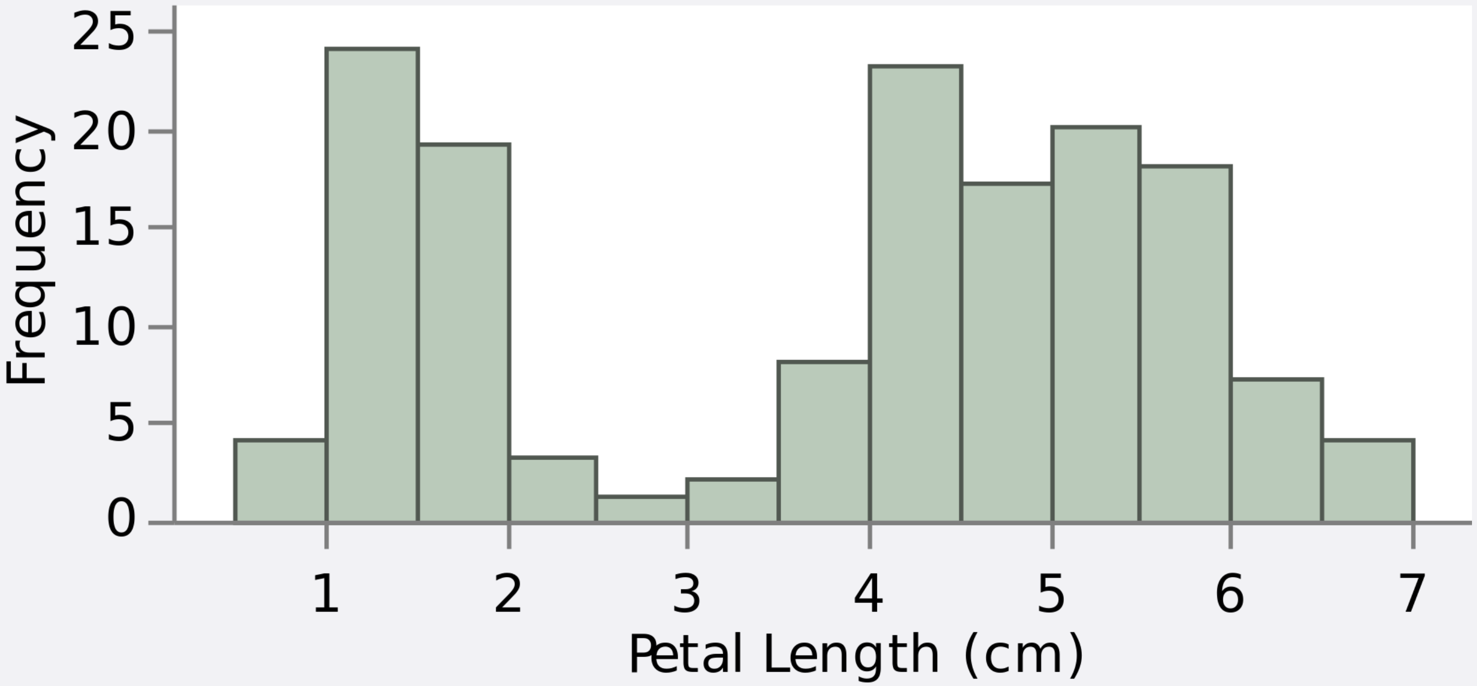
average rating for Star Wars:

```
curl -XGET
'127.0.0.1:9200/ratings/rating/_search?size=0&pretty' -d ‘
{
        "query": {
                "match_phrase": {
                        "title": "Star Wars Episode IV"
                }
        },
        "aggs" : {
                "avg_rating": {
                        "avg": {
                                "field" : "rating"
                        }
                }
        }
}'
```

histograms

what is a histogram

display totals of documents bucketed by some interval range



display ratings by 1.0-rating intervals

```
curl -XGET
'127.0.0.1:9200/ratings/rating/_search?size=0&pretty' -d ‘
{
        "aggs" : {
                "whole_ratings": {
                        "histogram": {
                                "field": "rating",
                                "interval": 1.0
                        }
                }
        }
}'
```

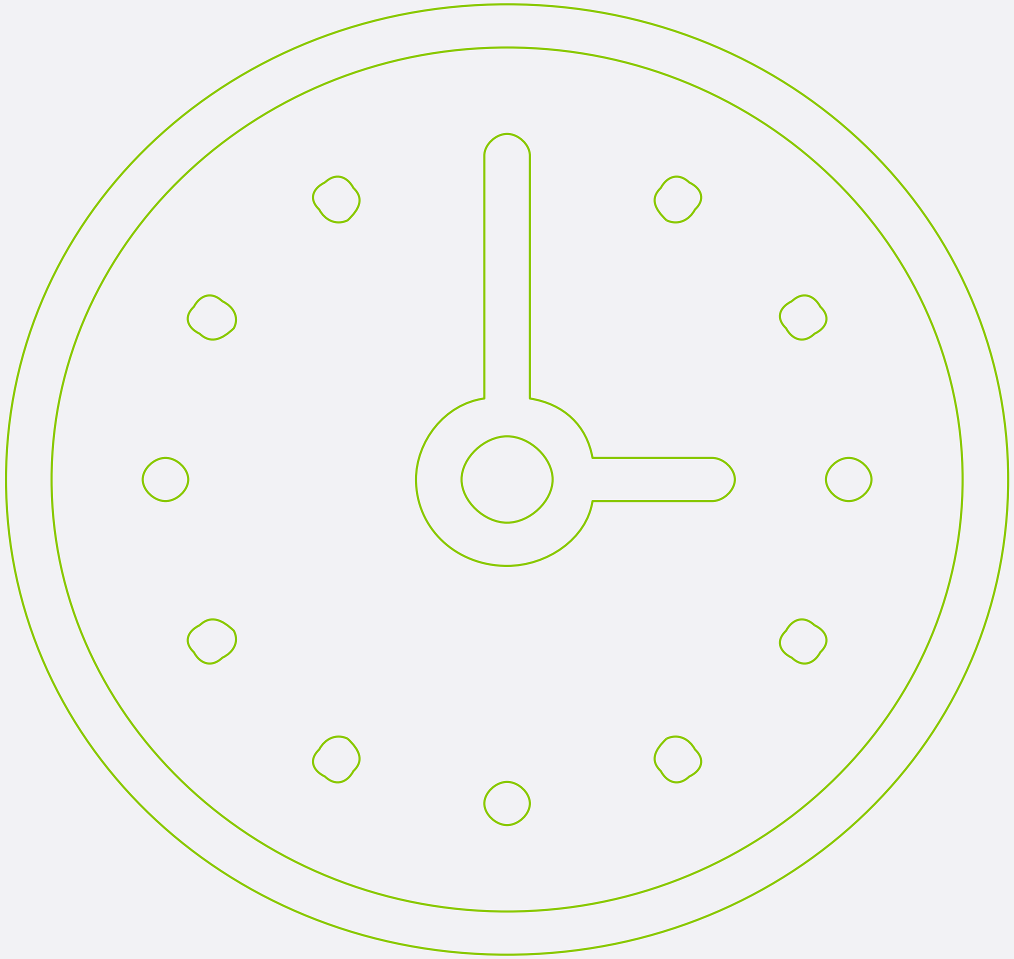
count up movies from each decade

```
curl -XGET
'127.0.0.1:9200/movies/movie/_search?size=0&pretty' -d ‘
{
        "aggs" : {
                "release": {
                        "histogram": {
                                "field": "year",
                                "interval": 10
                        }
                }
        }
}
```

time series

dealing with time

Elasticsearch can bucket and aggregate fields that contain time and dates properly. You can aggregate by “year” or “month” and it knows about calendar rules.



break down website hits by hour:

```
curl -XGET '127.0.0.1:9200/logstash-
2015.12.04/logs/_search?size=0&pretty' -d ‘
{
        "aggs" : {
                "timestamp": {
                        "date_histogram": {
                                "field": "@timestamp",
                                "interval": "hour"
                        }
                }
        }
}'
```

when does google scrape me?

```
curl -XGET '127.0.0.1:9200/logstash-
2015.12.04/logs/_search?size=0&pretty' -d ‘
{
        "query" : {
                "match": {
                        "agent": "Googlebot"
                }
        },
        "aggs" : {
                “timestamp": {
                        "date_histogram": {
                                "field": "@timestamp",
                                "interval": "hour"
                        }
                }
        }
}'
```

exercise

when did my site go down on december 4, 2015? (bucket 500 status codes by the minute in logstash-2015.12.04/logs)

my solution

```
GET /logstash-2015.12.04/logs/_search?size=0&pretty
{
        "query" : {
                "match": {
                        "response": "500"
                }
        },

        "aggs" : {
                "timestamp": {
                        "date_histogram": {
                                "field": "@timestamp",
                                "interval": "minute"
                        }
                }
        }
}
```

nested aggregations

nested aggregations

Aggregations can be nested for more powerful queries.

For example, what's the average rating for each Star Wars movie?

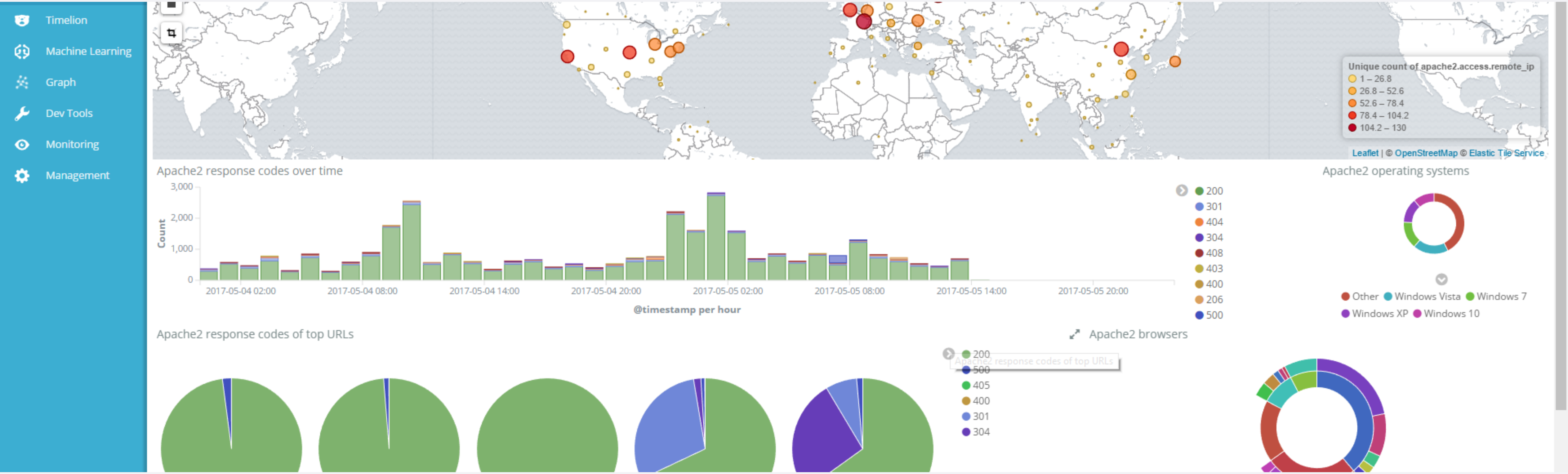
Let's undertake this as an activity – and show you what can go wrong along the way.

for reference, here’s the final query

```
curl -XGET '127.0.0.1:9200/ratings/rating/_search?size=0&pretty' -d ‘
{
        "query": {
                "match_phrase": {
                        "title": "Star Wars"
                }
        },
        "aggs" : {
                "titles": {
                        "terms": {
                                "field": "title.raw"
                        },
                        "aggs": {
                                "avg_rating": {
                                        "avg": {
                                                "field" : "rating"
                                        }
                                }
                        }
                }
        }
}'
```

using kibana

what is
kibana



installing kibana

```
sudo apt-get install kibana
sudo vi /etc/kibana/kibana.yml
        change server.host to 0.0.0.0
        add xpack.security.enabled: false

sudo /bin/systemctl daemon-reload
sudo /bin/systemctl enable kibana.service
sudo /bin/systemctl start kibana.service
```

kibana is now available on port 5601

playing with kibana

let’s analyze the works of william shakespeare…

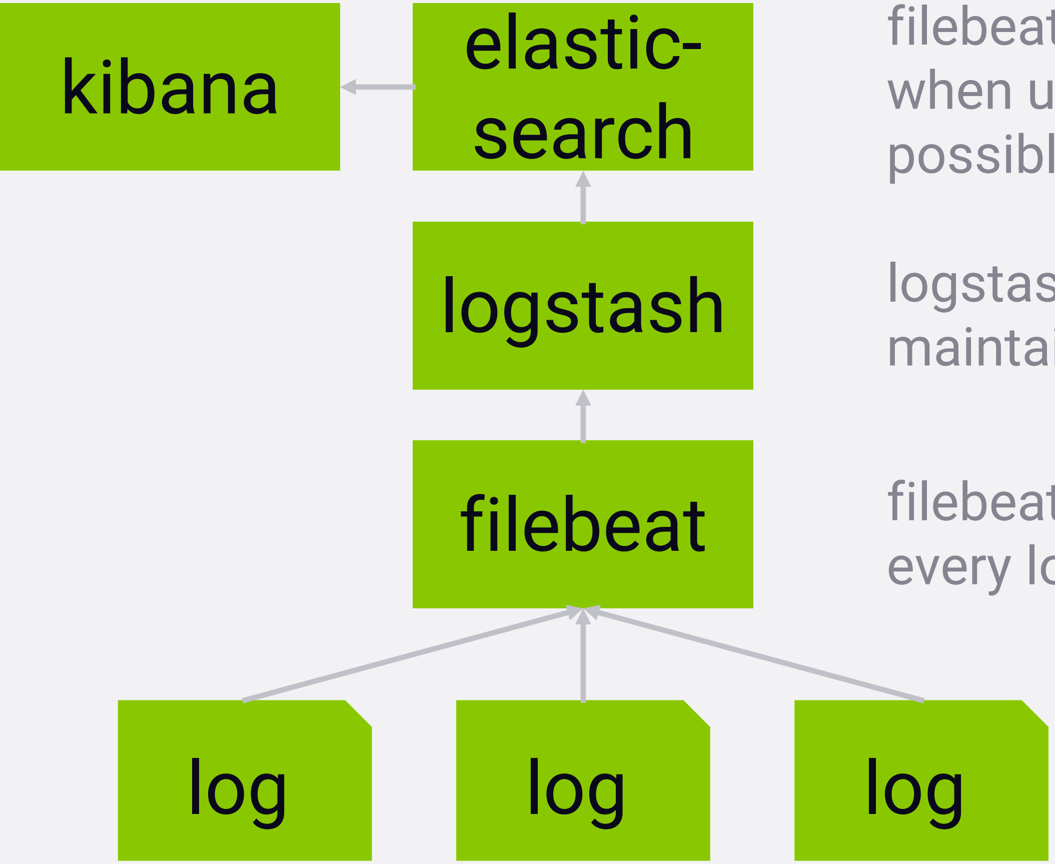
because we can.

exercise

find the longest shakespeare plays – create a vertical bar chart that aggregates the count of documents by play name in descending order.

using filebeat

filebeat is a lightweight shipper for logs



filebeat can optionally talk directly to elasticsearch. when using logstash, elasticsearch is just one of many possible destinations!

logstash and filebeat can communicate to maintain “backpressure” when things back up

filebeat maintains a read pointer on the logs. every log line acts like a queue.

logs can be from apache, nginx, auditd, or mysql

this is called the elastic stack

prior to beats, you'd hear about the "ELK stack" – elasticsearch, logstash, kibana.

why use filebeat and logstash and not just one or the other?

- it won’t let you overload your pipeline.
- you get more flexibility on scaling your cluster.

installing filebeat

installing and testing filebeat

```
  sudo apt-get update && sudo apt-get install filebeat

  cd /usr/share/elasticsearch/
  sudo bin/elasticsearch-plugin install ingest-geoip
  sudo bin/elasticsearch-plugin install ingest-user-agent
  sudo /bin/systemctl stop elasticsearch.service
  sudo /bin/systemctl start elasticsearch.service

  sudo vi /etc/filebeat/filebeat.yml

Comment out existing log section, add at the bottom:

filebeat.modules:
- module: apache2
 access:
   var.paths: ["/home/fkane/logs/access*"]
 error:
   var.paths: ["/home/fkane/logs/error*"]

cd /usr/share/filebeat
sudo scripts/import_dashboards
sudo /bin/systemctl stop kibana.service
sudo /bin/systemctl start kibana.service

  Make /home/<username>/logs
  cd into it
  wget http://media.sundog-soft.com/es/access_log
  sudo /bin/systemctl start filebeat.service
```

analyzing logs with kibana

exercise

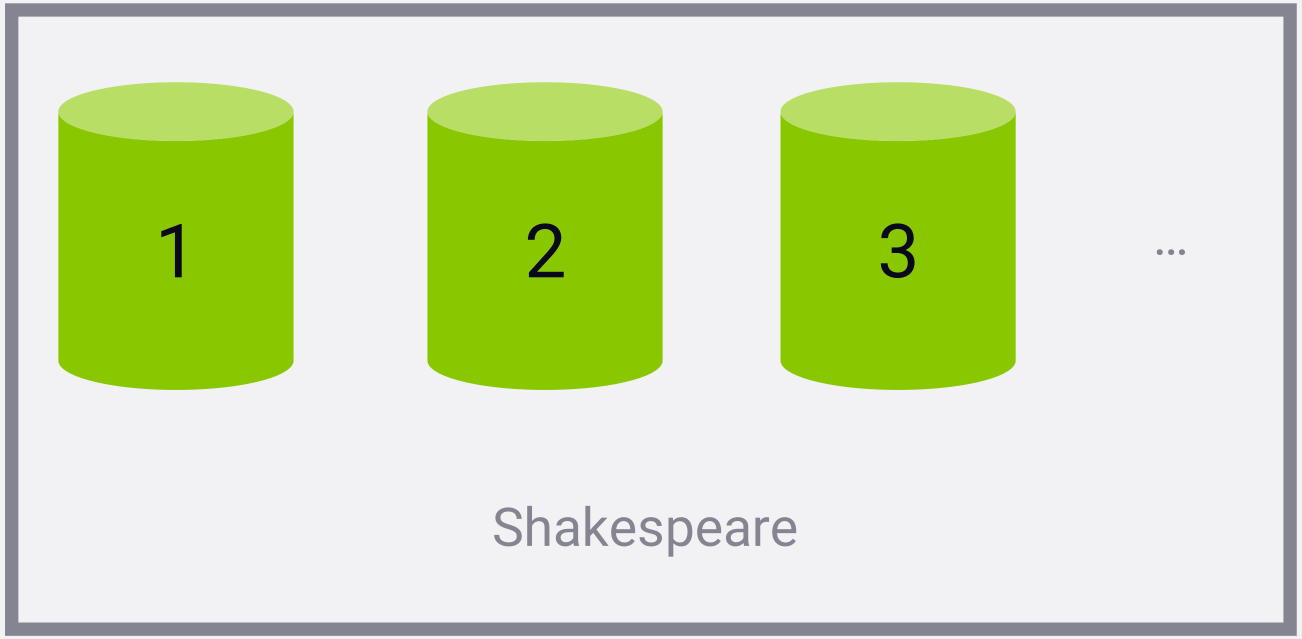
between 9:30 – 10:00 AM on May 4, 2017, which cities were generating 404 errors?

elasticsearch operations

choosing your shards

an index is split into shards.

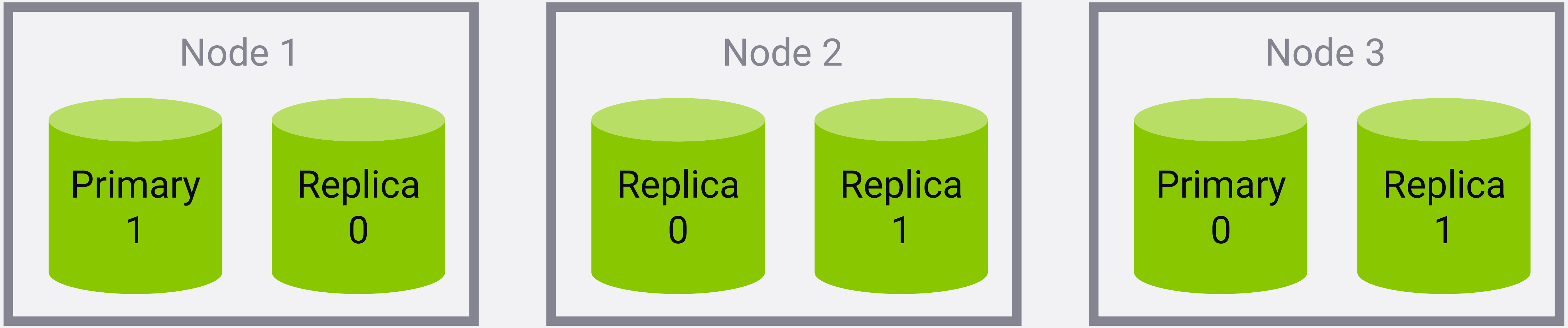
Documents are hashed to a particular shard.



Each shard may be on a different node in a cluster.
Every shard is a self-contained Lucene index of its own.

primary and replica shards

This index has two primary shards and two replicas.
Your application should round-robin requests amongst nodes.



Write requests are routed to the primary shard, then replicated
Read requests are routed to the primary or any replica

how many shards do i need?

- you can’t add more shards later without re-indexing
- but shards aren’t free – you can just make 1,000 of them and stick them on one node at first.
- you want to overallocate, but not too much
- consider scaling out in phases, so you have time to re-index before you hit the next phase

really? that’s kind of hand-wavy.

- the “right” number of shards depends on your data and your application. there’s no secret formula.
- start with a single server using the same hardware you use in production, with one shard and no replication.
- fill it with real documents and hit it with real queries.
- push it until it breaks – now you know the capacity of a single shard.

remember replica shards can be added

- read-heavy applications can add more replica shards without re-indexing.
- note this only helps if you put the new replicas on extra hardware!

adding an index

creating a new index

```
PUT /new_index
{
        “settings”: {
                “number_of_shards”:     10,
                “number_of_replicas”:    1
        }
}
```

You can use *index templates* to automatically apply mappings, analyzers, aliases, etc.

multiple indices as a scaling strategy

- make a new index to hold new data
- search both indices
- use *index aliases* to make this easy to do

multiple indices as a scaling strategy

- with time-based data, you can have one index per time frame
- common strategy for log data where you usually just want current data, but don’t want to delete old data either
- again you can use index aliases, ie “logs_current”, “last_3_months”, to point to specific indices as they rotate

alias rotation example

```
POST /_aliases
{
        “actions”: [
                { “add”:          { “alias”: “logs_current”, “index”: “logs_2017_06” }},
                { “remove”:       { “alias”: “logs_current”, “index”: “logs_2017_05” }},
                { “add”:          { “alias”: “logs_last_3_months”, “index”: “logs_2017_06” }},
                { “remove”:       { “alias”: “logs_last_3_months”, “index”: “logs_2017_03” }}
        ]
}
```

optionally….

```
DELETE /logs_2017_03
```

choosing your hardware

RAM is likely your bottleneck

64GB per machine is the sweet spot (32GB to elasticsearch, 32GB to the OS / disk cache for lucene)

under 8GB not recommended

other hardware considerations

- fast disks are better – SSD’s if possible (with deadline or noop i/o scheduler)
- user RAID0 – your cluster is already redundant
- cpu not that important
- need a fast network
- don’t use NAS
- use medium to large configurations; too big is bad, and too many small boxes is bad too.

heap sizing

your heap size is wrong

the default heap size is only 1GB!

half or less of your physical memory should be allocated to elasticsearch

- the other half can be used by lucene for caching
- if you're not aggregating on analyzed string fields, consider using less than half for elasticsearch
- smaller heaps result in faster garbage collection and more memory for caching

```
export ES_HEAP_SIZE=10g
```

or

```
ES_JAVA_OPTS="-Xms10g –Xmx10g" ./bin/elasticsearch
```

don't cross 32GB! pointers blow up then.

monitoring with x-pack

what is x-pack?

- an elastic stack extension
- security, monitoring, alerting, reporting, graph, and machine learning
- formerly shield / watcher / marvel
- only parts can be had for free – requires a paid license or trial otherwise

let’s install x-pack and mess around with it.

```
cd /usr/share/elasticsearch
sudo bin/elasticsearch-plugin install x-pack

sudo vi /etc/elasticsearch/elasticsearch.yml
(Add xpack.security.enabled:false)

sudo /bin/systemctl stop elasticsearch.service

sudo /bin/systemctl start elasticsearch.service
cd /usr/share/kibana/
sudo -u kibana bin/kibana-plugin install x-pack
sudo /bin/systemctl stop kibana.service

sudo /bin/systemctl start kibana.service
```

failover in action

in this activity, we’ll…

- Set up a second elasticsearch node on our virtual machine
- Observe how elasticsearch automatically expands across this new node
- Stop our original node, and observe everything move to the new one
- Restart our original node, and observe everything going back to normal… automatically!

using snapshots

snapshots let you back up your indices

store backups to NAS, Amazon S3, HDFS, Azure

smart enough to only store changes since last snapshot

create a repository

add it into elasticsearch.yml:

```
path.repo: ["/home/<user>/backups"]
```

```
PUT _snapshot/backup-repo
{
 "type": "fs",
 "settings": {
   "location": "/home/<user>/backups/backup-repo"
 }
}
```

using snapshots

snapshot all open indices:

```
PUT _snapshot/backup-repo/snapshot-1
```

get information about a snapshot:

```
GET _snapshot/backup-repo/snapshot-1
```

monitor snapshot progress:

```
GET _snapshot/backup-repo/snapshot-1/_status
```

restore a snapshot of all indices:

```
POST /_all/_close
POST _snapshot/backup-repo/snapshot-1/_restore
```

rolling restarts

restarting your cluster

sometimes you have to… OS updates, elasticsearch version updates, etc.

to make this go quickly and smoothly, you want to disable index reallocation while doing this.

rolling restart procedure

1. stop indexing new data if possible
2. disable shard allocation
3. shut down one node
4. perform your maintenance on it and restart, confirm it joins the cluster.
5. re-enable shard allocation
6. wait for the cluster to return to green status
7. repeat steps 2-6 for all other nodes
8. resume indexing new data

cheat sheet

```
PUT _cluster/settings
{
 "transient": {
   "cluster.routing.allocation.enable": "none"
 }
}
```

Disable shard allocation

```
sudo /bin/systemctl stop elasticsearch.service
```

Stop elasticsearch safely

```
PUT _cluster/settings
{
 "transient": {
   "cluster.routing.allocation.enable": "all"
 }
}
```

Enable shard allocation

let’s practice

amazon
elasticsearch
service

let’s walk through setting this up

amazon es lets you quickly rent and configure an elasticsearch cluster

this costs real money! Just watch if that bothers you

the main thing that’s different with amazon es is security

amazon es +logstash

let’s do something a little more complicated

- set up secure access to your cluster from kibana and from logstash
- need to create a IAM user and its credentials
- simultaneously allow access to the IP you’re connecting to kibana from and this user
- configure logstash with that user’s credentials for secure communication to the ES cluster

our access policy

substitute your own aws account ID, IAM user, cluster name, and IP address

```
{
 "Version": "2012-10-17",
 "Statement": [
   {
     "Effect": "Allow",
     "Principal": {
       "AWS": [
         "arn:aws:iam::159XXXXXXX66:user/estest",
         "arn:aws:iam:: 159XXXXXXX66:user/estest :root"
       ]
     },
     "Action": "es:*",
     "Resource": "arn:aws:es:us-east-1: 159XXXXXXX66:user/estest :domain/frank-test/*"
   },
   {
     "Effect": "Allow",
     "Principal": {
       "AWS": "*"
     },
     "Action": [
       "es:ESHttpGet",
       "es:ESHttpPut",
       "es:ESHttpPost",
       "es:ESHttpHead"
     ],
     "Resource": "arn:aws:es:us-east-1: 159XXXXXXX66:user/estest :domain/frank-test/*",
     "Condition": {
       "IpAddress": {
         "aws:SourceIp": [
           "192.168.1.1",
           "127.0.0.1",
           "68.204.31.192"
         ]
       }
     }
   }
 ]
}
```

our logstash configuration

Substitute your own log path, elasticsearch endpoint, region, and credentials

```
input {
	file {
		path => "/home/fkane/access_log-2"
	}
}

output {
	amazon_es {
		hosts => ["search-test-logstash-tdjkXXXXXXdtp3o3hcy.us-east-1.es.amazonaws.com"]
		region => "us-east-1"
		aws_access_key_id => 'AKIXXXXXXK7XYQQ'
		aws_secret_access_key => '7rvZyxmUudcXXXXXXXXXgTunpuSyw2HGuF'
		index => "production-logs-%{+YYYY.MM.dd}"
	}
```

elastic cloud

what is elastic cloud?

elastic's hosted solution
built on top of aws
includes x-pack (unlike amazon es)
simpler setup ui
x-pack security simplifies things
this costs extra!

let’s set up a
trial cluster.

wrapping up

you made it!

you learned a lot:

- installing elasticsearch
- mapping and indexing data
- searching data
- importing data
- aggregating data
- using kibana
- using logstash, beats, and the elastic stack
- elasticsearch operations and deployment
- using hosted elasticsearch clusters

learning more

- https://www.elastic.co/learn
- elasticsearch: the definitive guide
- documentation
- live training and videos
- keep experimenting!

THANK YOU