ThoughtWorks®

Workshop

# ELASTICSEARCH

*Felipe Dornelas*

# AGENDA

- *Part 1*
  - Introduction
  - Document Store
  - Search Examples
  - Data Resiliency
  - Comparison with Solr
- *Part 2*
  - Search
  - Analytics

# AGENDA

- ***Part 3***
  - Inverted Index
  - Analyzers
  - Mapping
  - Proximity Matching
  - Fuzzy Matching
- ***Part 4***
  - Inside a Cluster
  - Data Modeling

→ github.com/felipead/elasticsearch-workshop

# PRE-REQUISITES

- ☐ Vagrant
- ☐ VirtualBox
- ☐ Git

# ENVIRONMENT SETUP

- ☐ git clone https://github.com/felipead/elasticsearch-workshop.git
- ☐ vagrant up
- ☐ vagrant ssh
- ☐ cd /vagrant

# VERIFY EVERYTHING IS WORKING

- ☐ `curl http://localhost:9200`

ThoughtWorks®

# PART 1

*Core concepts*

ThoughtWorks®

# 1-1 INTRODUCTION

*You know, for search*

# WHAT IS ELASTICSEARCH?

*A real-time distributed* ***search*** *and* ***analytics*** *engine*

# IT CAN BE USED FOR

- ☐ *Full-text* search
- ☐ *Structured* search
- ☐ Real-time *analytics*
- ☐ ...or any combination of the above

# FEATURES

- Distributed *document store*:
  - *RESTful API*
  - Automatic scale
  - Plug & Play ™

# FEATURES

- Handles the *human language*:
  - Score results by *relevance*
  - *Synonyms*
  - *Typos* and *misspellings*
  - Internationalization

# FEATURES

- Powerful *analytics*:
  - Comprehensive *aggregations*
  - Geolocations
  - Can be combined with *search*
  - *Real-time* (no batch-processing)

# FEATURES

- ☐ *Free* and *open source*
- ☐ Community support
- ☐ Backed by Elastic

# MOTIVATION

*Most databases are inept at extracting **knowledge** from your data*

# SQL DATABASES

*SQL = **Structured** Query Language*

# SQL DATABASES

- Can only filter by ***exact values***
- ***Unable*** to perform ***full-text*** search
- Queries can be ***complex*** and ***inefficient***
- Often requires ***big-batch*** processing

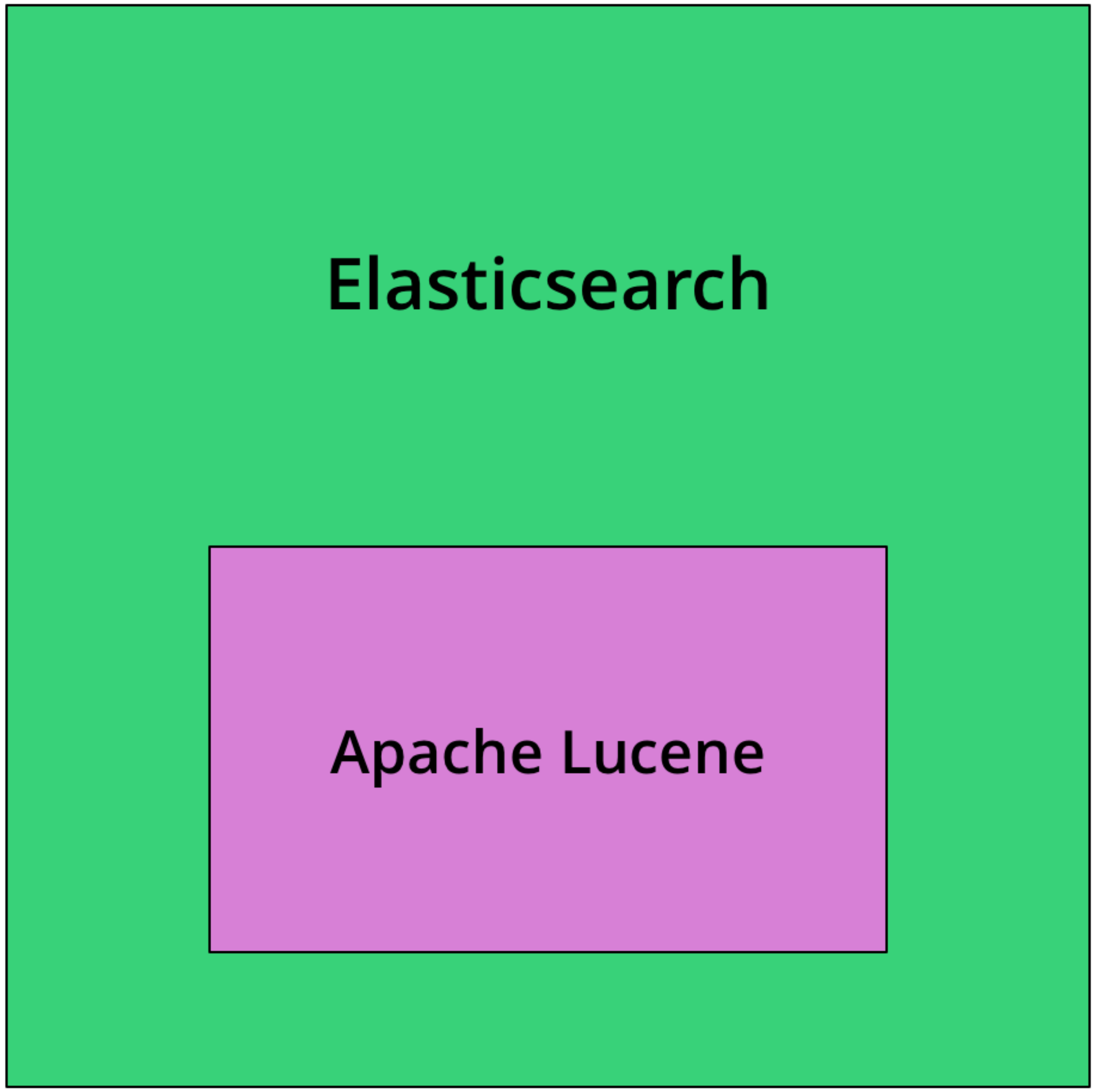
# APACHE LUCENE

- Arguably, the ***best*** search engine
- High performance
- Near real-time indexing
- Open source

# APACHE LUCENE

- But…
  - It’s just a *Java Library*
  - Hard to use

# ELASTICSEARCH




- ☐ Document Store
- ☐ Distributed
- ☐ Scalable
- ☐ Real Time
- ☐ Analytics
- ☐ RESTful API
- ☐ *Easy to Use*

# DOCUMENT ORIENTED

- *Documents* instead of rows / columns
- Every field is *indexed* and *searchable*
- Serialized to *JSON*
- *Schemaless*

# WHO USES

- GitHub
- Wikipedia
- Stack Overflow
- The Guardian

# TALKING TO ELASTICSEARCH

- *Java API*
  - Port 9300
  - Native transport protocol
  - Node client (*joins the cluster*)
  - Transport client (*doesn't join the cluster*)

# TALKING TO ELASTICSEARCH

- *RESTful API*
  - Port 9200
  - JSON over HTTP

# TALKING TO ELASTICSEARCH

*We will only cover the* ***RESTful API***

# USING CURL

```
curl -X <VERB> <URL> -d <BODY>
```

*or*

```
curl -X <VERB> <URL> -d @<FILE>
```

# THE EMPTY QUERY

```
curl -X GET
-d @part-1/empty-query.json
localhost:9200/_count?pretty
```

# REQUEST

```
{
  "query": {
    "match_all": {}
  }
}
```

# RESPONSE

```
{
  "count": 0,
  "_shards": {
    "total": 0,
    "successful": 0,
    "failed": 0
  }
}
```

ThoughtWorks®

# 1-2 DOCUMENT STORE

# THE PROBLEM WITH RELATIONAL DATABASES

- Stores data in ***columns*** and ***rows***
- Equivalent of using a ***spreadsheet***
- *Inflexible* storage medium
- Not suitable for ***rich objects***

# DOCUMENTS

```
{
  "name": "John Smith",
  "age": 42,
  "confirmed": true,
  "join_date": "2015-06-01",
  "home": {"lat": 51.5, "lon": 0.1},
  "accounts": [
    {"type": "facebook", "id": "johnsmith"},
    {"type": "twitter", "id": "johnsmith"}
  ]
}
```

# DOCUMENT METADATA

- *Index* - Where the document lives
- *Type* - Class of object that the document represents
- *Id* - Unique identifier for the document

# DOCUMENT METADATA

| | | | | |
|---|---|---|---|---|
| *Relational DB* | *Databases* | *Tables* | *Rows* | *Columns* |
| *Elasticsearch* | *Indices* | *Types* | *Documents* | *Fields* |

# RESTFUL API

```
[VERB] /{index}/{type}/{id}?pretty

GET | POST | PUT | DELETE | HEAD
```

# RESTFUL API

- *JSON*-only
- Adding ***pretty*** to the query-string parameters pretty-prints the response

# INDEXING A DOCUMENT WITH YOUR OWN ID

```
PUT /{index}/{type}/{id}
```

# INDEXING A DOCUMENT WITH YOUR OWN ID

```
curl -X PUT
-d @part-1/first-blog-post.json
localhost:9200/blog/post/123?pretty
```

# REQUEST

```
{
  "title": "My first blog post",
  "text": "Just trying this out...",
  "date": "2014-01-01"
}
```

# RESPONSE

```
{
  "_index" : "blog",
  "_type" : "post",
  "_id" : "123",
  "_version" : 1,
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 0
  },
  "created" : true
}
```

# INDEXING A DOCUMENT WITH AUTOGENERATED ID

```
POST /{index}/{type}
```

** Autogenerated IDs are **Base64**-encoded **UUIDs***

# INDEXING A DOCUMENT WITH AUTOGENERATED ID

```
curl -X POST
-d @part-1/second-blog-post.json
localhost:9200/blog/post?pretty
```

# REQUEST

```
{
   "title": "Second blog post",
   "text": "Still trying this out...",
   "date": "2014-01-01"
}
```

# RESPONSE

```
{
  "_index" : "blog",
  "_type" : "post",
  "_id" : "AVFWIbMf7YZ6Se7RwMws",
  "_version" : 1,
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 0
  },
  "created" : true
}
```

# RETRIEVING A DOCUMENT WITH METADATA

```
GET /{index}/{type}/{id}
```

# RETRIEVING A DOCUMENT WITH METADATA

```
curl -X GET
localhost:9200/blog/post/123?pretty
```

# RESPONSE


```
{
  "_index" : "blog",
  "_type" : "post",
  "_id" : "123",
  "_version" : 1,
  "found" : true,
  "_source": {
    "title": "My first blog entry",
    "text": "Just trying this out...",
    "date": "2014-01-01"
  }
}
```

# RETRIEVING A DOCUMENT WITHOUT METADATA

```
GET /{index}/{type}/{id}/_source
```

# RETRIEVING A DOCUMENT WITHOUT METADATA

```
curl -X GET
localhost:9200/blog/post/123/
_source?pretty
```

# RESPONSE


```
{
  "title": "My first blog entry",
  "text": "Just trying this out...",
  "date": "2014-01-01"
}
```

# RETRIEVING PART OF A DOCUMENT

```
GET /{index}/{type}/{id}
?_source={fields}
```

# RETRIEVING PART OF A DOCUMENT

```
curl -X GET
'localhost:9200/blog/post/123?
_source=title,date&pretty'
```

# RESPONSE


```
{
  "_index" : "blog",
  "_type" : "post",
  "_id" : "123",
  "_version" : 1,
  "found" : true,
  "_source": {
    "title": "My first blog entry",
    "date": "2014-01-01"
  }
}
```

# CHECKING WHETHER A DOCUMENT EXISTS

```
HEAD /{index}/{type}/{id}
```

# CHECKING WHETHER A DOCUMENT EXISTS

```
curl -i –X HEAD
localhost:9200/blog/post/123
```

# RESPONSE

```
HTTP/1.1 200 OK
Content-Length: 0
```

# CHECKING WHETHER A DOCUMENT EXISTS

```
curl -i –X HEAD
localhost:9200/blog/post/666
```

# RESPONSE

```
HTTP/1.1 404 Not Found
Content-Length: 0
```

# UPDATING A WHOLE DOCUMENT

```
PUT /{index}/{type}/{id}
```

# UPDATING A WHOLE DOCUMENT

```
curl -X PUT
-d @part-1/updated-blog-post.json
localhost:9200/blog/post/123?pretty
```

# REQUEST

```
{
   "title": "My first blog post",
   "text": "I am starting to get the hang of this...",
   "date": "2014-01-02"
}
```

# RESPONSE


```
{
  "_index" : "blog",
  "_type" : "post",
  "_id" : "123",
  "_version" : 2,
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 0
  },
  "created" : false
}
```

# DELETING A DOCUMENT

```
DELETE /{index}/{type}/{id}
```

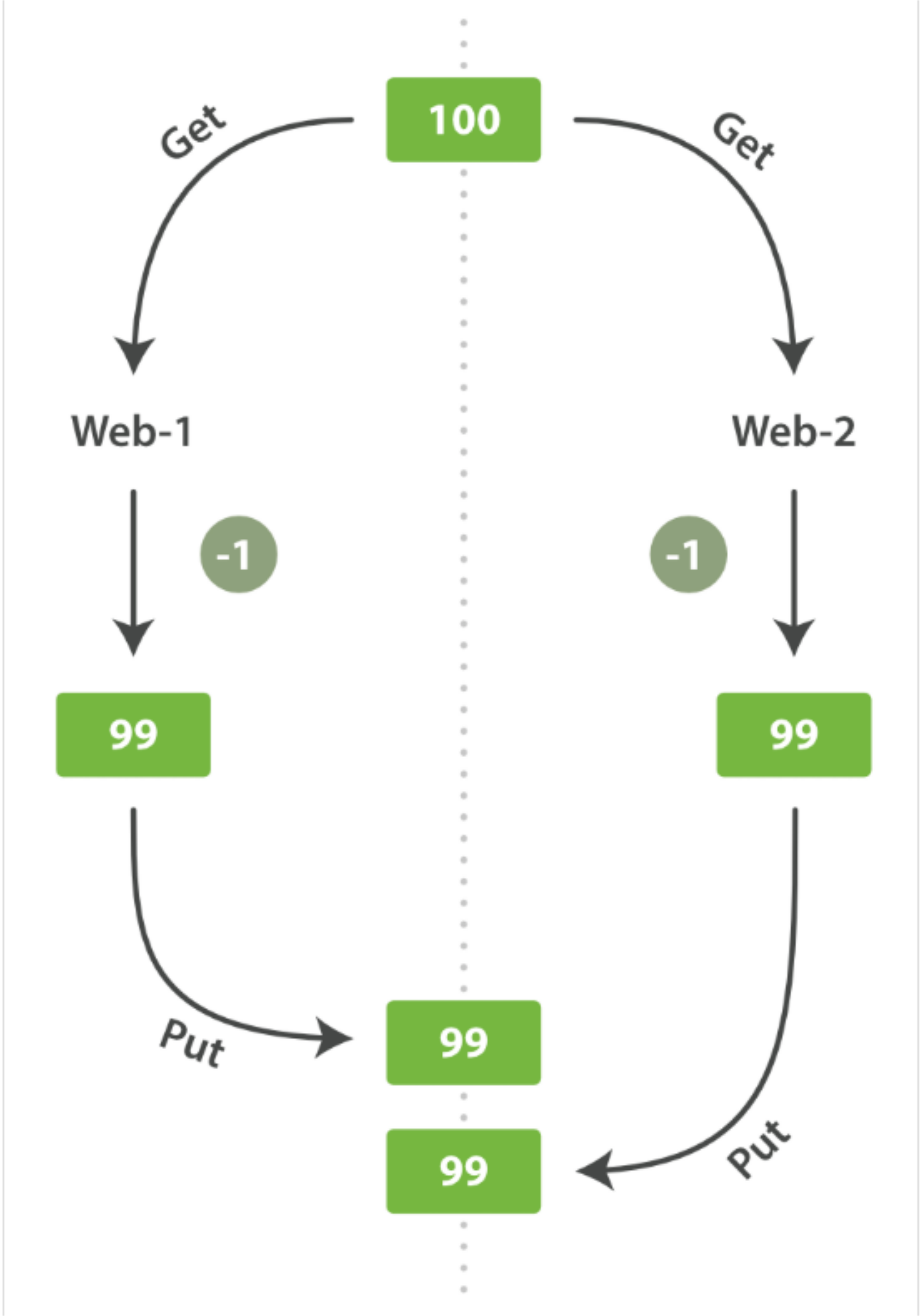
# DELETING A DOCUMENT

```
curl -X DELETE
localhost:9200/blog/post/123?pretty
```

# RESPONSE


```
{
  "found" : true,
  "_index" : "blog",
  "_type" : "post",
  "_id" : "123",
  "_version" : 3,
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 0
  }
}
```

# DEALING WITH CONFLICTS



# PESSIMISTIC CONCURRENCY CONTROL

- Used by relational databases
- Assumes conflicts are likely to happen (*pessimist*)
- *Blocks* access to resources

# OPTIMISTIC CONCURRENCY CONTROL

- Assumes conflicts are unlikely to happen (*optimist*)
- Does *not* block operations
- If conflict happens, update *fails*

# HOW ELASTICSEARCH DEALS WITH CONFLICTS

- Locking distributed resources would be very *inefficient*
- Uses *Optimistic Concurrency Control*
- Auto-increments `_version` number

# HOW ELASTICSEARCH DEALS WITH CONFLICTS

- `PUT /blog/post/123?version=1`
- If version is outdated returns *409 Conflict*

ThoughtWorks®

# 1-3 SEARCH EXAMPLES

# EMPLOYEE DIRECTORY EXAMPLE

- Index: ***megacorp***
- Type: ***employee***
- Ex: John Smith, Jane Smith, Douglas Fir

# EMPLOYEE DIRECTORY EXAMPLE

```
curl -X PUT
-d @part-1/john-smith.json
localhost:9200/megacorp/employee/1
```

# REQUEST


```
{
  "first_name": "John",
  "last_name": "Smith",
  "age": 25,
  "about": "I love to go rock climbing",
  "interests": ["sports", "music"]
}
```

# EMPLOYEE DIRECTORY EXAMPLE

```
curl -X PUT
-d @part-1/jane-smith.json
localhost:9200/megacorp/employee/2
```

# REQUEST

```
{
  "first_name": "Jane",
  "last_name": "Smith",
  "age": 32,
  "about": "I like to collect rock albums",
  "interests": ["music"]
}
```

# EMPLOYEE DIRECTORY EXAMPLE

```
curl -X PUT
-d @part-1/douglas-fir.json
localhost:9200/megacorp/employee/3
```

# REQUEST


```
{
  "first_name": "Douglas",
  "last_name": "Fir",
  "age": 35,
  "about": "I like to build cabinets",
  "interests": ["forestry"]
}
```

# SEARCHES ALL EMPLOYEES

```
GET /megacorp/employee/_search
```

# SEARCHES ALL EMPLOYEES

```
curl -X GET
localhost:9200/megacorp/employee/
_search?pretty
```

# SEARCH WITH QUERY-STRING

```
GET /megacorp/employee/_search
?q=last_name:Smith
```

# SEARCH WITH QUERY-STRING

```
curl -X GET
'localhost:9200/megacorp/employee/
_search?q=last_name:Smith&pretty'
```

# RESPONSE

```
"hits" : {
  "total" : 2,
  "max_score" : 0.30685282,
  "hits" : [ {
    …
    "_score" : 0.30685282,
    "_source": {
      "first_name": "Jane",
      "last_name": "Smith", … }
  }, {
    …
    "_score" : 0.30685282,
    "_source": {
      "first_name": "John",
      "last_name": "Smith", … }
    } ]
}
```

# SEARCH WITH QUERY DSL

```
curl -X GET
-d @part-1/last-name-query.json
localhost:9200/megacorp/employee/
_search?pretty
```

# REQUEST

```
{
  "query": {
    "match": {
      "last_name": "Smith"
    }
  }
}
```

# RESPONSE

```
"hits" : {
  "total" : 2,
  "max_score" : 0.30685282,
  "hits" : [ {
    …
    "_score" : 0.30685282,
    "_source": {
      "first_name": "Jane",
      "last_name": "Smith", … }
  }, {
    …
    "_score" : 0.30685282,
    "_source": {
      "first_name": "John",
      "last_name": "Smith", … }
    } ]
}
```

# SEARCH WITH QUERY DSL AND FILTER

```
curl -X GET
-d @part-1/last-name-age-query.json
localhost:9200/megacorp/employee/
_search?pretty
```

# REQUEST

```
"query": {
    "filtered": {
      "filter": {
        "range": {
          "age": { "gt": 30 }
        }
      },
      "query": {
        "match": { "last_name": "Smith" }
      }
    }
  }
```

# RESPONSE

```
"hits" : {
  "total" : 1,
  "max_score" : 0.30685282,
  "hits" : [ {
    …
    "_score" : 0.30685282,
    "_source": {
      "first_name": "Jane",
      "last_name": "Smith",
      "age": 32, … }
  } ]
```

# FULL-TEXT SEARCH

```
curl -X GET
-d @part-1/full-text-search.json
localhost:9200/megacorp/employee/
_search?pretty
```

# REQUEST

```
{
  "query": {
    "match": {
      "about": "rock climbing"
    }
  }
}
```

# RESPONSE

```
"hits" : [{ …
    "_score" : 0.16273327,
    "_source": {
      "first_name": "John", "last_name": "Smith",
      "about": "I love to go rock climbing", … }
  }, { …
    "_score" : 0.016878016,
    "_source": {
      "first_name": "Jane", "last_name": "Smith",
      "about": "I like to collect rock albums", … }
}]
```

# RELEVANCE SCORES

- The `_score` field ranks searches results
- The *higher* the score, the ***better***

# PHRASE SEARCH

```
curl -X GET
-d @part-1/phrase-search.json
localhost:9200/megacorp/employee/
_search?pretty
```

# REQUEST

```
{
  "query": {
    "match_phrase": {
      "about": "rock climbing"
    }
  }
}
```

# RESPONSE

```
"hits" : {
  "total" : 1,
  "max_score" : 0.23013961,
  "hits" : [ {
    …
    "_score" : 0.23013961,
    "_source": {
      "first_name": "John",
      "last_name": "Smith",
      "about": "I love to go rock climbing"
      … }
  } ]
}
```

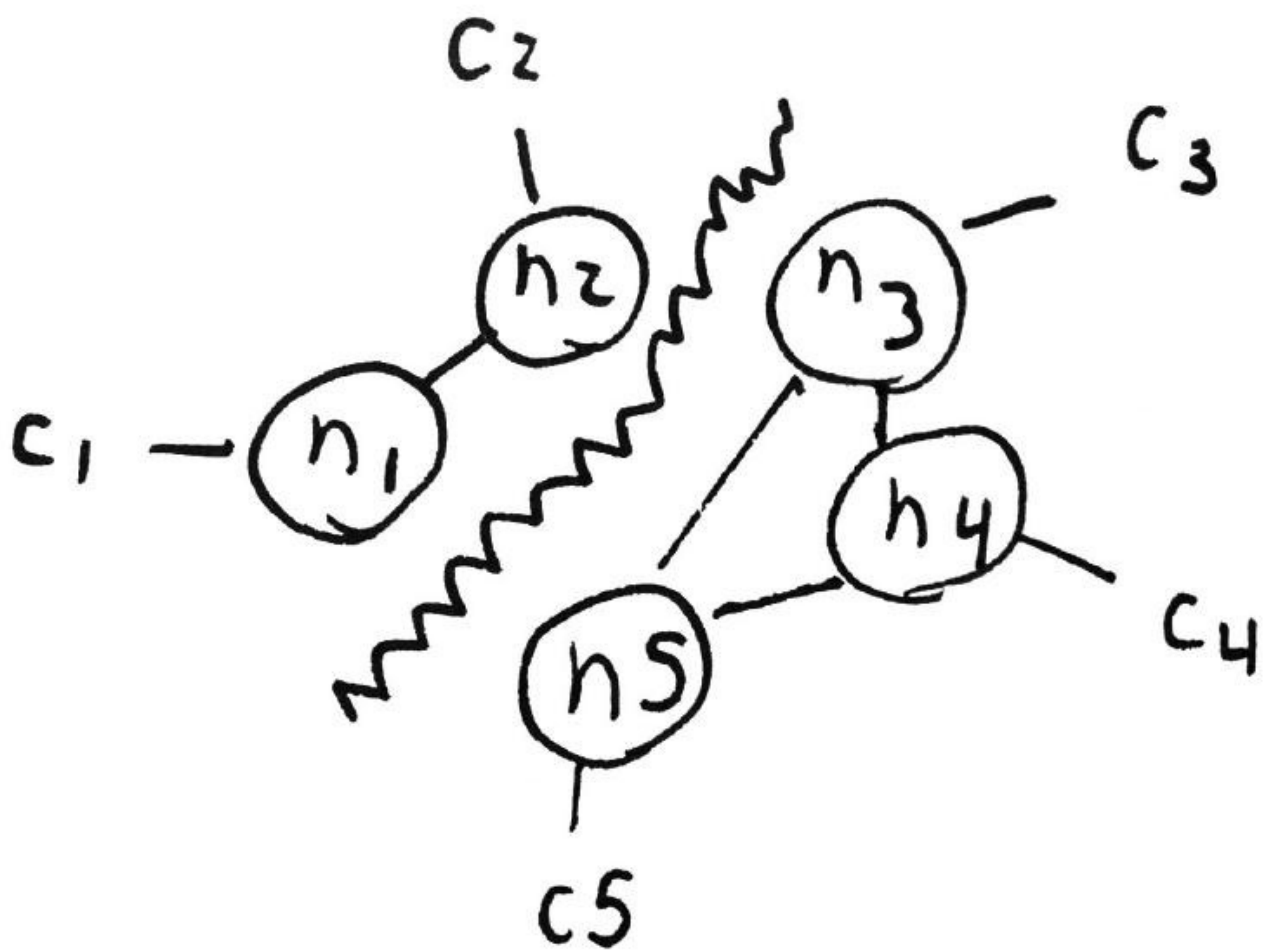
ThoughtWorks®

# 1-4 DATA RESILIENCY

# CALL ME MAYBE

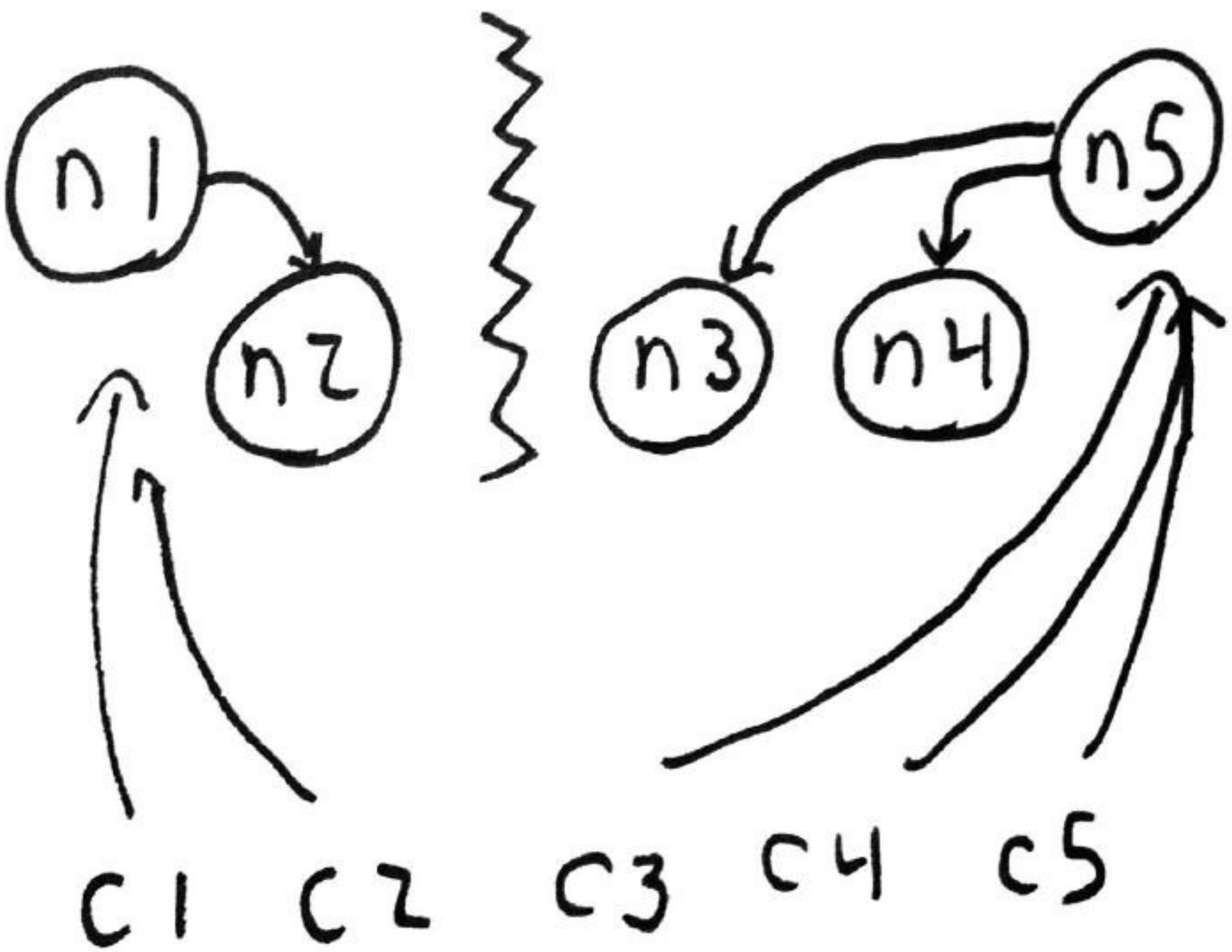
- *Jepsen Tests*
- Simulates ***network partition*** scenarios
- Run several operations against a distributed system
- Verify that the history of those operations makes sense

# NETWORK PARTITION



# ELASTICSEARCH STATUS

- Risk of data loss on network partition and *split-brain* scenarios



# IT IS NOT SO BAD...

- Still much more resilient than *MongoDB*
- Elastic is working hard to improve it
- *Two-phase commits* are planned

# IF YOU REALLY CARE ABOUT YOUR DATA

- Use a more reliable *primary* data store:
  - Cassandra
  - Postgres
- *Synchronize* it to Elasticsearch
- ...or set-up comprehensive *back-up*

*There's* ***no such thing*** *as a 100% reliable distributed system*

# 1-5 SOLR COMPARISON

# SOLR

- *SolrCloud*
- Both:
  - Are *open-source* and *mature*
  - Are based on *Apache Lucene*
  - Have more or less *similar features*

# SOLR API

- *HTTP GET*
- Query parameters passed in as URL parameters
- Is ***not*** RESTful
- Multiple formats (JSON, XML…)

# SOLR API

- Version 4.4 added *Schemaless* API
- Older versions require up-front *Schema*

# ELASTICSEARCH API

- *RESTful*
- *Schemaless*
- CRUD document operations
- Manage indices, read metrics, etc…

# ELASTICSEARCH API

- *Query DSL*
- Better ***readability***
- JSON-only

# SEARCH

- Both are very good with ***text search***
- Both based on ***Apache Lucene***

# EASYNESS OF USE

- Elasticsearch is ***simpler***:
  - Just a single process
  - Easier API
- SolrCloud requires *Apache ZooKeeper*

# SOLRCLOUD DATA RESILIENCY

- SolrCloud uses Apache ZooKeeper to discover nodes
- Better at preventing *split-brain* conditions
- *Jepsen Tests* pass

# ANALYTICS

- Elasticsearch is the choice for ***analytics:***
  - Comprehensive aggregations
  - Thousands of metrics
  - SolrCloud is not even close

ThoughtWorks®

# PART 2

*Search and Analytics*

ThoughtWorks®

# 2-1 SEARCH

*Finding the needle in the haystack*

# TWEETS EXAMPLE

- /<country_code>/user
- /<country_code>/tweet

# TWEETS EXAMPLE

```
/us/user/1

{
 "email": "john@smith.com",
 "name": "John Smith",
 "username": "@john"
}
```

# TWEETS EXAMPLE

```
/gb/user/2

{
 "email": "mary@jones.com",
 "name": "Mary Jones",
 "username": "@mary"
}
```

# TWEET EXAMPLE

/gb/tweet/3

```
{
  "date": "2014-09-13",
  "name": "Mary Jones",
  "tweet": "Elasticsearch means full text search has never been so easy",
  "user_id": 2
}
```

# TWEETS EXAMPLE

```
./part-2/load-tweet-data.sh
```

# THE EMPTY SEARCH

- Returns all documents on all indices

```
GET /_search
```

# THE EMPTY SEARCH

```
curl -X GET
localhost:9200/_search?pretty
```

# THE EMPTY SEARCH

```
"hits" : {
  "total" : 14,
  "hits" : [
    {
      "_index": "us",
      "_type": "tweet",
      "_id": "7",
      "_score": 1,
      "_source": {
        "date": "2014-09-17",
        "name": "John Smith",
        "tweet": "The Query DSL is really powerful and flexible",
        "user_id": 2
      }
    },
  ... 9 RESULTS REMOVED ...
  ]
}
```

# MULTI-INDEX, MULTITYPE SEARCH

- `/_search`
- `/gb/_search`
- `/gb,us/_search`
- `/gb/user/_search`
- `/_all/user,tweet/_search`

# PAGINATION

- Returns ***10 results*** per request (default)
- Control parameters:
  - *size*: number of results to return
  - *from*: number of results to skip

# PAGINATION

- GET /_search?size=5
- GET /_search?size=5&from=5
- GET /_search?size=5&from=10

# TYPES OF SEARCH

- *Structured query* on concrete fields (similar to SQL)
- *Full-text query* (sorts results by relevance)
- Combination of the two

# SEARCH BY EXACT VALUES

- Examples:
  - date
  - user ID
  - username
- “Does this document *match* the query?”

# SEARCH BY EXACT VALUES

- SQL queries:

```
SELECT * FROM user
WHERE name = "John Smith"
    AND user_id = 2
    AND date > "2014-09-15"
```

# FULL-TEXT SEARCH

- Examples:
  - the text of a tweet
  - body of an email
- "*How well* does this document match the query?"

# FULL-TEXT SEARCH

- *UK* should also match *United Kingdom*
- *jump* should also match *jumped*, *jumps*, *jumping* and *leap*

# FULL-TEXT SEARCH

- *fox news hunting* should return stories about *hunting on Fox News*
- *fox hunting news* should return news stories about *fox hunting*

# HOW ELASTICSEARCH PERFORMS TEXT SEARCH

- *Analyzes* the text
  - *Tokenizes* into terms
  - *Normalizes* the terms
- Builds an *inverted index*

# LIST OF INDEXED DOCUMENTS

| ID | Text |
|---|---|
| 1 | Baseball is played during summer months. |
| 2 | Summer is the time for picnics here. |
| 3 | Months later we found out why. |
| 4 | Why is summer so hot here. |

# INVERTED INDEX

| Term | Frequency | Document IDs |
|---|---|---|
| *baseball* | 1 | 1 |
| *during* | 1 | 1 |
| *found* | 1 | 3 |
| *here* | 2 | 2, 4 |
| *hot* | 1 | 4 |
| *is* | 3 | 1, 2, 4 |
| *months* | 2 | 1, 3 |
| *summer* | 3 | 1, 2, 4 |
| *the* | 1 | 2 |
| *why* | 2 | 3, 4 |

# QUERY DSL

```
GET /_search

{
   "query": YOUR_QUERY_HERE
}
```

# QUERY BY FIELD

```
{
  "match": {
    "tweet": "elasticsearch"
  }
}
```

# QUERY BY FIELD

```
curl -X GET -d
@part-2/elasticsearch-tweets-query.json
localhost:9200/_all/tweet/_search
```

# QUERY WITH MULTIPLE CLAUSES

```
{
  "bool":
    "must": {
      "match": { "tweet": "elasticsearch"}
    },
    "must_not": {
      "match": { "name": "mary" }
    },
    "should": {
      "match": { "tweet": "full text" }
    }
}
```

# QUERY WITH MULTIPLE CLAUSES

```
curl -X GET -d
@part-2/combining-tweet-queries.json
localhost:9200/_all/tweet/_search
```

# QUERY WITH MULTIPLE CLAUSES

```
"_score": 0.07082729, "_source": { …
 "name": "John Smith",
 "tweet": "The Elasticsearch API is really easy to use"
}, …
"_score": 0.049890988, "_source": { …
  "name": "John Smith",
  "tweet": "Elasticsearch surely is one of the hottest new NoSQL products"
}, …
"_score": 0.03991279, "_source": { …
  "name": "John Smith",
  "tweet": "Elasticsearch and I have left the honeymoon stage, and I still love her." }
```

# MOST IMPORTANT QUERIES

- *match*
- *match_all*
- *multi_match*
- *bool*

# QUERIES VS. FILTERS

- *Queries*:
  - full-text
  - "*how well* does the document match?"
- *Filters*:
  - exact values
  - *yes-no* questions

# QUERIES VS. FILTERS

- The goal of ***filters*** is to reduce the number of documents that have to be examined by a ***query***

# PERFORMANCE COMPARISON

- Filters are easy to *cache* and can be reused efficiently
- Queries are heavier and *non-cacheable*

# WHEN TO USE WHICH

- Use queries only for *full-text search*
- Use filters for anything else

# FILTER BY EXACT FIELD VALUES

```
"filtered": {
  "filter": {
    "term": {
      "user_id": 1
    }
  }
}
```

# FILTER BY EXACT FIELD VALUES

```
curl -X GET -d
@part-2/user-id-filter.json
localhost:9200/_search
```

# FILTER BY EXACT FIELD VALUES

```
"filtered": {
  "filter": {
    "range": {
      "date": {
        "gte": "2014-09-20"
      }
    }
  }
}
```

# FILTER BY EXACT FIELD VALUES

```
curl -X GET -d
@part-2/date-filter.json
localhost:9200/_search
```

# MOST IMPORTANT FILTERS

- ☐ *term*
- ☐ *terms*
- ☐ *range*
- ☐ *exists* and *missing*
- ☐ *bool*

# COMBINING QUERIES WITH FILTERS

```
"filtered": {
  "query": {
    "match": {
      "tweet": "elasticsearch"
    }
  },
  "filter": {
    "term": { "user_id": 1 }
  }
}
```

# COMBINING QUERIES WITH FILTERS

```
curl -X GET -d
@part-2/filtered–tweet-query.json
localhost:9200/_search
```

# SORTING

- *Relevance score*
- The ***higher*** the score, the ***better***
- By default, results are returned in ***descending*** order of relevance
- You can sort by ***any field***

# RELEVANCE SCORE

- *Similarity algorithm*
  - Term Frequency / Inverse Document Frequency (*TF/IDF*)

# RELEVANCE SCORE

- *Term frequency*
  - How often does the term appear in the field?
  - The more often, the ***more*** relevant

# RELEVANCE SCORE

- *Inverse document frequency*
  - How often does each term appear in the index?
  - The more often, the *less* relevant

# RELEVANCE SCORE

- *Field-length norm*
  - How long is the field?
  - The longer it is, the *less likely* it is that words in the field will be relevant

ThoughtWorks®

# 2-2 ANALYTICS

*How many needles are in the haystack?*

# SEARCH

- Just looks for the ***needle*** in the haystack

# BUSINESS QUESTIONS

- ☐ How ***many*** needles are in the haystack?
- ☐ What is the needle ***average length***?
- ☐ What is the ***median length*** of the needles, ***by manufacturer***?
- ☐ How many needles were added to the haystack ***each month***?

# BUSINESS QUESTIONS

- What are your ***most popular*** needle manufactures?
- Are there any ***anomalous*** clumps of needles?

# AGGREGATIONS

- Answer *Analytics* questions
- Can be combined with *Search*
- Near *real-time* in Elasticsearch
  - SQL queries can take *days*

*Buckets + Metrics*

# BUCKETS

- Collection of documents that meet a certain *criteria*
- Can be *nested* inside other buckets

# BUCKETS

- *Employee* ⇒ *male* or *female* bucket
- *San Francisco* ⇒ *California* bucket
- *2014-10-28* ⇒ *October* bucket

# METRICS

- *Calculations* on top of buckets
- Answer the questions
- Ex: ***min***, ***max***, ***mean***, ***sum***…

# EXAMPLE

- Partition by ***country*** (*bucket*)
- …then partition by ***gender*** (*bucket*)
- …then partition by ***age ranges*** (*bucket*)
- …calculate the ***average salary*** for each age range (*metric*)

# CAR TRANSACTIONS EXAMPLE

- `/cars/transactions`

# CAR TRANSACTIONS EXAMPLE

```
/cars/transactions/
AVFr1xbVmdUYWpF46Ps4

{
   "price" : 10000,
   "color" : "red",
   "make" : "honda",
   "sold" : "2014-10-28"
}
```

# CAR TRANSACTIONS EXAMPLE

```
./part-2/load-car-data.sh
```

# BEST SELLING CAR COLOR

```
{
  "aggs": {
    "colors": {
      "terms": {
        "fields": "color"
      }
    }
  }
}
```

# BEST SELLING CAR COLOR

```
curl -X GET -d
@part-2/best-selling-car-color.json
'localhost:9200/cars/transactions/
_search?search_type=count&pretty'
```

# BEST SELLING CAR COLOR

```
"colors" : {
  "buckets" : [{
    "key" : "red",
    "doc_count" : 16
  }, {
    "key" : "blue",
    "doc_count" : 8
  }, {
    "key" : "green",
    "doc_count" : 8
  }]
}
```

# AVERAGE CAR COLOR PRICE

```
{
  "aggs": {
    "colors": {
      "terms": { "field": "color" },
      "aggs": {
        "avg_price": {
          "avg": { "field": "price" }
        }
      }
    }
  }
}
```

# AVERAGE CAR COLOR PRICE

```
curl -X GET -d
@part-2/average-car-color-price.json
'localhost:9200/cars/transactions/
_search?search_type=count&pretty'
```

# AVERAGE CAR COLOR PRICE

```
"colors" : {
  "buckets": [{
    "key": "red", "doc_count": 16,
    "avg_price": { "value": 32500.0 }
  }, {
    "key": "blue", "doc_count": 8,
    "avg_price": { "value": 20000.0 }
  }, {
    "key": "green", "doc_count": 8,
    "avg_price": { "value": 21000.0 }
  }]
}
```

# BUILDING BAR CHARTS

- Very easy to convert aggregations to *charts* and *graphs*
- Ex: *histograms* and *time-series*

# CAR SALES REVENUE HISTOGRAM

```
{
  "aggs": {
    "price": {
      "histogram": {
        "field": "price",
        "interval": 20000
      },
      "aggs": {
        "revenue": {"sum": {"field" : "price"}}
      }
    }
  }
}
```

# CAR SALES REVENUE HISTOGRAM

```
curl -X GET -d
@part-2/car-revenue-histogram.json
'localhost:9200/cars/transactions/
_search?search_type=count&pretty'
```

# CAR SALES REVENUE HISTOGRAM


```
"price" : {
  "buckets": [
  { "key": 0, "doc_count": 12,
    "revenue": {"value": 148000.0} },
  { "key": 20000, "doc_count": 16,
    "revenue": {"value": 380000.0} },
  { "key": 40000, "doc_count": 0,
    "revenue": {"value": 0.0} },
  { "key": 60000, "doc_count": 0,
    "revenue": {"value": 0.0} },
  { "key": 80000, "doc_count": 4,
    "revenue": {"value" : 320000.0} }
]}
```

# CAR SALES REVENUE HISTOGRAM

# TIME-SERIES DATA

- Data with a ***timestamp***:
  - How many cars sold ***each month*** this year?
  - What was the price of this stock for the ***last 12 hours***?
  - What was the average latency of our website ***every hour*** in the last week?

# HOW MANY CARS SOLD PER MONTH?

```
{
  "aggs": {
    "sales": {
      "date_histogram": {
        "field": "sold",
        "interval": "month",
        "format": "yyyy-MM-dd"
      }
    }
  }
}
```

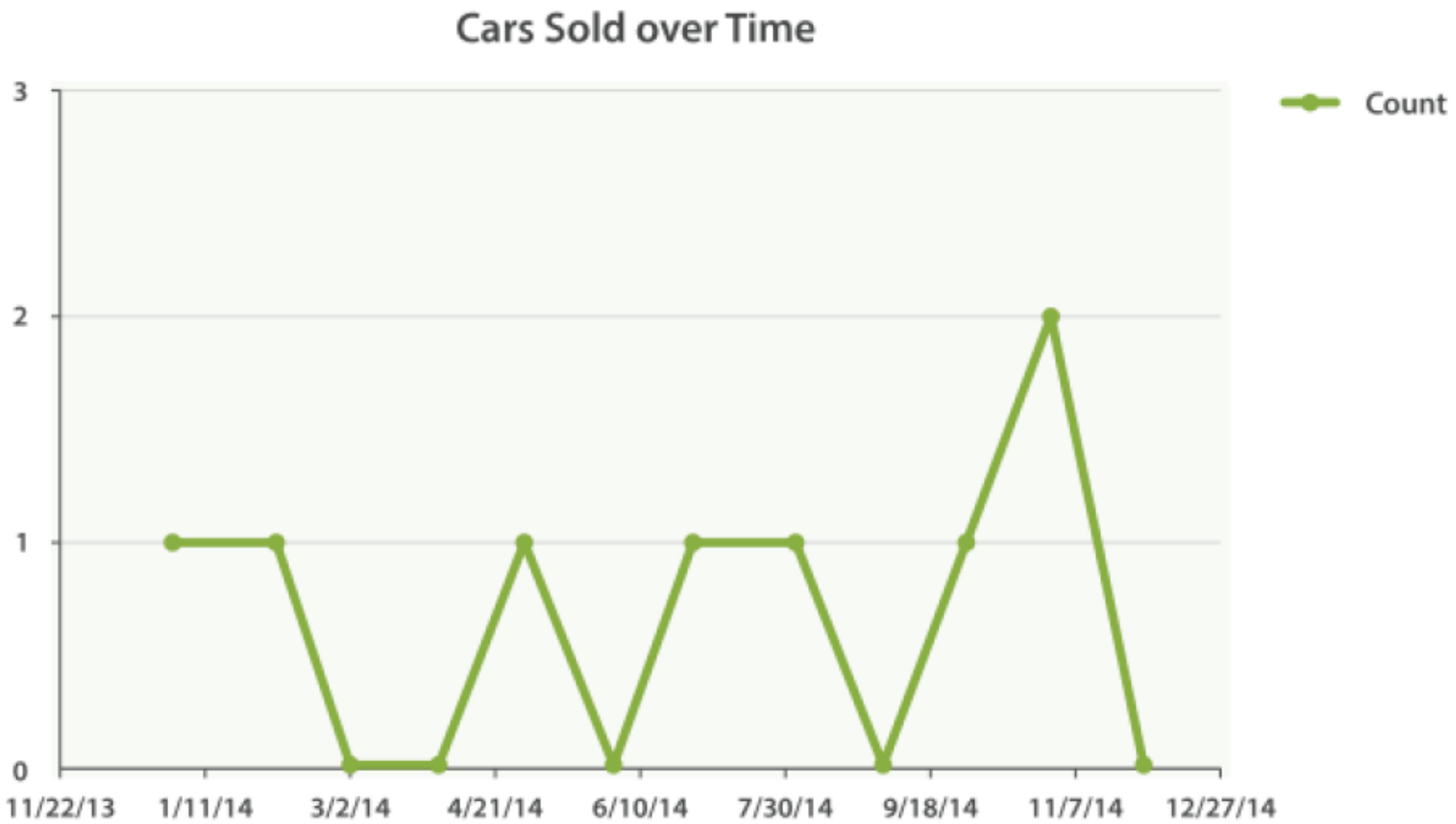
# HOW MANY CARS SOLD PER MONTH?

```
curl -X GET -d
@part-2/car-sales-per-month.json
'localhost:9200/cars/transactions/
_search?search_type=count&pretty'
```

# HOW MANY CARS SOLD PER MONTH?

```
"sales" : {
  "buckets" : [
    {"key_as_string": "2014-01-01", "doc_count": 4},
    {"key_as_string": "2014-02-01", "doc_count": 4},
    {"key_as_string": "2014-03-01", "doc_count": 0},
    {"key_as_string": "2014-04-01", "doc_count": 0},
    {"key_as_string": "2014-05-01", "doc_count": 4},
    {"key_as_string": "2014-06-01", "doc_count": 0},
    {"key_as_string": "2014-07-01", "doc_count": 4},
    {"key_as_string": "2014-08-01", "doc_count": 4},
    {"key_as_string": "2014-09-01", "doc_count": 0},
    {"key_as_string": "2014-10-01", "doc_count": 4},
    {"key_as_string": "2014-11-01", "doc_count": 8}
  ]
}
```

# HOW MANY CARS SOLD PER MONTH?



# PART 3

*Dealing with human language*

# 3-1 INVERTED INDEX

# INVERTED INDEX

- Data structure
- Efficient *full-text* search

# EXAMPLE

*Document 1*

```
The quick brown fox jumped
over the lazy dog
```

*Document 2*

```
Quick brown foxes leap over
lazy dogs in summer
```

# TOKENIZATION

***Document 1***

```
["The", "quick", "brown",
 "fox", "jumped", "over",
 "the", "lazy", "dog"]
```

***Document 2***

```
["Quick", "brown", "foxes",
 "leap", "over", "lazy",
 "dogs", "in", "summer"]
```

| Term | Document 1 | Document 2 |
| --- | --- | --- |
| *Quick* | | × |
| *The* | × | |
| *brown* | × | × |
| *dog* | × | |
| *dogs* | | × |
| *fox* | × | |
| *foxes* | | × |
| *in* | | × |
| *jumped* | × | |
| *lazy* | × | × |
| *leap* | | × |
| *over* | × | × |
| *quick* | × | |
| *summer* | | × |
| *the* | × | |

# EXAMPLE

- Searching for *"quick brown"*

| Term | Document 1 | Document 2 |
|---|---|---|
| *brown* | × | × |
| *quick* | × | |
| **Total** | 2 | 1 |

- *Naive similarity* algorithm:
  - Document 1 is a *better match*

# A FEW PROBLEMS

- *Quick* and *quick* are the same word
- *fox* and *foxes* are pretty similar
- *jumped* and *leap* are synonyms

# NORMALIZATION

- *Quick* ***lowercased*** to *quick*
- *foxes* ***stemmed*** to *fox*
- *jumped* and *leap* ***replaced*** by *jump*

# BETTER INVERTED INDEX

| Term | Document 1 | Document 2 |
|---|---|---|
| *brown* | × | × |
| *dog* | × | × |
| *fox* | × | × |
| *in* | | × |
| *jump* | × | × |
| *lazy* | × | × |
| *over* | × | × |
| *quick* | × | × |
| *summer* | | × |
| *the* | × | × |

# SEARCH INPUT

- You can only find terms that exist in the *inverted index*
- The *query string* is also *normalized*

ThoughtWorks®

# 3-2 ANALYZERS

# ANALYSIS

- *Tokenizes* a block of text into ***terms***
- *Normalizes* terms to standard form
- Improves searchability

# ANALYZERS

- Pipeline:
  - Character filters
  - Tokenizer
  - Token filters

# BUILT-IN ANALYZERS

- *Standard* analyzer
- *Language-specific* analyzers
  - 30+ languages supported

# TESTING THE STANDARD ANALYZER

```
GET /_analyze?
analyzer=standard
```

```
The quick brown fox jumped
over the lazy dog.
```

# TESTING THE STANDARD ANALYZER

```
curl -X GET -d
@part-3/quick-brown-fox.txt
'localhost:9200/_analyze?
analyzer=standard&pretty'
```

# TESTING THE STANDARD ANALYZER

```
"tokens" : [
   {"token": "the", …},
   {"token": "quick", …},
   {"token": "brown", …},
   {"token": "fox", …},
   {"token": "jumps", …},
   {"token": "over", …},
   {"token": "the", …},
   {"token": "lazy", …},
   {"token": "dog", …}
]
```

# TESTING THE ENGLISH ANALYZER

```
GET /_analyze?analyzer=english

The quick brown fox jumped
over the lazy dog.
```

# TESTING THE ENGLISH ANALYZER

```
curl -X GET -d
@part-3/quick-brown-fox.txt
'localhost:9200/_analyze?
analyzer=english&pretty'
```

# TESTING THE ENGLISH ANALYZER

```
"tokens" : [
   {"token": "quick", …},
   {"token": "brown", …},
   {"token": "fox", …},
   {"token": "jump", …},
   {"token": "over", …},
   {"token": "lazi", …},
   {"token": "dog", …}
]
```

# TESTING THE BRAZILIAN ANALYZER

```
GET /_analyze?
analyzer=brazilian
```

```
A rápida raposa marrom pulou
sobre o cachorro preguiçoso.
```

# TESTING THE BRAZILIAN ANALYZER

```
curl -X GET -d
@part-3/raposa-rapida.txt
'localhost:9200/_analyze?
analyzer=brazilian&pretty'
```

# TESTING THE BRAZILIAN ANALYZER

```
"tokens" : [
   {"token": "rap", …},
   {"token": "rapos", …},
   {"token": "marrom", …},
   {"token": "pul", …},
   {"token": "cachorr", …},
   {"token": "preguic", …}
]
```

# STEMMERS

- *Algorithmic* stemmers:
  - Faster
  - Less precise
- *Dictionary* stemmers:
  - Slower
  - More precise

ThoughtWorks®

# 3-3 MAPPING

# MAPPING

- Every document has a ***type***
- Every type has its own ***mapping***
- A mapping defines:
  - The ***fields***
  - The ***datatype*** for each field

# MAPPING

- ☐ Elasticsearch ***guesses*** the mapping when a new field is added
- ☐ Should customize the mapping for improved ***search*** and ***performance***
- ☐ Must customize the mapping when ***type is created***

# MAPPING

- A field's mapping *cannot be changed*
- You can still add ***new fields***
- Only option is to ***reindex*** all documents
- Reindexing with zero-downtime:
  - *index aliases*

# CORE FIELD TYPES

- String
- Integer
- Floating-point
- Boolean
- Date
- *Inner Objects*

# VIEWING THE MAPPING

```
GET /{index}/_mapping/{type}
```

# VIEWING THE MAPPING

```
curl -X GET
'localhost:9200/gb/_mapping/
tweet?pretty'
```

# VIEWING THE MAPPING

```
"date": {
  "type": "date",
  "format":
    "strict_date_optional_time…"
},
"name": {
  "type": "string"
},
"tweet": {
  "type": "string"
},
"user_id": {
  "type": "long"
}
```

# CUSTOMIZING FIELD MAPPINGS

- Distinguish between:
  - *Full-text* string fields
  - *Exact value* string fields
- Use *language-specific* analyzers

# STRING MAPPING ATTRIBUTES

- `index`:
  - `analyzed` (***full-text search***, *default*)
  - `not_analyzed` (***exact value***)
- `analyzer`:
  - `standard` (*default*)
  - `english`
  - ...

# ADDING NEW SEARCHABLE FIELD

```
PUT /gb,us/_mapping/tweet

{
  "properties": {
    "description": {
      "type": "string",
      "index": "analyzed",
      "analyzer": "english"
    }
  }
}
```

# ADDING NEW SEARCHABLE FIELD

```
curl -X PUT -d
@part-3/add-new-mapping.json
'localhost:9200/gb,us/
_mapping/tweet?pretty'
```

# ADDING NEW SEARCHABLE FIELD

```
curl -X GET
'localhost:9200/us,gb/
_mapping/tweet?pretty'
```

# ADDING NEW SEARCHABLE FIELD

```
...
"description": {
  "type": "string",
  "analyzer": "english"
}...
```

ThoughtWorks®

# 3-4 PROXIMITY MATCHING

# THE PROBLEM

- ***Sue*** *ate the* ***alligator***
- *The* ***alligator*** *ate* ***Sue***
- ***Sue*** *never goes anywhere without her* ***alligator****-skin purse*

# THE PROBLEM

- Search for ***“sue alligator”*** would match all three
- *Sue* and *alligator* may be separated by ***paragraphs*** of other text

# HEURISTIC

- Words that appear ***near each other*** are probably related
- Give documents in which the words are close together a ***higher relevance score***

# TERM POSITIONS

```
GET /_analyze?
analyzer=standard
```

```
Quick brown fox.
```

# TERM POSITIONS

```
"tokens": [
  { "token": "quick", …
    "position": 1 },
  { "token": "brown", …
    "position": 2 },
  { "token": "fox", …
    "position": 3 }
]
```

# EXACT PHRASE MATCHING

```
GET /{index}/{type}/_search

{
  "query": {
    "match_phrase": {
      "title": "quick brown fox"
    }
  }
}
```

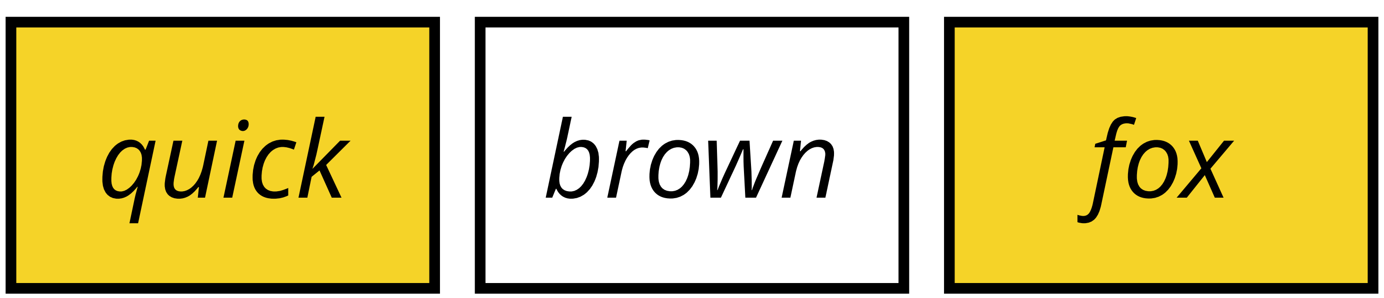
# EXACT PHRASE MATCHING

- *quick*, *brown* and *fox* must all appear
- The position of ***brown*** must be 1 greater than the position of ***quick***
- The position of ***fox*** must be 2 greater than the position of ***quick***

| *quick* | *brown* | *fox* |
|---|---|---|

# FLEXIBLE PHRASE MATCHING

- Exact phrase matching is ***too strict***
- *"quick fox"* should also match
- *Slop* matching



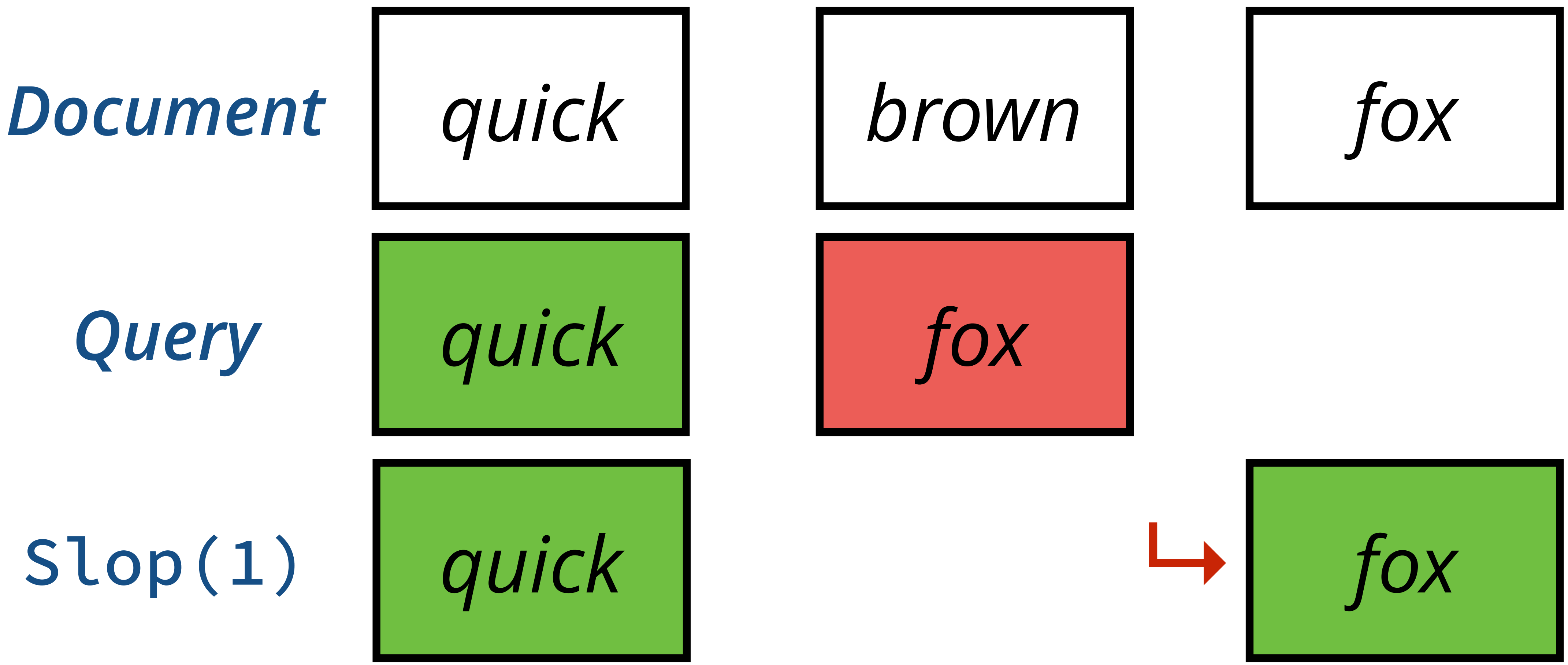
# FLEXIBLE PHRASE MATCHING

```
"query": {
  "match_phrase": {
    "title": {
      "query": "quick fox",
      "slop": 1
    }
  }
}
```

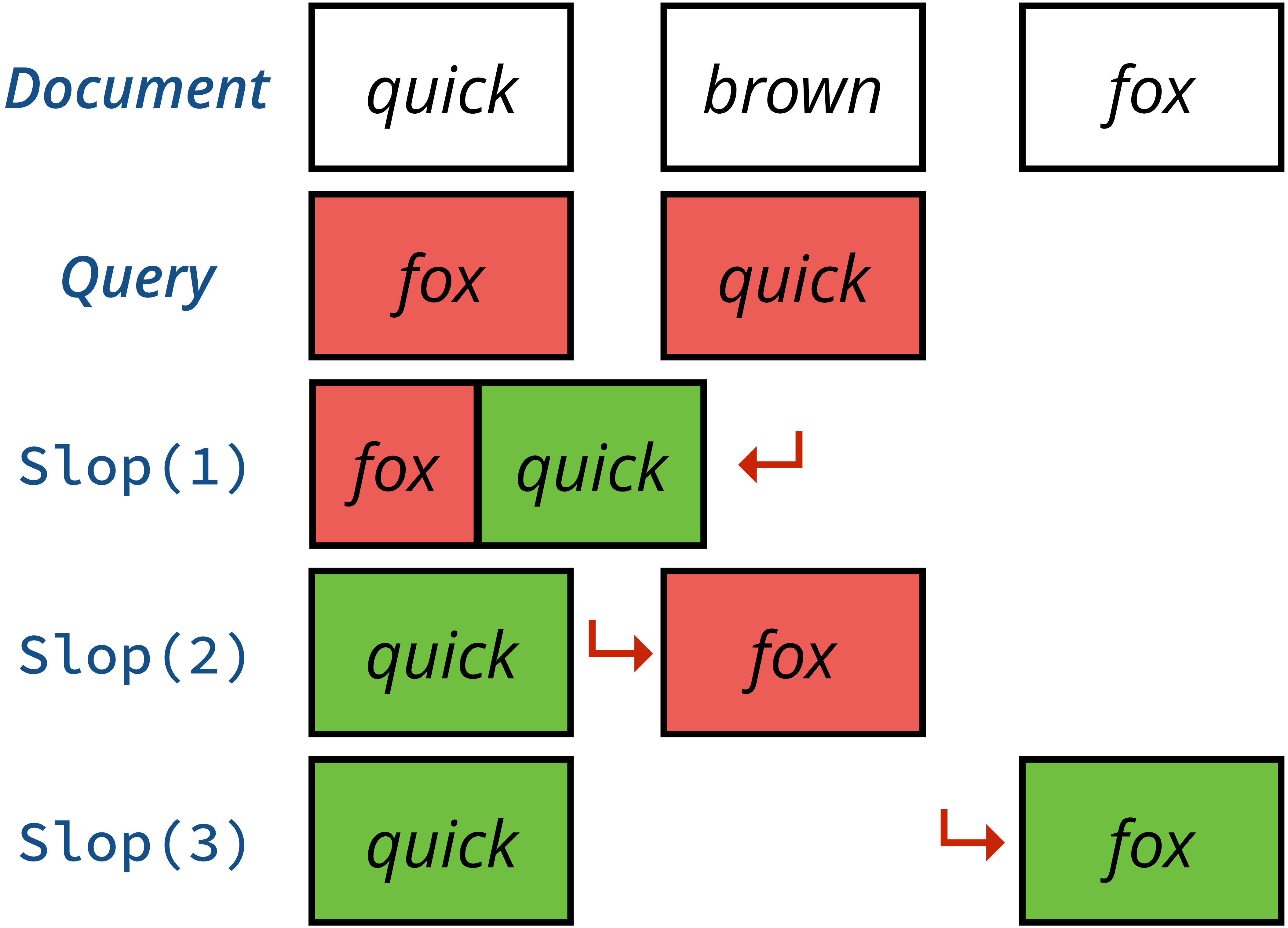
# SLOP MATCHING

- How *many times* you are allowed to *move a term* in order to make the query and document match?
  - `Slop(n)`

# SLOP MATCHING



# SLOP MATCHING



# 3-5 FUZZY MATCHING

# FUZZY MATCHING

- ***quick*** *brown fox* → ***fast*** *brown fox**es***
- *Johnn**y** Walker* → *Johnn**ie** Walker*
- *S**hc**warzen**n**eger* → *S**ch**warzene**g**ger*

# DAMERAU-LEVENSHTEIN EDIT DISTANCE

- One-character edits:
  - *Substitution*
  - *Insertion*
  - *Deletion*
  - *Transposition* of two adjacent characters

# DAMERAU-LEVENSHTEIN EDIT DISTANCE

- One-character *substitution*:
  - fox → box

# DAMERAU-LEVENSHTEIN EDIT DISTANCE

- *Insertion* of a new character:
  - sic → sick

# DAMERAU-LEVENSHTEIN EDIT DISTANCE

- *Deletion* of a character:
  - black → back

# DAMERAU-LEVENSHTEIN EDIT DISTANCE

- *Transposition* of two adjacent characters:
  - star → tsar

# DAMERAU-LEVENSHTEIN EDIT DISTANCE

- Converting ***bieber*** into ***beaver***
  1. *Substitute*: bieber → biever
  2. *Substitute*: biever → baever
  3. *Transpose*: baever → beaver
- ***Edit distance*** of 3

# FUZINESS

- *80% of human misspellings* have an Edit Distance of 1
- Elasticsearch supports a *maximum* Edit Distance of 2
- `fuziness` operator

# FUZZINESS EXAMPLE

```
./part-3/load-surprise-data.sh
```

# QUERY WITHOUT FUZZINESS

```
GET /example/surprise/_search

{
  "query": {
    "match": {
      "text": {
        "query": "surprize"
      }
    }
  }
}
```

# QUERY WITHOUT FUZZINESS

```
curl -X GET -d
@part-3/surprize-query.json
'localhost:9200/example/
surprise/_search?pretty'
```

# QUERY WITHOUT FUZZINESS

```
"hits": {
  "total": 0,
  "max_score": null,
  "hits": [ ]
}
```

# QUERY WITH FUZZINESS

```
GET /example/surprise/_search

{
  "query": {
    "match": {
      "text": {
        "query": "surprize",
        "fuzziness": "1"
      }
    }
  }
}
```

# QUERY WITH FUZZINESS

```
curl -X GET -d
@part-3/surprize-fuzzy-
query.json
'localhost:9200/example/
surprise/_search?pretty'
```

# QUERY WITH FUZZINESS


```
"hits": [ {
  "_index": "example",
  "_type": "surprise",
  "_id": "1",
  "_score": 0.19178301,
  "_source":{ "text": "Surprise me!"}
}]
```

# AUTO-FUZINESS

- 0 for strings of *one* or *two* characters
- 1 for strings of *three*, *four* or *five* characters
- 2 for strings of *more than five* characters

ThoughtWorks®

# PART 4

*Data modeling*

# 4-1 INSIDE A CLUSTER

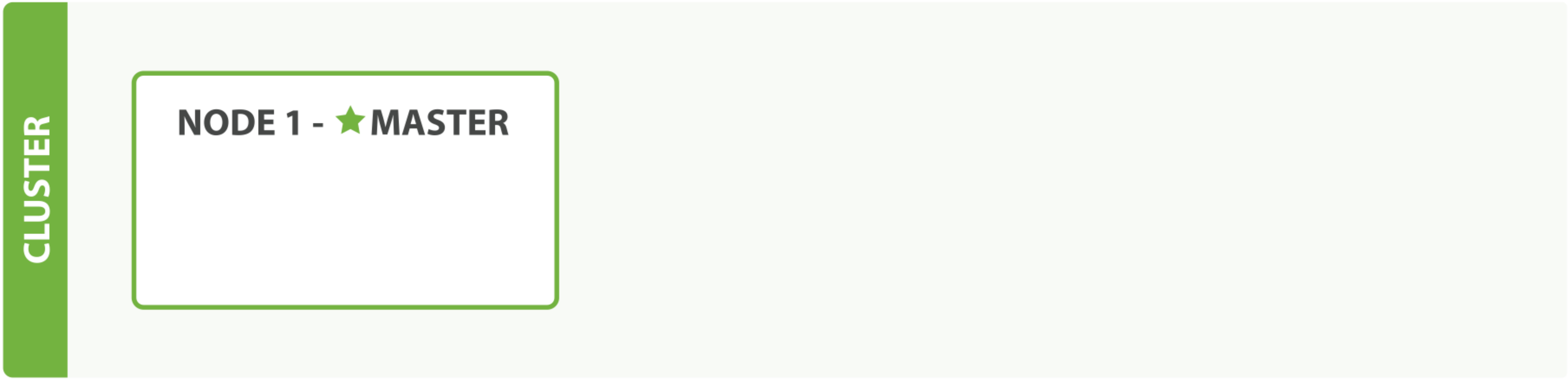
# NODES AND CLUSTERS

- A ***node*** is a machine running Elasticsearch
- A cluster is a set of ***nodes*** in the same network and with the same ***cluster name***

# SHARDS

- A node stores data inside its ***shards***
- Shards are the ***smallest unit*** of ***scale*** and ***replication***
- Each shard is a completely independent ***Lucene index***

# AN EMPTY CLUSTER



# CLUSTER HEALTH

```
GET /_cluster/health
```

# CLUSTER HEALTH

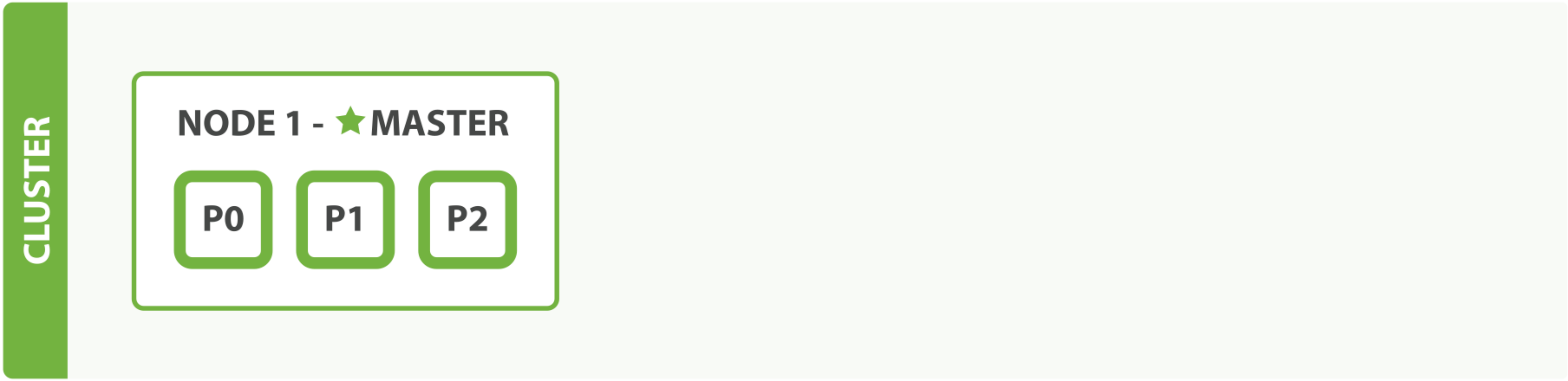

```
"cluster_name": "elasticsearch",
"status": "green",
"number_of_nodes": 1,
"number_of_data_nodes": 1,
"active_primary_shards": 0,
"active_shards": 0,
"relocating_shards": 0,
"initializing_shards": 0,
"unassigned_shards": 0
```

# ADD AN INDEX

```
PUT /blogs

"settings": {
  "number_of_shards": 3,
  "number_of_replicas": 1
}
```

# ADD AN INDEX



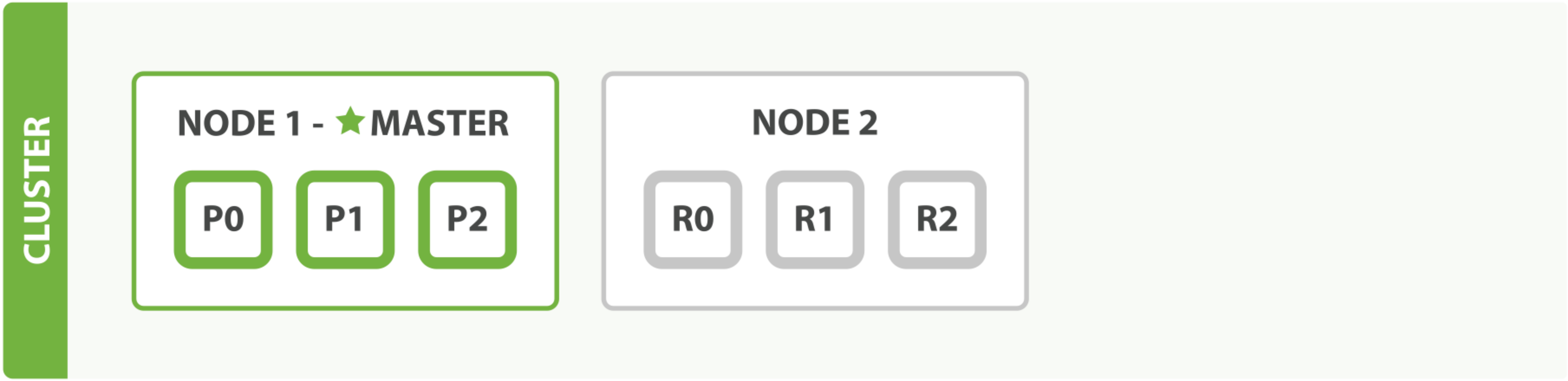
# CLUSTER HEALTH

```
GET /_cluster/health
```

# CLUSTER HEALTH

```
"cluster_name": "elasticsearch",
"status": "yellow",
"number_of_nodes": 1,
"number_of_data_nodes": 1,
"active_primary_shards": 3,
"active_shards": 3,
"relocating_shards": 0,
"initializing_shards": 0,
"unassigned_shards": 3
```

# ADD A BACKUP NODE



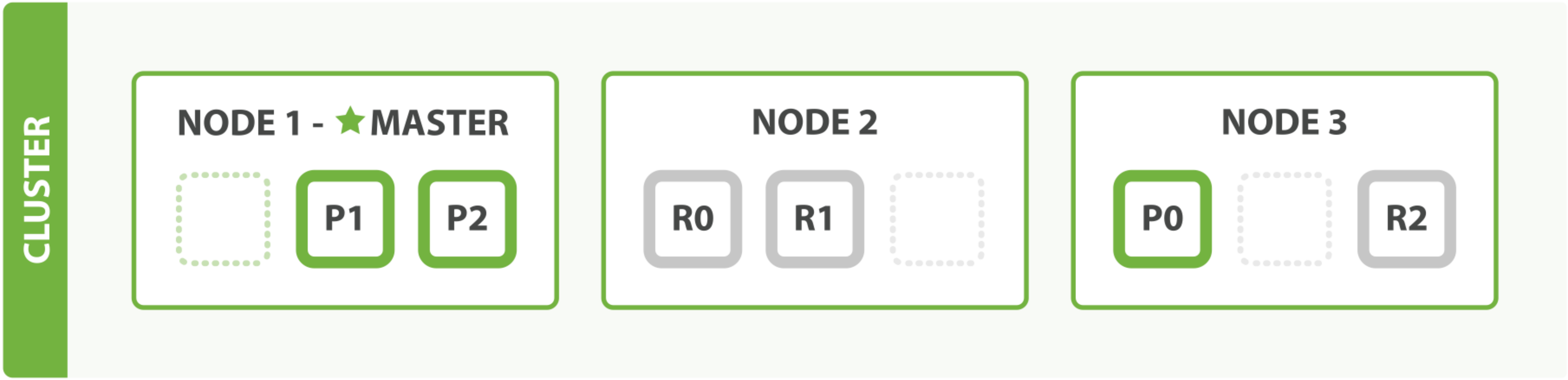
# CLUSTER HEALTH

```
GET /_cluster/health
```

# CLUSTER HEALTH

```
"cluster_name": "elasticsearch",
"status": "green",
"number_of_nodes": 2,
"number_of_data_nodes": 2,
"active_primary_shards": 3,
"active_shards": 6,
"relocating_shards": 0,
"initializing_shards": 0,
"unassigned_shards": 0
```

# THREE NODES

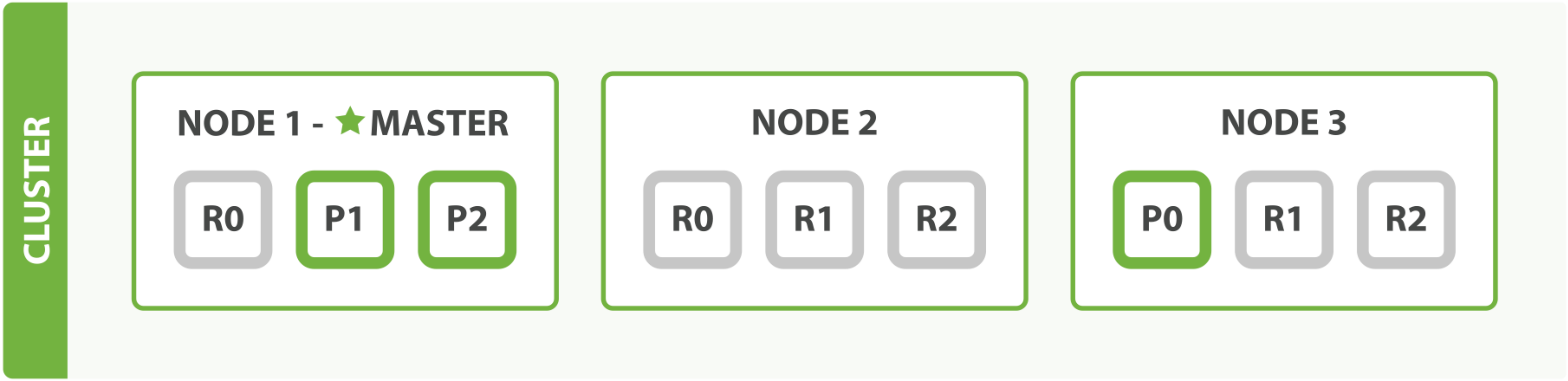


# INCREASING THE NUMBER OF REPLICAS

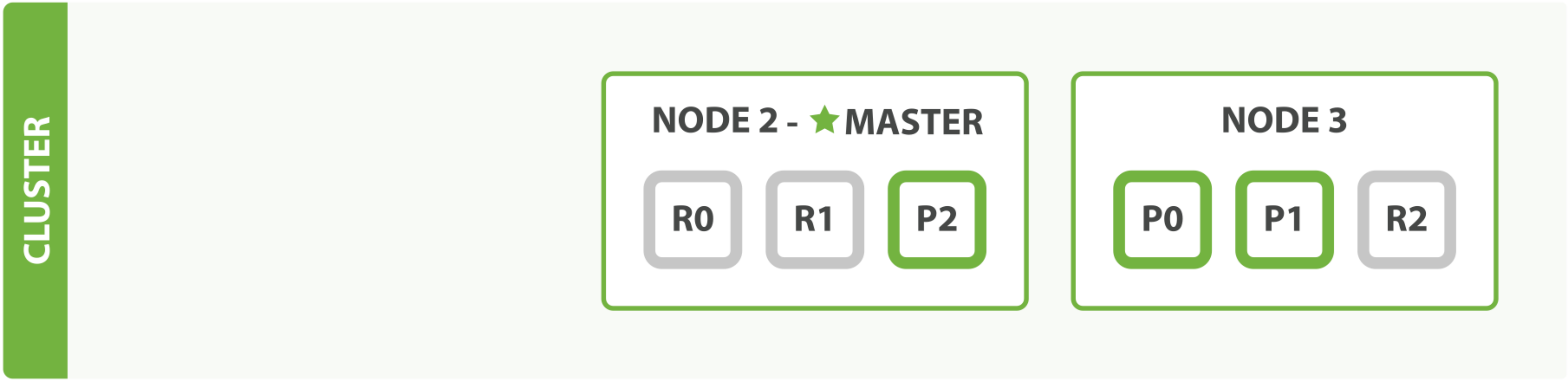
```
PUT /blogs

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

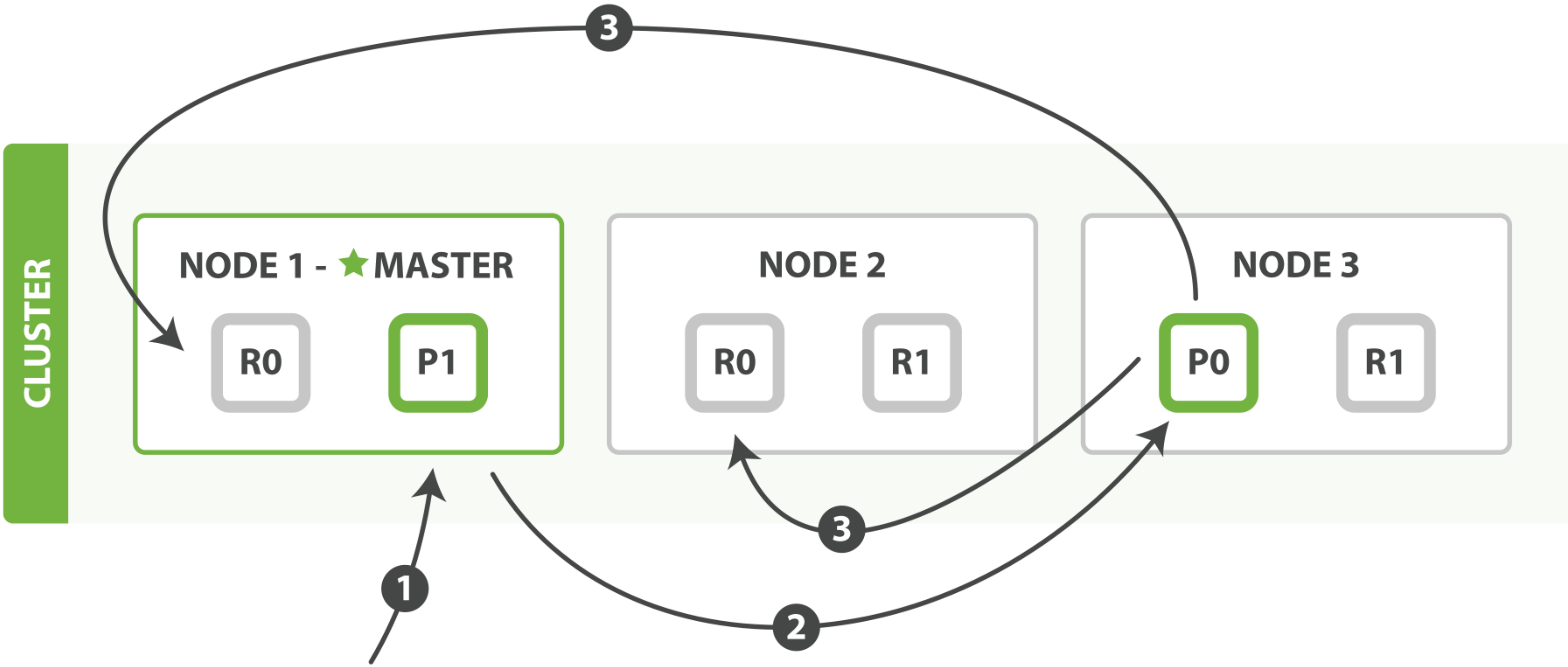
# INCREASING THE NUMBER OF REPLICAS



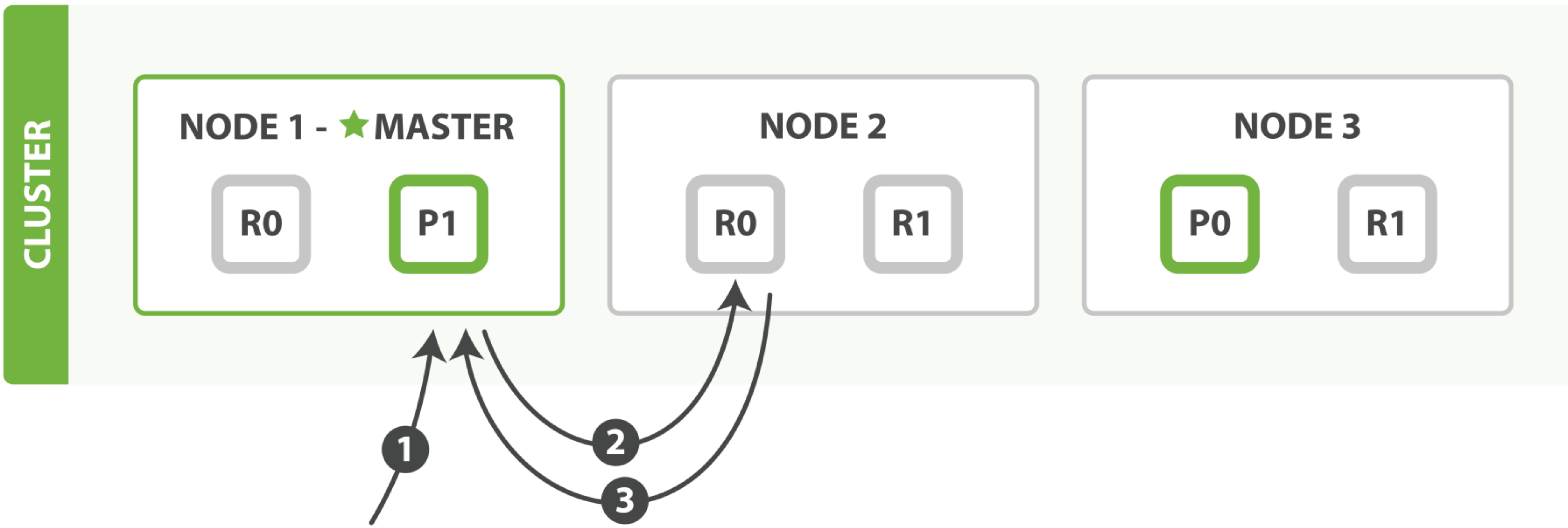
# NODE 1 FAILS



# CREATING, INDEXING AND DELETING A DOCUMENT



# RETRIEVING A DOCUMENT



ThoughtWorks®

# 4-2 RELATIONSHIPS

# RELATIONSHIPS MATTER

- Blog Posts ↔ Comments
- Bank Accounts ↔ Transactions
- Orders ↔ Items
- Directories ↔ Files
- …

# SQL DATABASES

- Entities have an ***unique primary key***
- *Normalization*:
  - Entity data is stored only once
  - Entities are referenced by primary key
- Updates happen in ***only one place***

# SQL DATABASES

- Entities are *joined* at query time

```
SELECT Customer.name, Order.status
FROM Order, Customer
WHERE Order.customer_id = Customer.id
```

# SQL DATABASES

- Changes are *ACID*
  - Atomicity
  - Consistency
  - Isolation
  - Durability

# ATOMICITY

- If one part of the transaction fails, ***the entire transaction fails***
- ...even in the event of power failure, crashes or errors
- *"all or nothing"*

# CONSISTENCY

- Any transaction will bring the database from ***one valid state to another***
- State must be valid according to all defined ***rules***:
  - Constraints
  - Cascades
  - Triggers

# ISOLATION

- The *concurrent execution* of transactions results in the same state that would be obtained if transactions were *executed serially*
- *Concurrency Control*

# DURABILITY

- A transaction ***will remain committed***
- ...even in the event of power failure, crashes or errors
- *Non-volatile memory*

# SQL DATABASES

- Joining entities at query time is *expensive*
- *Impractical* with multiple nodes

# ELASTICSEARCH

- Treats the world as *flat*
- An index is a flat collection of *independent documents*
- A single document should contain all information to *match a search request*

# ELASTICSEARCH

- ACID support for changes on ***single documents***
- No ACID transactions on ***multiple documents***

# ELASTICSEARCH

- Indexing and searching are *fast* and *lock-free*
- Massive amounts of data can be spread across *multiple nodes*

- But we need *relationships*!

# ELASTICSEARCH

- Application-side joins
- Data denormalization
- Nested objects
- Parent/child relationships

ThoughtWorks®

# 4-3 APPLICATION-SIDE JOINS

# APPLICATION-SIDE JOINS

- Emulates a relational database
- Joins at *application level*
- `(index, type, id)` = *primary key*

# EXAMPLE

```
PUT /example/user/1

{
  "name":  "John Smith",
  "email": "john@smith.com",
  "born":  "1970-10-24"
}
```

# EXAMPLE

```
PUT /example/blogpost/2

{
  "title": "Relationships",
  "body":  "It's complicated",
  "user":  1
}
```

# EXAMPLE

- `(example, user, 1)` = *primary key*
- Store only the ***id***
- ***Index*** and ***type*** are hard-coded into the application logic

# EXAMPLE

```
GET /example/blogpost/_search

"query": {
  "filtered": {
    "filter": {
      "term": { "user": 1 }
    }
  }
}
```

# EXAMPLE

- Blogposts written by *"John"*:
  - ***Find ids*** of users with name *"John"*
  - Find blogposts that ***match the user ids***

# EXAMPLE

```
GET /example/user/_search

"query": {
  "match": {
    "name": "John"
  }
}
```

# EXAMPLE

- For each *user id* from the first query:

```
GET /example/blogpost/_search

"query": {
  "filtered": {
    "filter": {
      "term": { "user": <ID> }
    }
  }
}
```

# ADVANTAGES

- ☐ Data is ***normalized***
- ☐ Change user data ***in just one place***

# DISADVANTAGES

- Run *extra queries* to join documents
- We could have *millions* of users named “John”
- *Less efficient* than SQL joins:
  - Several API requests
  - Harder to optimize

# WHEN TO USE

- First entity has a ***small number*** of documents and they ***hardly change***
- First query results can be ***cached***

# 4-4 DATA DENORMALIZATION

# DATA DENORMALIZATION

- *No joins*
- Store ***redundant copies*** of the data you need to query

# EXAMPLE

```
PUT /example/user/1

{
  "name":  "John Smith",
  "email": "john@smith.com",
  "born":  "1970-10-24"
}
```

# EXAMPLE

```
PUT /example/blogpost/2

{
   "title": "Relationships",
   "body":  "It's complicated",
   "user":  {
     "id":    1,
     "name": "John Smith"
   }
}
```

# EXAMPLE

```
GET /example/blogpost/_search

"query": {
  "bool": {
    "must": [
      { "match": {
        "title": "relationships" }},
      { "match": {
        "user.name": "John" }}
]}}
```

# ADVANTAGES

- *Speed*
- No need for expensive joins

# DISADVANTAGES

- Uses more disk space (cheap)
- Update the same data in ***several places***
  - ***scroll*** and ***bulk*** APIs can help
- *Concurrency issues*
  - *Locking* can help

# WHEN TO USE

- Need for *fast search*
- Denormalized data *does not change* very often

# 4-5 NESTED OBJECTS

# MOTIVATION

- Elasticsearch supports *ACID* when updating ***single documents***
- Querying related data in the same document is ***faster*** (no joins)
- We want to ***avoid denormalization***

# THE PROBLEM WITH MULTILEVEL OBJECTS

```
PUT /example/blogpost/1

{
  "title": "Nest eggs",
  "body": "Making money...",
  "tags": [ "cash", "shares" ],
  "comments": […]
}
```

# THE PROBLEM WITH MULTILEVEL OBJECTS

```
[{
  "name": "John Smith",
  "comment": "Great article",
  "age": 28, "stars": 4,
  "date": "2014-09-01"
}, {
  "name": "Alice White",
  "comment": "More like this",
  "age": 31,"stars": 5,
  "date": "2014-10-22"
}]
```

# THE PROBLEM WITH MULTILEVEL OBJECTS

```
GET /example/blogpost/_search

"query": {
  "bool": {
    "must": [
      {"match": {"name": "Alice"}},
      {"match": {"age": "28"}}
]}}
```

# THE PROBLEM WITH MULTILEVEL OBJECTS

```
[{
  "name": "John Smith",
  "comment": "Great article",
  "age": 28, "stars": 4,
  "date": "2014-09-01"
}, {
  "name": "Alice White",
  "comment": "More like this",
  "age": 31,"stars": 5,
  "date": "2014-10-22"
}]
```

# THE PROBLEM WITH MULTILEVEL OBJECTS

- Alice is 31, not 28!
- It matched the age of John
- This is because indexed documents are stored as a *flattened dictionary*
- The correlation between *Alice* and 31 is *irretrievably lost*

# THE PROBLEM WITH MULTILEVEL OBJECTS

```
{"title": [eggs, nest],
 "body": [making, money],
 "tags": [cash, shares],
 "comments.name":
   [alice, john, smith, white],
 "comments.comment":
   [article, great, like, more, this],
 "comments.age": [28, 31],
 "comments.stars": [4, 5],
 "comments.date":
   [2014-09-01, 2014-10-22]}
```

# NESTED OBJECTS

- Nested objects are indexed as ***hidden separate documents***
- Relationships are ***preserved***
- Joining nested documents is ***very fast***

# NESTED OBJECTS

```
{"comments.name": [john, smith],
 "comments.comment": [article, great],
 "comments.age": [28],
 "comments.stars": [4],
 "comments.date": [2014-09-01]}

{"comments.name": [alice, white],
 "comments.comment": [like, more, this],
 "comments.age": [31],
 "comments.stars": [5],
 "comments.date": [2014-10-22]}
```

# NESTED OBJECTS

```
{
 "title": [eggs, nest],
 "body": [making, money],
 "tags": [cash, shares]
}
```

# NESTED OBJECTS

- Need to be enabled by *updating the mapping* of the index

# MAPPING A NESTED OBJECT

```
PUT /example

"mappings": {
  "blogpost": { "properties": {
    "comments": { "type": "nested",
      "properties": {
        "name": {"type": "string"},
        "comment": {"type": "string"},
        "age": {"type": "short"},
        "stars": {"type":"short"},
        "date": {"type": "date"}
}}}}}
```

# QUERYING A NESTED OBJECT

```
GET /example/blogpost/_search

"query": {
  "bool": {
    "must": [
      {"match": {"title": "eggs"}}
      {"nested": <NESTED QUERY>}
    ]
  }
}
```

# NESTED QUERY

```
"nested": {
  "path": "comments",
  "query": {
    "bool": {
      "must": [
        {"match":
          {"comments.name": "john"}},
        {"match":
          {"comments.age": 28}}
]}}}
```

# THERE'S MORE

- ☐ Nested *filters*
- ☐ Nested *aggregations*
- ☐ *Sorting* by nested fields

# ADVANTAGES

- ☐ *Very fast* query-time joins
- ☐ *ACID support* (single documents)
- ☐ Convenient search using *nested queries*

# DISADVANTAGES

- To add, change or delete a nested object, the ***whole document*** must be ***reindexed***
- Search requests return the ***whole document***

# WHEN TO USE

- When there is one main entity with a ***limited number*** of ***closely related entities***
  - Ex: blogposts and comments
- Inefficient if there are ***too many*** nested objects

ThoughtWorks®

# 4-6 PARENT-CHILD RELATIONSHIP

# PARENT-CHILD RELATIONSHIP

- *One-to-many* relationship
- Similar to the nested model
- Nested objects live in the *same document*
- Parent and children are *completely separate documents*

# EXAMPLE

- Company with ***branches*** and ***employees***
- Branch is the ***parent***
- Employee are ***children***

# EXAMPLE

```
PUT /company

"mappings": {
  "branch": {},
  "employee": {
    "_parent": {
      "type": "branch"
    }
  }
}
```

# EXAMPLE

```
PUT /company/branch/london

{
   "name": "London Westminster",
   "city": "London",
   "country": "UK"
}
```

# EXAMPLE

```
PUT /company/employee/1?
parent=london

{
  "name": "Alice Smith",
  "born": "1970-10-24",
  "hobby": "hiking"
}
```

# FINDING PARENTS BY THEIR CHILDREN

```
GET /company/branch/_search
"query": {
  "has_child": {
    "type": "employee",
    "query": {
      "range": {
        "born": {
          "gte": "1980-01-01" }
}}}}
```

# FINDING CHILDREN BY THEIR PARENTS

```
GET /company/employee/_search

"query": {
  "has_parent": {
    "type": "branch",
    "query": {
      "match": {
        "country": "UK" }
}}}
```

# THERE'S MORE

- `min_children` and `max_children`
- Children *aggregations*
- *Grandparents* and *grandchildren*

# ADVANTAGES

- Parent document can be updated ***without reindexing*** the children
- Child documents can be updated ***without affecting*** the parent
- Child documents can be returned in ***search results*** without the parent

# ADVANTAGES

- Parent and children live on the ***same shard***
- ***Faster*** than application-side joins

# DISADVANTAGES

- Parent document and *all of its children* must live on the *same shard*
- 5 to 10 times *slower than nested queries*

# WHEN TO USE

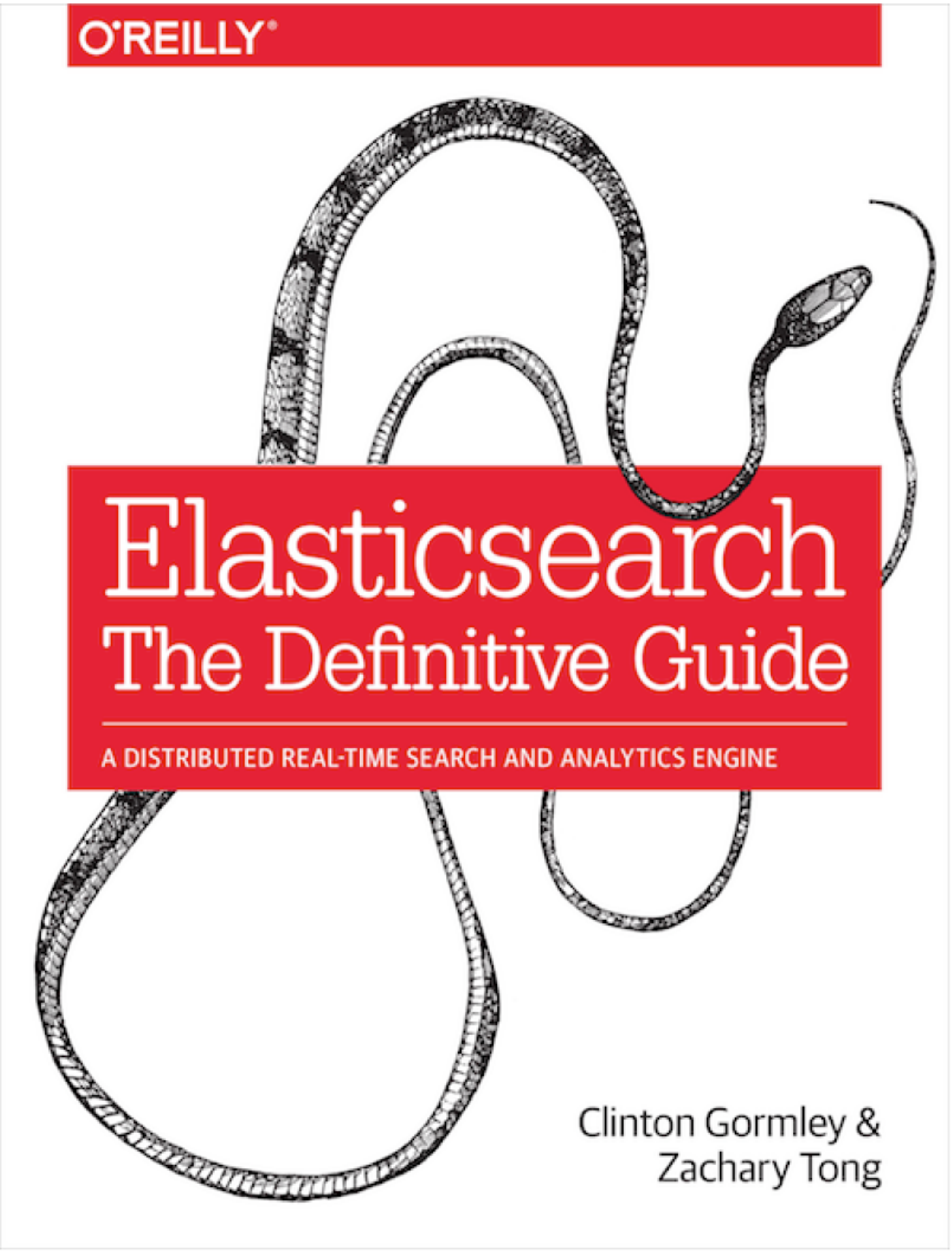
- ***One-to-many*** relationships
- When ***index-time*** is more important than ***search-time*** performance
- Otherwise, use ***nested objects***

ThoughtWorks®

# REFERENCES

# MAIN REFERENCE

- *Elasticsearch, The Definitive guide*
  - *Gormley & Tong*
  - *O'Reilly*



# OTHER REFERENCES

- *"Jepsen: simulating network partitions in DBs"*, http://github.com/aphyr/jepsen
- *"Call me maybe: Elasticsearch 1.5.0"*, http://aphyr.com/posts/323-call-me-maybe-elasticsearch-1-5-0
- *"Call me maybe: MongoDB stale reads"*, http://aphyr.com/posts/322-call-me-maybe-mongodb-stale-reads

# OTHER REFERENCES

- *"Elasticsearch Data Resiliency Status"*, http://www.elastic.co/guide/en/elasticsearch/resiliency/current/index.html
- *"Solr vs. Elasticsearch — How to Decide?"*, http://blog.sematext.com/2015/01/30/solr-elasticsearch-comparison/

# OTHER REFERENCES

- *"Changing Mapping with Zero Downtime"*, http://www.elastic.co/blog/changing-mapping-with-zero-downtime

# THANK YOU


***Felipe Dornelas***
*felipedornelas.com*
*@felipead*


ThoughtWorks®