



Elasticsearch Workshop

December 2022, Niš

<https://github.com/deavensoft/es-workshop-deavensoft>

Agenda

Introduction

What is a Document?

Basic Searching

Mapping and Analysis

Query DSL

Sorting & Relevance

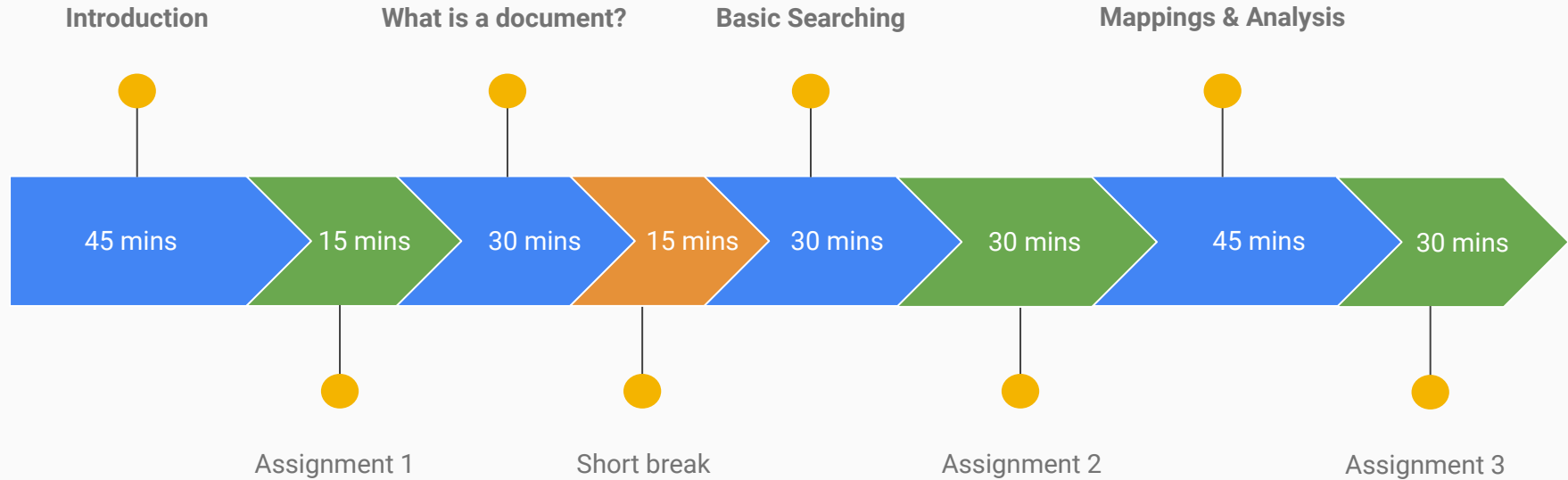
Dealing with Human Language

Aggregations

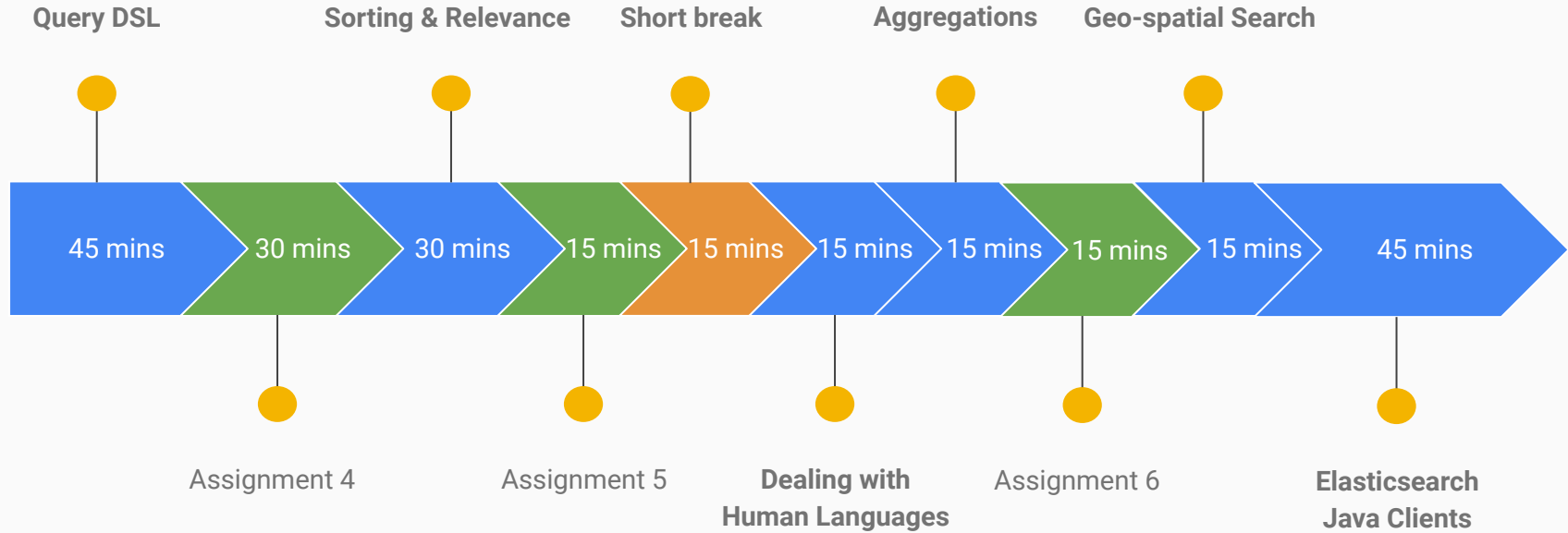
Geo-spatial Search

Elasticsearch Java Client

Morning



Afternoon



Introduction

What is Elasticsearch?

Elasticsearch is an open-source, RESTful, distributed search and analytics engine built on Apache Lucene

Released in 2010 and quickly became the most popular search engine

Commonly used for log analytics, full-text search and operational intelligence

Easy-to-use search APIs



Elasticsearch Benefits



Fast



**Near Real-time
Index Updates**



Easy-to-use APIs



**Support for your
Favorite Development
Language**



**Complementary Tooling
and Plug-ins**

Introduction

Basic Concepts

Cluster

Collection of one or more nodes (servers) holding entire data and providing federated indexing and search capabilities

Identified by unique cluster name (default name: `elasticsearch`)

Node

A single *server* that is part of a cluster

Identified by a name (default: *random UUID*). Name is important for admin purposes and to know which server in a network corresponds to which node

Can join only one *Cluster* (if not set, auto-joins `elasticsearch` cluster)

Index

Elasticsearch index is a collection of documents that have somewhat similar characteristics

E.g. you can have index for customer data, another index for a product catalog etc.

Identified by name (all lowercase)

In a single cluster, you can define as many indexes as you want

Document

A document is a basic unit of information that can be indexed

E.g. you can have document for a single customer, another document for a single product

Documents expressed as JSON

Within an index you can store as many documents as you want

Shards

An *Index* can store a large amount of data, which could not fit into a single node (server)

Another problem: single node could be too slow to serve search requests

Shards adds ability to subdivide index into multiple pieces, where each Shard is an “independent” index (Lucene)

Shards can run on any *Node*

Sharding benefits:

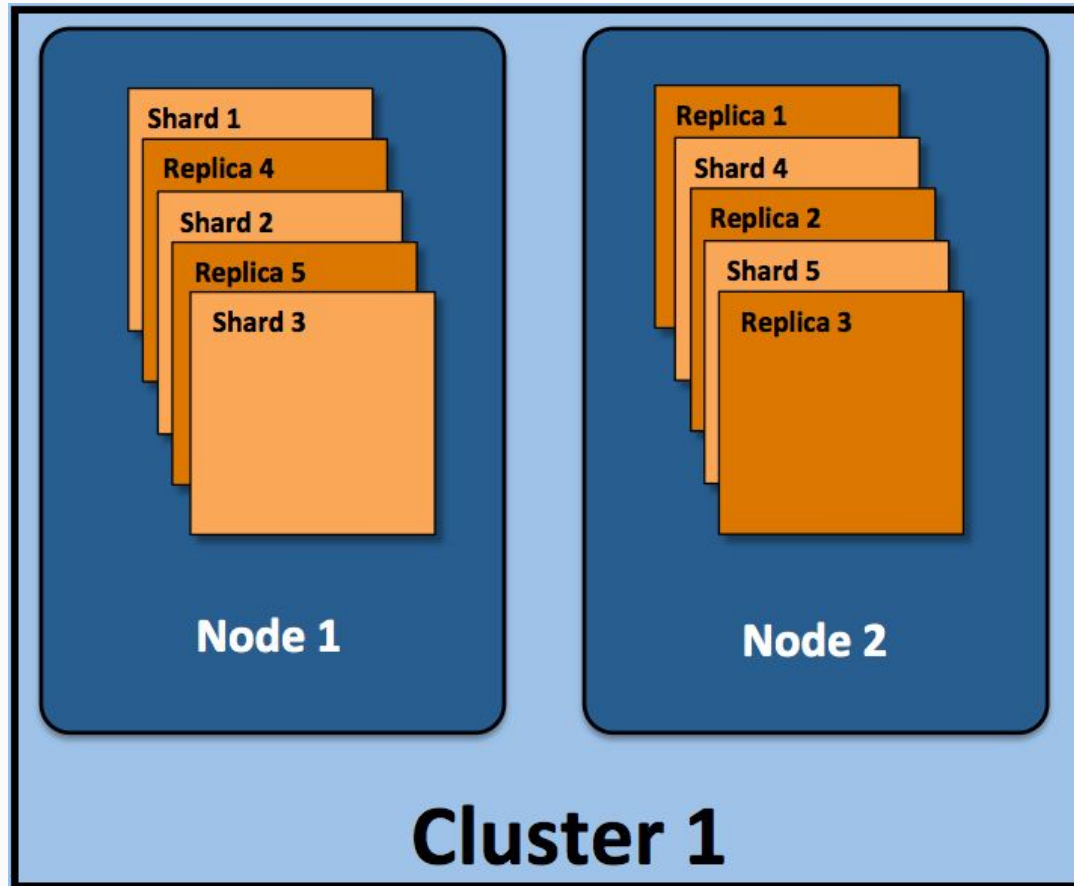
- Allows to horizontally split/scale your content volume
- Allows to distribute and parallelize operations across Shards

Replicas

To handle failures and high availability Elasticsearch can make one or more copies of Index's Shards called *Replicas*

Replica shard is never allocated on the same Node as original/primary Shard

Basic Concepts



Near-real time (NRT)

Elasticsearch is a near real time search platform (as Apache Lucene)

There is a slight latency from when you index a document until it becomes searchable

Normally one second

Type

A Type used to be a logical category/partition of your index to allow you to store different types of documents in the same index

Deprecated in 6.0.0

Indices created in Elasticsearch 6.0.0 or later may only contain a single mapping type

Will be removed in future versions

Introduction

Apache Lucene

About Apache Lucene?

Apache Lucene is a high-performance, full-featured text search engine library written entirely in Java.

It is a technology suitable for nearly any application that requires full-text search, especially cross-platform



Scalable, High-Performance Indexing

- over 150GB/hour on modern hardware
- small RAM requirements -- only 1MB heap
- incremental indexing as fast as batch indexing
- index size roughly 20-30% the size of text indexed

Powerful, Accurate and Efficient Search Algorithms

- ranked searching -- best results returned first
- many powerful query types: phrase queries, wildcard queries, proximity queries, range queries and more
- fielded searching (e.g. title, author, contents)
- sorting by any field
- multiple-index searching with merged results
- allows simultaneous update and searching
- flexible faceting, highlighting, joins and result grouping
- fast, memory-efficient and typo-tolerant suggesters
- pluggable ranking models, including the Vector Space Model and Okapi BM25
- configurable storage engine (codecs)

Cross-Platform Solution

- Available as Open Source software under the Apache License which lets you use Lucene in both commercial and Open Source programs
- 100%-pure Java
- Implementations in other programming languages available that are index-compatible

Apache Lucene is a DNA of most of the search technologies in today's market. Even the machine learning languages like Mahout, Famous web crawler like Nutch, etc are using Lucene.

Elasticsearch and Apache Lucene

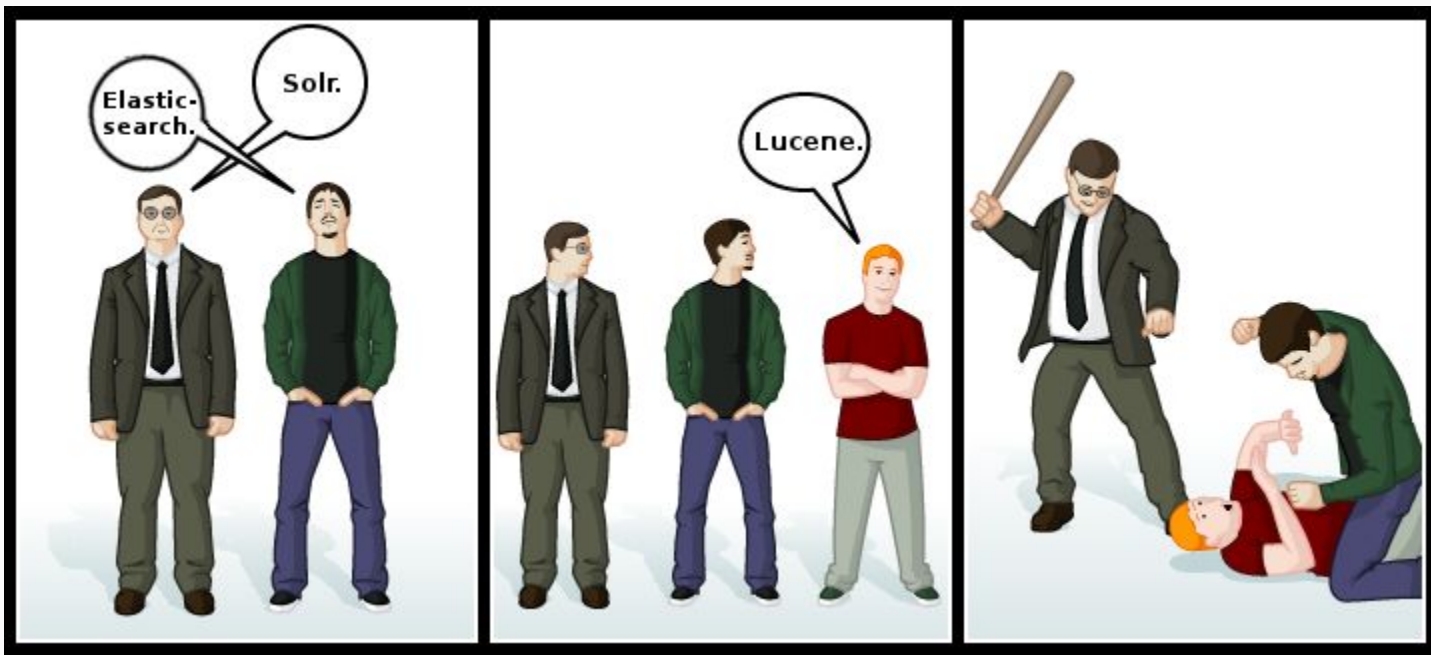
Apache Lucene is doing the heavy lifting for Elasticsearch. But...

Lucene is a java library. You can include it in your project and refer to its functions using function calls.

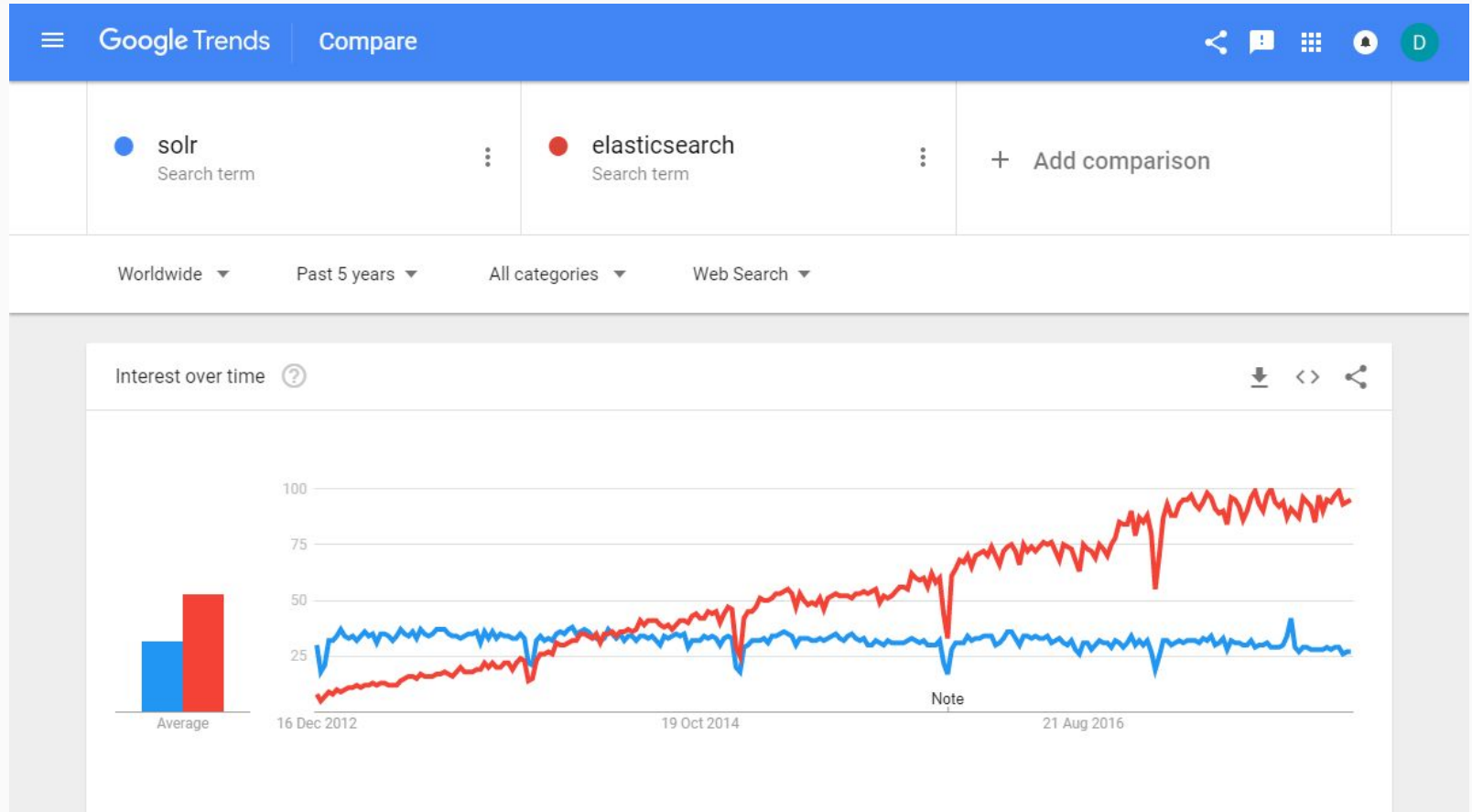
Elasticsearch is built on top of Lucene and provides:

- *a JSON based REST API* to refer to Lucene features
- *a distributed system* on top of Lucene (abstraction of distributed structure)
- *other supporting features* like thread-pool, queues, node/cluster monitoring API, data monitoring API, Cluster management, etc.

Hipster in the Hood



SOLR vs Elasticsearch



Introduction

Installation

Installation Options

- Download binaries
- Docker images
- Cloud options (AWS, GCP)

DEMO

Elasticsearch - The First Bite

Installing Elasticsearch

Execute some commands
and queries against
running Elasticsearch
instance



Elastic Stack



Kibana gives shape to your data and is the extensible user interface for configuring and managing all aspects of the Elastic Stack



Beats is a platform for lightweight shippers that send data from edge machines to Logstash and Elasticsearch



Logstash is a dynamic data collection pipeline with an extensible plugin ecosystem and strong Elasticsearch synergy

Assignment 1 - Installation

Install **Elasticsearch** and **Kibana** on a local machine.

The most optimum way is to install the environment on the local machine using **docker-compose**.

Alternatively ,follow the next guides to install from binaries:

- <https://www.elastic.co/downloads/elasticsearch>
- <https://www.elastic.co/downloads/kibana>

You will also need a **Java 8+** installed on your machine. If needed, go to <http://www.oracle.com/technetwork/java/javase/downloads/index.html> and install it

What is a Document?

JSON

Most entities in most applications can be serialized into a **JSON object**

A **key** is the name of a field or property

A **value** can be a string, a number, a Boolean, another object, an array of values, or some other specialized type such as a string representing a date or an object representing a geolocation

Note: Field names can be any valid string, but may not include periods

JSON Document with Objects

```
{
  "name":      "John Smith",
  "age":       42,
  "confirmed": true,
  "join_date": "2014-06-01",
  "home": {
    "lat":     51.5,
    "lon":     0.1
  },
  "accounts": [
    {
      "type": "facebook",
      "id":   "johnsmith"
    },
    {
      "type": "twitter",
      "id":   "johnsmith"
    }
  ]
}
```

Object

Object is just a JSON object
Objects may contain other objects

Document

In Elasticsearch, *document* has a specific meaning - it refers to the top-level or *root objects* that is *serialized into JSON* and stored under a *unique ID*

Document Metadata

Document also has **metadata** - information *about* the document.

The three required metadata elements are as follows:

<code>_index</code>	Where the document lives
<code>_id</code>	The unique identifier for the document

Note: `_index` and `_id` uniquely identify the document

Index API - PUT

Documents are *indexed* — stored and made searchable — by using the **index API**:

```
PUT /{index}/_doc/{id}
{
  "field": "value",
  ...
}
```

Example:

```
PUT /website/_doc/123
{
  "title": "My first blog entry",
  "text": "Just trying this out...",
  "date": "2014/01/01"
}
```

Index API - POST

If our data doesn't have a natural ID, Elasticsearch autogenerate one for us:

```
POST /{index}/_doc/  
{  
  "field": "value",  
  ...  
}
```

Example:

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

-> generates:

```
"_id": "AVFgSgVHUP18jI2wRx0w"
```


Index API - GET (Retrieving a Document)

To get the document, use the same approach but with HTTP verb **GET**:

```
GET /website/_doc/123
```

Result:

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

Index API - HEAD

Instead of GET you can use **HEAD** to check whether a document exists:

```
HEAD /website/_doc/123
```

```
HEAD /website/_doc/999
```

Result (if exists):

HTTP/1.1 200 OK

Content-Type: text/plain; charset=UTF-8

Content-Length: 0

Result (if does not exists):

HTTP/1.1 404 Not Found

Content-Type: text/plain; charset=UTF-8

Content-Length: 0

DEMO

Indices, Documents and Types

Creating an Index

Indexing some data

Retrieving data



Updating a Document

Documents in Elasticsearch are *immutable*; we cannot change them

We can **reindex** or replace the document with all its data

1. delete the old document
2. index a new document

Partial updates are also supported:

1. Retrieve the JSON from the old document
2. Change it
3. Delete the old document
4. Index a new document

Deleting a Document

Similarly, to delete a document use HTTP verb **DELETE**:

```
DELETE /website/_doc/123
```

*Note: Deleting a document doesn't immediately remove the document from disk;
it just marks it as deleted*

Optimistic Concurrency Control

Elasticsearch is *distributed* and also *asynchronous* and *concurrent*

When documents are created, updated, or deleted, the new version of the document has to be replicated to other nodes in the cluster. Updates may arrive *out of sequence*.

`_version` metadata is incremented whenever a document is changed

When updating a document `_version` can be used to prevent conflicting changes

Other APIs for Document Manipulations

`_update` Partial updates (even using scripts)

`_mget` Retrieving Multiple Documents

`_bulk` Allows making multiple *create, index, update* or *delete* requests in a single step

Break



15 minutes

Basic Searching

Elasticsearch Power

We saw how Elasticsearch can be used as a simple NoSQL-style distributed document store.

We can throw JSON documents at Elasticsearch and retrieve each one by ID

But the real power of Elasticsearch lies in its ability to make sense out of chaos

Elasticsearch not only *stores* the document, but also *indexes* the content of the document in order to make it searchable.

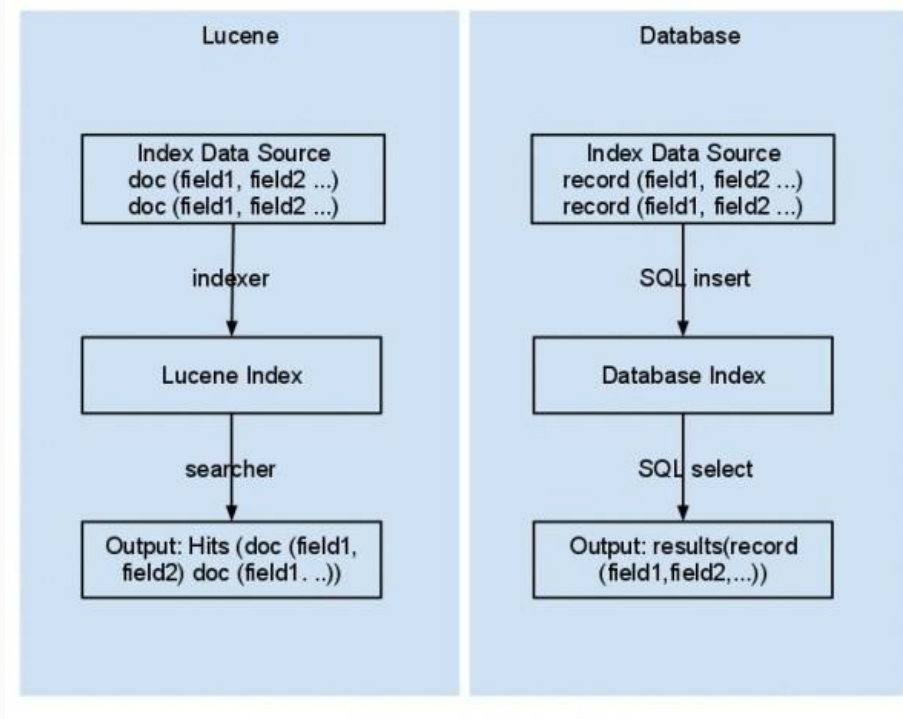
Apache Lucene vs RDBMS: Full-Text Search Mechanism

The RDBMS database index is not designed for the full-text index

The database query with fuzzy terms, LIKE, can harm performance

If you need more than one fuzzy matching words:

like "% keyword1%" and like "% keyword2%" ... the efficiency also will be damaged more



Search Queries

- A *structured query* on concrete fields like `gender` or `age`, sorted by a field like `join_date`, similar to SQL query
- A *full-text query*, which finds all documents matching the search keywords, and returns them sorted by *relevance*
- A combination of the two

The Empty Search

GET /_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...",
          "user_id": 2
        }
      },
      ... 9 RESULTS REMOVED ...
    ],
    "max_score" : 1
  },
  "took" : 4,
  "_shards" : {
    "failed" : 0,
    "successful" : 10,
    "total" : 10
  },
  "timed_out" : false
}
```

The Empty Search

`hits` - contains the total number of documents matched the query

`hits[]` - array contains matched documents - the result, where each contains:

- `_index` and `_id` of the document, plus the `_source` field

- `_score` is a relevance score,
 - measure of how well the document matches the query
 - (default sorting: by *score descending*
 - meaning most relevant documents first)

`max_score` - the highest `_score` of any document that matches the query

`took` - how many milliseconds the entire search request took to execute

`_shards` - the total number of shards that were involved in the query

DEMO

Basic Searching

Index some test data

Search the data in various
ways



Multi-index, Multi-type Searches

`/gb/_search`

Search all document in the gb index

`/gb,us/_search`

Search all documents in the gb and us indices

`/g*,u*/_search`

Search all document in any indices beginning with g or beginning with u

Looks for *john* in the `name` field and *mary* in the `tweet` field

Actual query is:

```
+name:john +tweet:mary
```

Search query specified as url-query parameter (URL encoded):

```
GET /_search?q=%2Bname%3Ajohn%2Btweet%3Amary
```

More complicated query:

```
+name:(mary john) +date:>2014-09-10 +(aggregations geo)
```

```
?q=%2Bname%3A(mary+john)+%2Bdate%3A%3E2014-09-10+%2B(aggregations+geo)
```

Query String Syntax

The query string “mini-language” is used by the Query String Query and by the `q` query string parameter in the `search` API

Parsed into a series of **terms** and **operators**

Terms can be:

single word - quick **or** brown

phrase - "quick brown"

Operators customizes the search

Field Names

`status:active`

`title:(quick OR brown)`

`title:(quick brown)`

`author:"John Smith"`

`book.*:(quick brown)`

`_missing_:title`

`_exists_:title`

Operators

Wildcards

qu?ck bro*

Regular expressions

name:/joh?n(ath[oa]n)/

Fuzziness

Uses the *Damerau-Levenshtein distance*

quikc~ brwn~ foks~

quikc~1

Proximity

"fox quick"~5

Ranges

For date, numeric or string fields

date:[2012-01-01 TO 2012-12-31]

count:[1 TO 5]

tag:{alpha TO omega}

count:[10 TO *]

date:{* TO 2012-01-01}

count:[1 TO 5}

age:>10

age:>=10

age:(>=10 AND <20)

age:(+>=10 +<20)

Operators

Boosting

`quick^2 fox`

`"john smith"^2 (foo bar)^4`

Grouping

`(quick OR brown) AND fox`

`status:(active OR pending)`

`title:(full text search)^2`

Reserved Characters

`+ - = && || > < ! () { } []`

`^ " ~ * ? : \ /`

Boolean Operators

By default, all terms are optional,
as long as one term matches

+ operator - *must* be present

- operator - *must not* be present

`quick brown +fox -news`

```
{
  "bool": {
    "must": {"match": "fox"},
    "should": { "match":
                  "quick brown" },
    "must_not": {"match": "news"}
  }
}
```

Recommendation

Lite query-string search is very powerful

Great for throwaway queries from the command line or during development

On the other hand:

- Queries can look cryptic and difficult to debug
- Allows any user to run potentially slow, heavy requests on any field, possibly exposing sensitive data or killing your cluster nodes

It is *not recommended* to expose query-string searches directly to your users

Assignment 2 - Querying using Search Lite

You are a bartender, who is happy to suggest cocktail ideas to guests in the bar



Mapping and Analysis

Exact Values Versus Full Text

Two types of data:

- Exact values (e.g. *user ID, email, age* etc.)
- Full text (textual data e.g. *tweet text, blog text* etc.)

Exact values are easy to query

Querying full-text data is much more subtle

Querying full-text

Not just: *“Does this document match the query”*

but also: *“How **well** does this document match the query?”*

In other words: **How relevant is this document to the given query?**

Querying full-text

We have no intention to match the whole full-text field exactly

Instead, we want to search *within* text fields

... and not only that...

- A search for `UK` should also return documents mentioning the `United Kingdom`.
- A search for `jump` should also match `jumped`, `jumps`, `jumping`, and perhaps even `leap`.
- `johnny walker` should match `Johnnie Walker`, and `johnnie depp` should match `Johnny Depp`.
- `fox news hunting` should return stories about hunting on Fox News, while `fox hunting news` should return news stories about fox hunting.

Inverted Index

To support full-text search:

- text is first Analyzed
- results are used to build an Inverted Index

Inverted Index is a structure, designed to allow very fast full-text searches

Tokenization

Example documents:

1. *The quick brown fox jumped over the lazy dog*
2. *Quick brown foxes leap over lazy dogs in summer*

Search for: quick brown

Term	Doc_1	Doc_2

brown	X	X
quick	X	

Total	2	1

Term	Doc_1	Doc_2

Quick		X
The	X	
brown	X	X
dog	X	
dogs		X
fox	X	
foxes		X
in		X
jumped	X	
lazy	X	X
leap		X
over	X	X
quick	X	
summer		X
the	X	

Normalizing Terms

Example documents:

1. *The quick brown fox jumped over the lazy dog*
2. *Quick brown foxes leap over lazy dogs in summer*

Quick can be lowercased to become quick.

foxes can be stemmed—reduced to its root form—to become fox. Similarly, dogs could be stemmed to dog.

jumped and leap are synonyms and can be indexed as just the single term jump.

Term	Doc_1	Doc_2

brown	X	X
dog	X	X
fox	X	X
in		X
jump	X	X
lazy	X	X
over	X	X
quick	X	X
summer		X
the	X	X

Analysis

You can find only term that exists in your inverted index

So, both *indexed text* and the *query string*
must be normalized into the same form

The process of tokenization and normalization is called:

Analysis

Analysis is a process consists of:

- First, **tokenizing** a block of text into individual terms
- Then **normalizing** these terms into a standard form (to improve their “searchability”)

Analyzer is a wrapper of:

- Character filters (tidy up the text before tokenization)
- Tokenizer (e.g. splitting the text when whitespace is encountered)
- Token filters
 - change terms by e.g. lowercasing,
 - removing terms by e.g. stopwords such *a, and, the*
 - adding terms by e.g. synonyms like *jump* and *leap*

Test String:

"Set the shape to semi-transparent by calling set_trans(5)"

Standard Analyzer (default analyzer)

set, the, shape, to, semi, transparent, by, calling, set_trans, 5

Simple Analyzer

set, the, shape, to, semi, transparent, by, calling, set, trans

Whitespace Analyzer

Set, the, shape, to, semi-transparent, by, calling, set_trans(5)

Language Analyzers (e.g. English analyzer)

set, shape, semi, transpar, call, set_tran, 5

When Analyzers are Used?

Analizers are used for:

- **Indexing** - full-text fields are analyzed into terms that are used to create the inverted index
- **Query** - when we search on a full-text field, we need to pass the query string through the same analysis process

Full-text queries understand how each field is defined, so they can:

- When you query a *full-text* field, the query will apply the same analyzer to the query string to produce the correct list of terms to search for
- When you query an *exact-value* field, the query will not analyze the query string, but instead search for the exact value that you have specified

Testing Analyzers

```
GET /_analyze
{
  "analyzer": "standard",
  "text": "Text to analyze"
}
```

token - term stored in index

position - term order in text

start_offset, end_offset -
character position of the word in
original text

```
{ "tokens": [ {
  "token": "text",
  "start_offset": 0,
  "end_offset": 4,
  "type": "<ALPHANUM>",
  "position": 1
}, {
  "token": "to",
  "start_offset": 5,
  "end_offset": 7,
  "type": "<ALPHANUM>",
  "position": 2
}, {
  "token": "analyze",
  "start_offset": 8,
  "end_offset": 15,
  "type": "<ALPHANUM>",
  "position": 3
}
] }
```

DEMO

Analizers

Analyzing text with
different analyzers

Defining custom analyzers



Mapping

When Elasticsearch detects a new string field in your documents, it automatically configures it as a full-text `text` field and analyzes it with the `standard analyzer`

You don't always want this

You can configure these fields manually by specifying the **mapping**

Mapping

Elasticsearch needs to know what type of data each field contains

This information is contained in the *mapping*

Each document in an index has a *type*

Every *type* has its own **mapping**, or **schema definition**

Mapping defines:

- fields within a type
- how the field should be handled by Elasticsearch
- metadata associated with a type



IMPORTANT

Indices created in Elasticsearch 6.0.0 or later may only contain a single mapping type

Mapping types will be completely removed in Elasticsearch 7.0.0

Dynamic Mapping

When you index a document that contains a new field—one previously not seen—Elasticsearch will use **dynamic mapping** to try to guess the field type from the basic data types available in JSON

JSON Type	Field Type
Boolean: <code>true</code> or <code>false</code>	<code>boolean</code>
Whole number: <code>123</code>	<code>long</code>
Floating point: <code>123.45</code>	<code>double</code>
String, valid date: <code>2014-09-15</code>	<code>date</code>
String: <code>foo bar</code>	<code>text</code>

Customizing Field Mappings

Basic field datatypes are sufficient for many cases

Yet you will often need to customize the mapping for individual fields, especially text fields, in order to:

- Distinguish between full-text string fields and exact value string fields
- Use language-specific analyzers
- Optimize a field for partial matching
- Specify custom date formats
- And much more

```
{  
  "number_of_clicks": {  
    "type": "integer"  
  }  
}
```


Field Datatypes

Each field has a data type which can be:

- a simple type like `text`, `keyword`, `date`, `long`, `double`, `boolean` or `ip`.
- a type which supports the hierarchical nature of JSON such as `object` or `nested`.
- or a specialised type like `geo_point`, `geo_shape`, or `completion`.

Often is useful to index the same field in different ways for different purposes

E.g. string field can be indexed as `text` for full-text search and as a `keyword` field for sorting or aggregations.

This is the purpose of *multi-fields*

Example Mapping

```
PUT my_index
{
  "mappings": {
    "doc": {
      "properties": {
        "title":    { "type": "text" },
        "name":     { "type": "text" },
        "age":      { "type": "integer" },
        "created":  {
          "type": "date",
          "format": "strict_date_optional_time||epoch_millis"
        }
      }
    }
  }
}
```



Existing field mappings cannot be updated.

You should create a new index with correct mapping and reindex your data into that index.

DEMO

Mapping

Viewing the type mapping

Testing the mapping

Multi-field



Empty Fields

There is no way of storing a `null` value in Lucene

These three fields would all be considered to be empty, and would not be indexed

```
"null_value":          null,  
"empty_array":         [],  
"array_with_null_value": [ null ]
```

Inner Objects

Inner objects are often used to embed one entity or object inside another. For instance, instead of having fields called `user_name` and `user_id` inside our `tweet` document, we could write:

```
{
  "tweet": "Elasticsearch is very flexible",
  "user": {
    "id": "@johnsmith",
    "gender": "male",
    "age": 26,
    "name": {
      "full": "John Smith",
      "first": "John",
      "last": "Smith"
    }
  }
}
```

Inner Objects

```
PUT my_index/my_type/1
{
  "region": "US",
  "manager": {
    "age": 30,
    "name": {
      "first": "John",
      "last": "Smith"
    }
  }
}
```

```
{
  "region": "US",
  "manager.age": 30,
  "manager.name.first": "John",
  "manager.name.last": "Smith"
}
```

Inner Objects

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "properties": {
        "region": {
          "type": "keyword"
        },
        "manager": {
          "properties": {
            "age": { "type": "integer" },
            "name": {
              "properties": {
                "first": { "type": "text" },
                "last": { "type": "text" }
              }
            }
          }
        }
      }
    }
  }
}
```

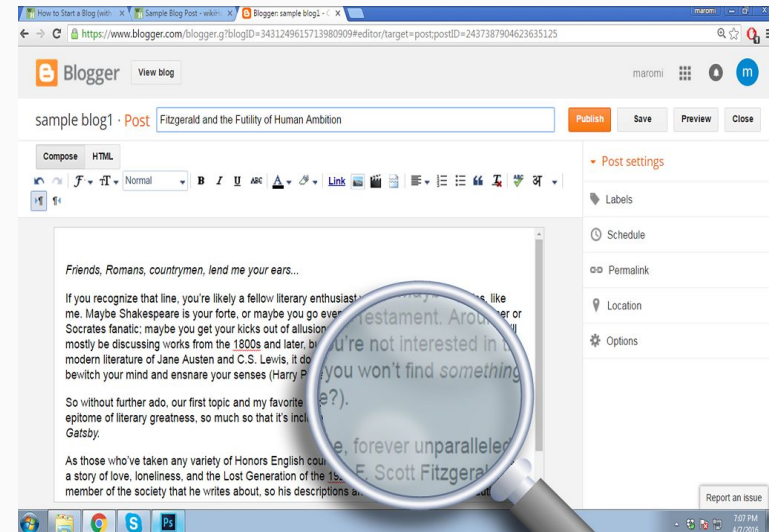
Assignment 3 - Analyzers and Mappings

Add search capabilities to a blog website

Use appropriate datatypes for the index fields

Use appropriate Analyzers for the fields

Run defined queries to validate mapping definitions



Elasticsearch Workshop

Afternoon



Query DSL

Full-Body Search

Search *lite*—a query-string search—is useful for ad hoc queries

To harness the full power of search, use the request body *search* API

Parameters are passed in the HTTP request body instead of in the query string

Request body search handles:

- the query itself, but also...
- return highlighted snippets from your results
- aggregate analytics across all results or subsets of results
- return did-you-mean suggestions, etc.

Empty Search

```
GET /_search
{ }
```

```
GET blog/_search
{
  "from": 30,
  "size": 10
}
```

```
POST blog/_search
{
  "from": 30,
  "size": 10
}
```

Note: The HTTP libraries of certain languages don't allow `GET` requests to have a request body. The truth is that RFC 7231 does not define what should happen to a `GET` request with a body.

Instead of the cryptic query-string approach, request body search allows writing queries by using the **query domain-specific language** - query DSL

Query DSL

- The query DSL is a flexible
- expressive search language
- expose most of the power of Lucene through a simple JSON interface.
- you should be using it to write your queries in production
- makes your queries more flexible, more precise, easier to read, and easier to debug.

```
GET /_search
{
  "query": YOUR_QUERY_HERE
}
```

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

Structure of a Query Clause

```
{  
  QUERY_NAME: {  
    ARGUMENT: VALUE,  
    ARGUMENT: VALUE, ...  
  }  
}
```

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

```
{  
  QUERY_NAME: {  
    FIELD_NAME: {  
      ARGUMENT: VALUE,  
      ARGUMENT: VALUE, ...  
    }  
  }  
}
```

```
GET /_search  
{  
  "query": {  
    "match": {  
      "tweet": "elasticsearch"  
    }  
  }  
}
```

Combining Multiple Clauses

Query clauses are simple building blocks to create complex queries:

- *Leaf* clauses (like the `match` clause)
- *Compound* clauses (combine other query clauses, like `bool`, `must` etc. or other compound clauses)

```
{
  "bool": {
    "must":      { "match": { "tweet": "elasticsearch" } },
    "must_not":  { "match": { "name":  "mary" } },
    "should":    { "match": { "tweet": "full text" } },
    "filter":    { "range": { "age" : { "gt" : 30 } } }
  }
}
```

Combining Multiple Clauses

The following query looks for emails that contain `business opportunity` and should either be starred, or be both in the Inbox and not marked as spam:

```
{
  "bool": {
    "must": { "match": { "email": "business opportunity" }},
    "should": [
      { "match": { "starred": true }},
      { "bool": {
        "must": { "match": { "folder": "inbox" }},
        "must_not": { "match": { "spam": true }}
      }}
    ],
    "minimum_should_match": 1
  }
}
```


Most Important Queries

match_all Query: { "match_all": {} }

match Query: { "match": { "tweet": "About Search" } }

{ "match": { "age": 26 } }

{ "match": { "date": "2014-09-01" } }

{ "match": { "public": true } }

{ "match": { "tag": "full_text" } }

multi_match Query: {

"multi_match": {

"query": "full text search",

"fields": ["title", "body"]

}

}

Most Important Queries

range Query:

```
{ "range": {  
  "age": {  
    "gte": 20,  
    "lt": 30  
  }  
}
```

gt Greater than
gte Greater than or equal to
lt Less than
lte Less than or equal to

term Query:

```
{ "term": { "age": 26 } }  
{ "term": { "date": "2014-09-01" } }  
{ "term": { "public": true } }  
{ "term": { "tag": "full_text" } }
```

terms Query:

```
{ "terms": { "tag": [ "search", "full_text", "nosql" ] } }
```

exists and missing Queries:

```
{ "exists": { "field": "title" } }  
{ "missing": { "field": "title" } }
```

Combining Queries Together - bool query

bool query combines multiple queries together in user-defined boolean combinations

```
{
  "bool": {
    "must":      { "match": { "title": "how to make millions" }},
    "must_not": { "match": { "tag":    "spam" }},
    "should": [
      { "match": { "tag": "starred" }},
      { "range": { "date": { "gte": "2014-01-01" }}}
    ]
  }
}
```

The relevance score is calculated for each sub-query and then merged together

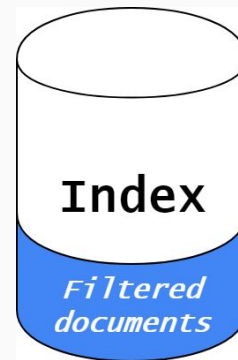


If there are no `must` clauses, at least one `should` clause has to match. However, if there is at least one `must` clause, no `should` clauses are required to match.

Filtering Query

If we want that some sub-query does not affect total relevance score - **filtering query**

```
{
  "bool": {
    "must":      { "match": { "title": "how to make millions" }},
    "must_not":  { "match": { "tag":   "spam" }},
    "should": [
      { "match": { "tag": "starred" }}
    ],
    "filter": {
      "range": { "date": { "gte": "2014-01-01" }}
    }
  }
}
```



Another benefit is increased performance by not calculating score and by caching

Use query clauses for *full-text* search or for any condition that should affect the *relevance* score, and use filters for everything else

query

relevance

full text

not cached

slower

filter

boolean yes/no

exact values

cached

faster

Filter first, then query remaining docs

Validating Queries

```
GET /gb/_validate/query
```

```
{
  "query": {
    "tweet": {
      "match": "really powerful"
    }
  }
}
```

```
GET /_validate/query?explain
```

```
{
  "query": {
    "match": {
      "tweet": "really powerful"
    }
  }
}
```

```
{
  "valid" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "failed" : 0
  }
}
```

```
{
  "valid" : true,
  "_shards" : { ... },
  "explanations" : [ {
    "index" : "us",
    "valid" : true,
    "explanation" : "tweet:really tweet:powerful"
  }, {
    "index" : "gb",
    "valid" : true,
    "explanation" : "tweet:realli tweet:power"
  } ]
}
```

Explain why a document (doesn't) match the given query

```
GET /us/_doc/1/_explain
{
  "query" : {
    "bool" : {
      "filter" : { "term" : { "user_id" : 2 } },
      "must" : { "match" : { "tweet" : "honeymoon" } }
    }
  }
}
```

```
"failure to match filter: cache(user_id:[2 TO 2])"
```

DEMO

Query DSL

Trying out various queries
using Query DSL



Assignment 4 - Query Data using Query DSL

Practice Elasticsearch Query DSL by Searching tweet messages.



Sorting & Relevance

Sorting

Default sorting in Elasticsearch (Lucene) is `_score` descending - the most relevant docs first

But you can also sort by any field value, use different sort options, use a script to calculate sort value, sort by geo coordinates etc.



IMPORTANT

Memory Considerations

When sorting, the relevant sorted field values are loaded into memory. This means that per shard, there should be enough memory to contain them. For string based types, the field sorted on should not be analyzed / tokenized. For numeric types, if possible, set the type to narrower types.

Sorting Example

```
GET /my_index/_search
```

```
{  
  "sort" : [  
    { "post_date" : { "order" : "asc" } },  
    "user",  
    { "name" : "desc" },  
    { "age" : "desc" },  
    "_score"  
  ],  
  "query" : {  
    "term" : { "user" : "kimchy" }  
  }  
}
```

asc Sort in ascending order

desc Sort in descending order

Sorting Multi-Valued Fields (Sort Mode)

```
PUT /my_index/_doc/1?refresh
```

```
{  
  "product": "chocolate",  
  "price": [20, 4]  
}
```

```
POST /_search
```

```
{  
  "query" : {  
    "term" : { "product" : "chocolate" }  
  },  
  "sort" : [  
    { "price" : { "order" : "asc", "mode" : "avg" } }  
  ]  
}
```

min	Pick the lowest value.
max	Pick the highest value.
sum	Use the sum of all values as sort value (num only).
avg	Use the average of all values as sort value (num only).
median	Use the median of all values as sort value (num only).

Script Based Sorting

GET /_search

<https://qbox.io/blog/elasticsearch-scripting-sorting>

```
{
  "query" : {
    "term" : { "user" : "kimchy" }
  },
  "sort" : {
    "_script" : {
      "type" : "number",
      "script" : {
        "lang": "painless",
        "source": "doc['field_name'].value * params.factor",
        "params" : {
          "factor" : 1.1
        }
      }
    },
    "order" : "asc"
  }
}
```

Document Relevance

In order to sort by *relevance*, we need to represent relevance as a *value*

The higher the value - the more the document is relevant to the search

`_score` - *relevance score* is represented by the floating-point number

Default sorting in Elasticsearch (Lucene) is `_score` descending

Calculating Document Relevance

How common is the term in this doc (term frequency)

-> more is better

How common is the term in ALL docs (Inverse document frequency)

-> less is better

How long is the doc? (Length norm)

-> shorter is better

Calculating Relevance in Bool Query

```
{  
  "bool": {  
    "must":      { "match": { "title": "how to make millions" }},  
    "must_not": { "match": { "tag":   "spam" }},  
    "should": [  
      { "match": { "tag": "starred" }},  
      { "range": { "date": { "gte": "2014-01-01" }}}  
    ]  
  }  
}
```

$\text{_score of bool query} = \frac{\text{sum (_score of each query)} \times \text{num of matching queries}}{\text{num of queries}}$

more matching “should” queries == better relevance score

Boosting

We can control the relative weight of a query clause by specifying a **boost** value

A boost value greater than 1 increases the relative weight of that clause

Note that the increase or decrease is not linear

(`boost` of 2 does not double the `_score`)

The new `_score` is *normalized* after the boost is applied (complex algorithm)

Suffice to say that a higher `boost` value results in a higher `_score`

Boosting

```
GET /_search
{
  "query": {
    "bool": {
      "should": [
        {
          "match": {
            "title": {
              "query": "quick brown fox",
              "boost": 2
            }
          }
        },
        {
          "match": {
            "content": "quick brown fox"
          }
        }
      ]
    }
  }
}
```

title query clause is twice as important as the content query clause, since it's boosted by factor 2

A query clause without a *boost* value has a neutral boost factor of 1

Query-time boosting is the main tool that you can use to tune relevance

Practically, there is no simple formula for deciding on the “correct” boost value for a particular query clause. It's a matter of try-it-and-see

Sometimes we just don't care about relevance the documents have for a query

We just want to know is that a certain word appears in a field

```
GET vacation_homes/_search
{
  "query": {
    "bool": {
      "should": [
        { "constant_score": {
          "query": { "match": { "description": "wifi" }}
        }},
        { "constant_score": {
          "query": { "match": { "description": "garden" }}
        }},
        { "constant_score": {
          "query": { "match": { "description": "pool" }}
        }}
      ]
    }
  }
}
```

Relevance Tuning is the Last 10%

Understanding the score-generation process is critical to manipulate the score for your particular business domain

In practice, simple combinations of queries will get you good search results

To get *great* search results you'll need to play with different tuning methods

The best approach is to put instrumentation code in place and monitor what happens when you change some parameter

DEMO

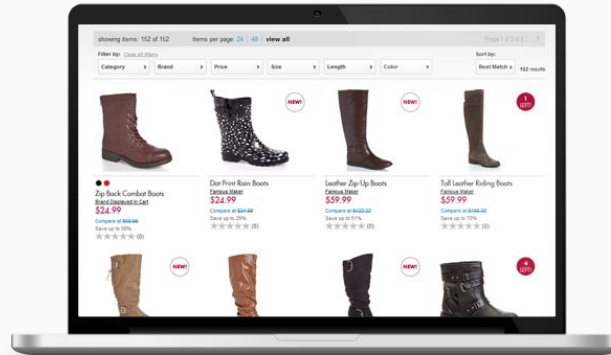
Sorting & Relevance

Checking how relevance is
calculated in different
queries



Assignment 5 - Tweaking Relevance

As an e-commerce shop manager, tweak search queries to promote products which are on sale or are high in inventory



Break



15 minutes

Dealing with Human Languages

Dealing with Human Languages

Full-text search is a battle between:

- *precision* — returning as few irrelevant documents as possible and
- *recall* — returning as many relevant documents as possible

By matching only the exact words user has queried, we would miss out many document the user would consider to be relevant

E.g. search for “quick brown fox” should match a document containing “fast brown foxes”, “Johnny Walker” to match “Johnnie Walker” etc.

Dealing with Human Languages

If documents exist that *do* contain exactly what the user has queried, those documents should appear at the top of the result set

... but weaker matches can be included further down the list.

If no documents match exactly, at least we can show the user potential matches; they may even be what the user originally intended!

There are several lines of attack to this problem

Dealing with Human Languages

- **Normalizing tokens**

Remove diacritics like ´, ^, and ¨

- **Reducing words to their root form**

Remove the distinction between singular and plural or between tenses

- **Stopwords**

Remove commonly used words like `the`, `and`, or `to` to improve search performance

- **Synonyms**

Including synonyms so that a query for `quick` could also match `fast`

- **Typos and misspellings**

Check for misspellings or alternate spellings, or match on homophones

Language Analyzers

Elasticsearch ships with a collection of language analyzers

Out-of-the-box support for many of the world's most common languages

Arabic, Armenian, Basque, Brazilian, Bulgarian, Catalan, Chinese, Czech, Danish, Dutch, English, Finnish, French, Galician, German, Greek, Hindi, Hungarian, Indonesian, Irish, Italian, Japanese, Korean, Kurdish, Norwegian, Persian, Portuguese, Romanian, Russian, Spanish, Swedish, Turkish, and Thai

You can also customize existing or create your own language analyzers

These analyzers typically perform four roles:

- Tokenize text into individual words

The quick brown foxes → [The, quick, brown, foxes]

- Lowercase tokens

The → the

- Remove common *stopwords*

[The, quick, brown, foxes] → [quick, brown, foxes]

- Stem tokens to their root form

foxes → fox

Additional transformations may be executed, too:

- The `english` analyzer removes the possessive `'s`
`John's` → `john`
- The `french` analyzer removes elisions `l'` and `qu'` and diacritics like `é`
`l'église` → `eglis`
- The `german` analyzer normalizes terms, replacing `ä` and `ae` with `a`, or `ß` with `ss`, among others
`äußerst` → `ausserst`

Multi-language Case

The goal, although not always possible, should be to keep languages separate
Mixing languages in the same inverted index can be problematic

Approaches are (at indexing time):

- One predominant language per *document*
- One predominant language per *field*
- A mixture of languages per field

At query time:

Try detecting user language and search and return results in that language only

Aggregations

Aggregations

With aggregations, we zoom out to get an overview of our data.

Instead of looking for individual documents, we want to analyze and summarize our complete set of data

While the functionality is completely different from search, it leverages the same data-structures

Aggregations execute quickly and are *near real-time*, just like search

This is extremely powerful for reporting and dashboards

Aggregations are so powerful that many companies have built large Elasticsearch clusters solely for analytics

High Level Concepts

Like the query DSL, aggregations have a composable syntax

There are only a few basic concepts to learn, but nearly limitless combinations of those basic components

Two main concepts are:

- **Buckets** - Collections of documents that meet a criterion
- **Metrics** - Statistics calculated on the documents in a bucket

Every **aggregation** is simply a combination of one or more buckets and zero or more metrics

Buckets are conceptually similar to grouping in SQL, while metrics are similar to `COUNT()`, `SUM()`, `MAX()`, and so forth

Buckets

Because buckets can be nested, we can derive a much more complex aggregation:

1. Partition documents by country (bucket).
2. Then partition each country bucket by gender (bucket).
3. Then partition each gender bucket by age ranges (bucket).
4. Finally, calculate the average salary for each age range (metric)

This will give you the average salary per `<country, gender, age>` combination.

All in one request and with one pass over the data!

DEMO

Aggregations

Applying various metrics to
buckets (grouped data)



Assignment 6 - Aggregating Data

Bank Accounts Balance Statistic Reporting

Help Bank management to sell more credit loans



Geo-spatial Searching

Geolocation

Gone are the days when we wander around a city with paper map

I'm not interested in restaurants in London—I want to know about restaurants *within a 5-minute walk of my current location.*

But geolocation is only one part of the puzzle

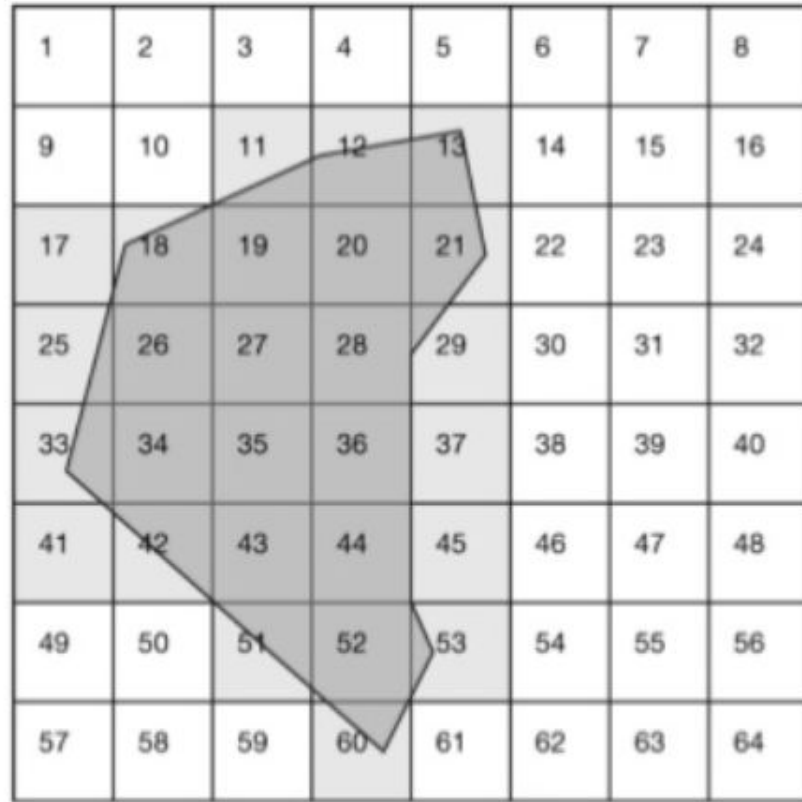
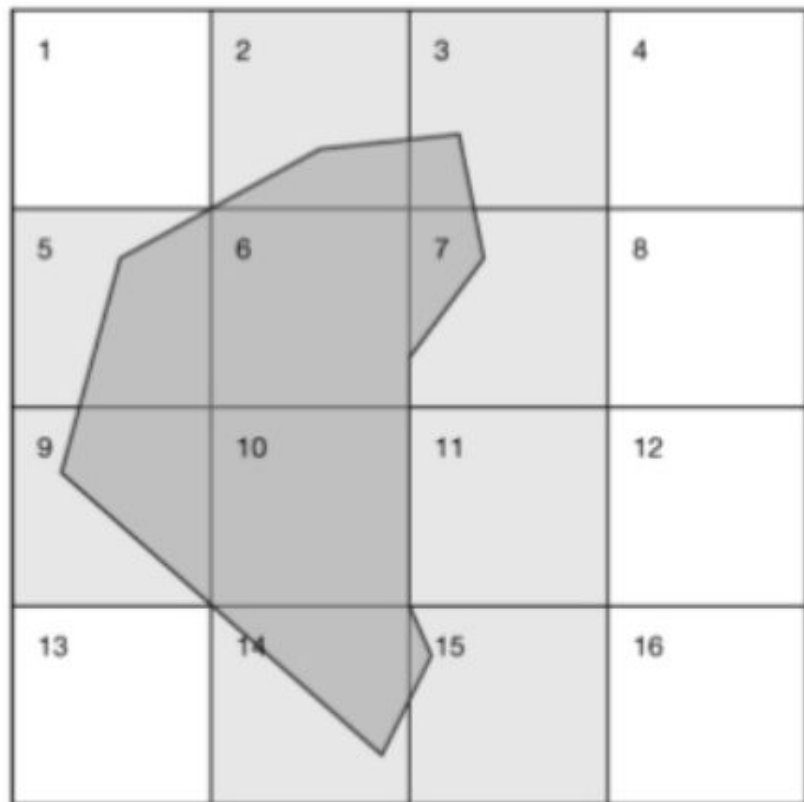
The beauty of Elasticsearch is that it allows you to combine geolocation with full-text search, structured search, and analytics

Geolocation

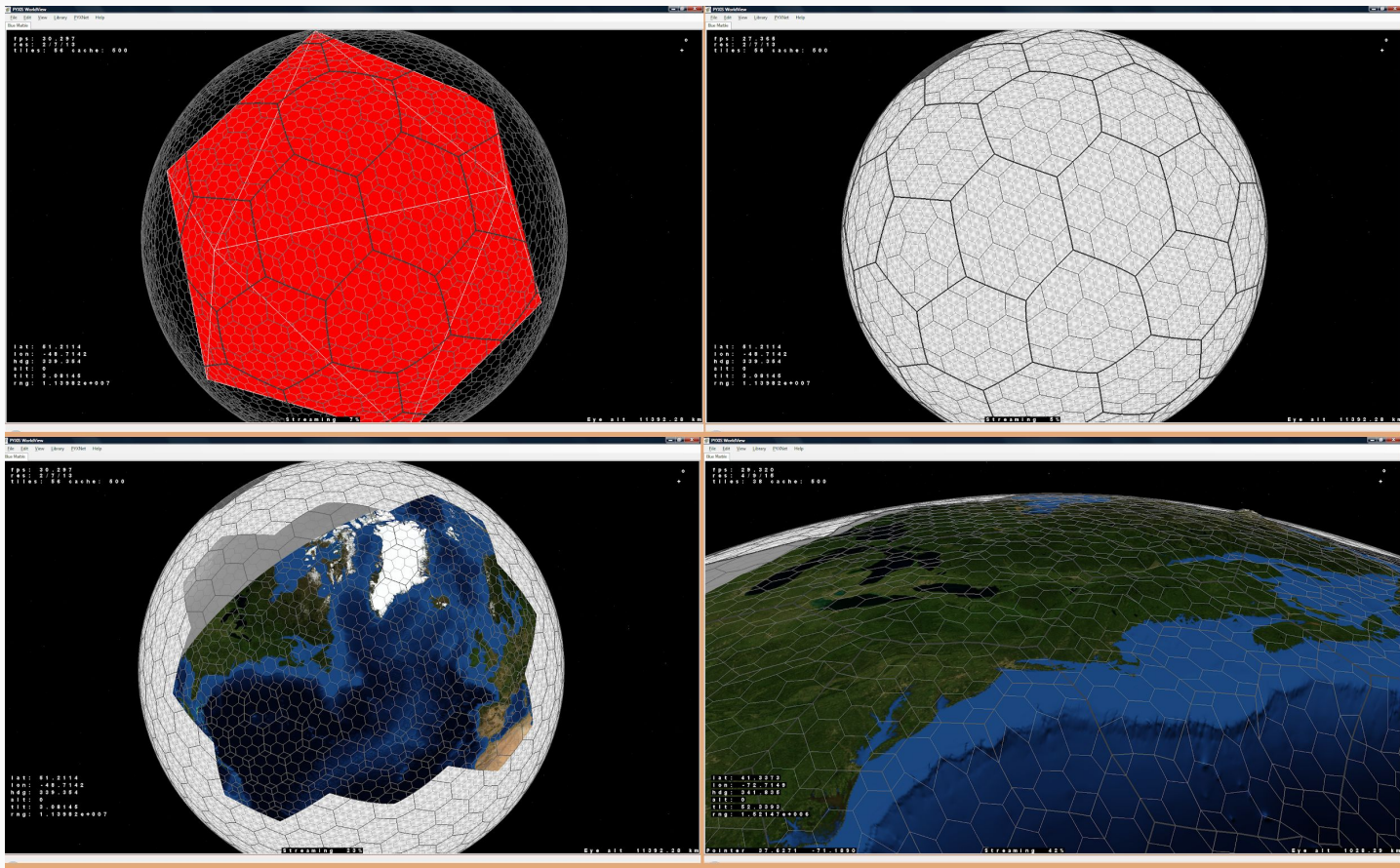
Examples:

- show me restaurants that mention vitello tonnato, are within a 5-minute walk, and are open at 11 p.m., and then rank them by a combination of user rating, distance, and price
- show me a map of vacation rental properties available in August throughout the city, and calculate the average price per zone

The Grid Structure of a Spatial Index



The Grid Structure of a Spatial Index



City	Latitude, Longitude	Geohash
Belgrade	44.7866, 20.4489	srywc35hz
Nis	43.3209, 21.8958	srxw6q249
Skopje	41.9973, 21.4280	srrnzt7kh
Minsk	53.9045, 27.5615	u9edejj8m
Brest	52.0976, 23.7341	u92bkh4ds

Geohashing

geohash length	lat bits	lng bits	lat error	lng error	km error
1	2	3	± 23	± 23	± 2500
2	5	5	± 2.8	± 5.6	± 630
3	7	8	± 0.70	± 0.70	± 78
4	10	10	± 0.087	± 0.18	± 20
5	12	13	± 0.022	± 0.022	± 2.4
6	15	15	± 0.0027	± 0.0055	± 0.61
7	17	18	± 0.00068	± 0.00068	± 0.076
8	20	20	± 0.000085	± 0.00017	± 0.019

City	Latitude, Longitude	Geohash
Belgrade	44.7866, 20.4489	srywc35hz
Nis	43.3209, 21.8958	srxw6q249
Skopje	41.9973, 21.4280	srrnzt7kh
Minsk	53.9045, 27.5615	u9edejj8m
Brest	52.0976, 23.7341	u92bkh4ds

<https://en.wikipedia.org/wiki/Geohash>

Representing Geolocations in Elasticsearch

`geo_points`

latitude-longitude points

`geo_shape`

complex shapes defined in GeoJSON

Geo Points

```
PUT /attractions
{
  "mappings": {
    "properties": {
      "name": {
        "type": "text"
      },
      "location": {
        "type": "geo_point"
      }
    }
  }
}
```

```
PUT /attractions/_doc/1
{
  "name":      "Chipotle Mexican Grill",
  "location": "40.715, -74.011"
}
```

```
PUT /attractions/_doc/2
{
  "name":      "Pala Pizza",
  "location": {
    "lat":      40.722,
    "lon":      -73.989
  }
}
```

```
PUT /attractions/_doc/3
{
  "name":      "Mini Munchies Pizza",
  "location": [ -73.983, 40.719 ]
}
```

Filtering by Geo Point

```
GET /attractions/_search
{
  "query": {
    "filtered": {
      "filter": {
        "geo_bounding_box": {
          "location": {
            "top_left": {
              "lat": 40.8,
              "lon": -74.0
            },
            "bottom_right": {
              "lat": 40.7,
              "lon": -73.0
            }
          }
        }
      }
    }
  }
}
```

```
GET /attractions/_search
{
  "query": {
    "filtered": {
      "filter": {
        "geo_distance": {
          "distance": "1km",
          "location": {
            "lat": 40.715,
            "lon": -73.988
          }
        }
      }
    }
  }
}
```


Sorting by Distance

```
GET /attractions/_search
```

```
{
  "query": {
    "filtered": {
      "filter": {
        "geo_bounding_box": {
          "type": "indexed",
          "location": {
            "top_left": {
              "lat": 40.8,
              "lon": -74.0
            },
            "bottom_right": {
              "lat": 40.4,
              "lon": -73.0
            }
          }
        }
      }
    }
  }
}
```

```
...
},
"sort": [
  {
    "_geo_distance": {
      "location": {
        "lat": 40.715,
        "lon": -73.998
      },
      "order": "asc",
      "unit": "km",
      "distance_type": "plane"
    }
  }
]
```

Geo Aggregations

A search may return too many results to be able to display each geo-point

Geo-aggregations can be used to cluster geo-points into more manageable buckets

Three aggregations work with fields of type `geo_point`:

- `geo_distance` - Groups documents into concentric circles around a central point.
- `geohash_grid` - Groups documents by geohash cell, for display on a map.
- `geo_bounds` - Returns the lat/lon coordinates of a bounding box that would encompass all of the geo-points. This is useful for choosing the correct zoom level when displaying a map.

Elasticsearch Java Clients

Elasticsearch Clients

- Elasticsearch Java Client
- JavaScript API
- Groovy API
- .NET API
- PHP API
- Perl API
- Python API
- Ruby API
- Community Contributed Clients

Spring Data Elasticsearch

Spring Data project provides:

- Abstractions for different data stores
- Helps avoid boilerplate code
- Speciality of each store available
- Dynamic Repository implementations
 - CRUD operations for the corresponding document class will be made available by default
- Popular modules
 - Spring Data JPA
 - Spring Data MongoDB

Repository

```
public interface DishRepository  
    extends ElasticsearchCrudRepository<Dish, String> { }
```

```
public interface DishRepository  
    extends ElasticsearchCrudRepository<Dish, String> {  
  
    List<Dish> findByFood(String food);  
  
    List<Dish> findByTagsAndFavoriteLocation(String tag, String location);  
  
    List<Dish> findByFavoritePriceLessThan(Double price);  
  
    @Query("{\"query\": {\"match_all\": {}}}")  
    List<Dish> customFindAll();  
}
```

Configuration

```
@Configuration
@EnableElasticsearchRepositories(basePackages = "com.company.spring.data.es.repository")
@ComponentScan(basePackages = {"com.company.spring.data.es.service"})
public class Config {
    @Bean
    public NodeBuilder nodeBuilder() {
        return new NodeBuilder();
    }
    @Bean
    public ElasticsearchOperations elasticsearchTemplate() {
        Settings.Builder elasticsearchSettings =
            Settings.settingsBuilder()
                .put("http.enabled", "false")
                .put("path.data", tmpDir.toAbsolutePath().toString())
                .put("path.home", "PATH_TO_YOUR_ELASTICSEARCH_DIRECTORY");
        logger.debug(tmpDir.toAbsolutePath().toString());
        return new ElasticsearchTemplate(nodeBuilder()
            .local(true)
            .settings(elasticsearchSettings.build())
            .node()
            .client());
    }
}
```

Entity

```
@Document(indexName = "blog", type = "article")
public class Article {

    @Id
    private String id;

    private String title;

    @Field(type = FieldType.Nested)
    private List<Author> authors;

    // standard getters and setters
}
```


Indexing Documents

```
// creating an index
```

```
elasticsearchTemplate.createIndex(Article.class) ;
```

```
// indexing a document
```

```
Article article = new Article("Spring Data Elasticsearch") ;
```

```
article.setAuthors(asList(new Author("John Smith"), new Author("John  
Doe")));
```

```
articleService.save(article) ;
```

Querying

```
// method name-based query
String nameToFind = "John Smith" ;
Page<Article> articleByAuthorName
    = articleService.findByAuthorName(nameToFind, new PageRequest(0, 10)) ;

// a custom query
SearchQuery searchQuery = new NativeSearchQueryBuilder()
    .withFilter(regexQuery("title", ".*data.*"))
    .build() ;
List<Article> articles = elasticsearchTemplate.queryForList(searchQuery,
Article.class) ;
```

Updating and Deleting

```
String articleTitle = "Spring Data Elasticsearch";
SearchQuery searchQuery = new NativeSearchQueryBuilder()
    .withQuery(matchQuery("title", articleTitle).minimumShouldMatch("75%"))
    .build();
List<Article> articles = elasticsearchTemplate
    .queryForList(searchQuery, Article.class);

// now update the article
article.setTitle("Getting started with Search Engines");
articleService.save(article);

// delete a document
articleService.delete(articles.get(0));
```

Thank you!