



Copyright 2018 Trainologic LTD

ElasticSearch

- Started as a project named Compass by Shay Banon.
- Grew to become a multi-tenant and distributed search engine.
- Several other tools were built around it for the integrated ELK stack.
- Let's go down memory lane...

ElasticSearch

- ElasticSearch (ES) is a NoSQL storage engine (DB).
- Its main benefits are full-text-search and geo-indexing.
- One of the common use cases for using ES is log aggregation, however you can also use it in many other scenarios.

Not for Commercial Use

ELK

- Because Logs analysis and aggregation became such an important use-case for ES, several other tools were developed to help with this popular use-case:
 - Kibana – a feature-rich front end for log analysis.
 - Logstash – log collector and pipe-line framework.
 - Beats – small and efficient log collectors.

Basic Concepts

- Document – the object being indexed by ES. A document's format is always JSON.
- Index – A collection of documents. Its name must be all lowercase and unique.
- Type – Many changes here and deprecations (we'll discuss it shortly).
- Node – a member of the cluster.
- Cluster – a collection of nodes composing the ElasticSearch database.

Type Deprecation

- Each document is assigned a single type (`_type` meta field).
- Combined with `_id` to generate the `_uid` field.
- In versions prior to 6.x, you could have specified multiple types in the same index (and search with multiple types).
- An analogy was made between index/types to database and tables.
- However, in the same index, same field name must have the same type.

Type Alternatives

- You can use an `index` per type.
- You could alternately use custom fields.
- In ES 6.x, you can have a single type associated with an index.
- The `_uid` is now alias for `_id`.

Indexing

- ElasticSearch is built on top of the awesome Lucene library.
- Provides the foundation for most indexing/search tools and frameworks in the Java ecosystem.
- At the heart of Lucene there is the **inverted index**.
- Let's understand it...

Inverted Index

- Let's say that we have the following documents and their contents:
- Document 1:
 - Hello, is anyone out there?
- Document 2:
 - There is an exit. Outside.
- Document 3:
 - One with the machine.

Inverted Index

- A regular index is a document-to-term mapping.
- An inverted index maps the terms to their respective documents:
- E.g.:
 - hello – 1.
 - is – 1, 2.
 - one – 3.
 - anyone – 1.

Not for Commercial Use

Mapping Data

- Before loading data into ES, we must create the mapping.
- This is because according to the mapping, text is processed.
- Not only on insert, but also on searches...

String Types

- ES provides two types for indexing *string* fields:
 - text.
 - keyword.
- The *keyword* type provides for exact value search, whereas *text* types are fully analyzed.
- Use *keyword* for filtering, sorting and aggregations.
- Consider emails, hostnames, etc..

Numeric Types

- Supported: *long, integer, short, byte, double, float, half_float, scaled_float.*
- All backed-by-floating-point types must have finite values and differentiate -0.0 from +0.0.
- All are signed.
- Java-based sizes.
- Scaled-float is more efficient if applicable.

Boolean Types

- Starting from version 6.x, accepts JSON *true/false*, or string-values: “true”/“false”.
- Before that, ‘off’, ‘no’, ‘0’ and “ ” were also accepted as false values.

Not for Commercial Use

Date Type

- Dates are stored as *long* epoch representation of UTC value.
- The field itself can be:
 - String formatted date/time.
 - Long value of milliseconds since epoch.
 - Int value of seconds since epoch.



Binary Type

- Accepts a Base64 value.
- Not stored or searchable by default.

Not for Commercial Use

Range Types

- A range type allows for specifying a range value in documents.
- Can be:
 - Numeric: integer_range, float_range, long_range, double_range,
 - Date/time: date_range.
 - IP (CIDR): ip_range.

The _source Field

- Enabled by default.
- Holds the original JSON document.
- Remember that even if this feature is enabled, the field is not indexed and can't be used for search.
- Incurs storage overhead.
- However, be careful before disabling this feature.
- Let's talk about it...

The `_source` Field

- When you disable `_source`, you lose the following features:
 - Reindexing into a new index.
 - Reindex, update and `update_by_query` APIs.
 - On-the-fly highlighting.
 - Some debugging ability.
- Regarding disk storage, consider higher compression over disabling `_source`.

The `_source` Field

- One of the classical use-cases for disabling `_source` is metrics collection.
- No need for highlighting, no need for updates or re-indexing (too fast data).
- Note that you should also probably disable the `_all` feature also.

The _all Field

- Deprecated in 6.x.
- A special field that is the concatenation of all the fields in the document.
- Space is the delimiter.
- Note that the field is analyzed and indexed, but not stored!

Kibana

- Kibana is the analytics and visualization tool in the ELK stack.
- Supports a wide variety of charts, dashboards and maps.
- Let's start with the basics...

Not for Commercial Use

Import Some Data

- Before loading some data into ES, we need to define the mapping.
- Let's see it for the example 'logs' dataset:

```
PUT /logstash-2015.05.18
{
  "mappings": {
    "log": {
      "properties": {
        "geo": {
          "properties": {
            "coordinates": {
              "type": "geo_point"
            }
          }
        }
      }
    }
  }
}
```

Task

- Explore a few lines from the logs.jsonl file.
- Execute “*grep "logstash-" * | sort -u*”
- Define the additional mappings according to the index names.

Not for Commercial Use



Bulk Insert

- For now, we won't go into the actual index/bulk-index APIs.
- Let's execute a bulk insert from the command line:

```
curl -H 'Content-Type: application/x-ndjson' -XPOST  
'localhost:9200/_bulk?pretty' --data-binary @logs.jsonl
```



Task

- Examine the results from the Kibana console using:
- GET _cat/indices?v

Not for Commercial Use

cat API

- ElasticSearch provides comprehensive JSON-based REST API.
- However, JSON is a format suited for machines.
- You can use the **cat** API to get tabular human-readable representation.
- Let's go over the results of the previous command...

Traffic Lights

- So, we see that the status is yellow for every index.
- In order to understand why, execute:
GET _cat/shards
- Now add the “?v” to the end of the URL.
- Can you figure the problem?

Traffic Lights

- Red means that at least one primary shard was not assigned.
- Yellow means that for at least one shard, there are no replicas.
- Green means, well you know ☺

Not for Commercial Use

Task

- For the ‘cat shards’ API, specify an index name before the ‘?’ character (to filter shards only for the specified index).
- Use a wildcard to display shards only for our *logs* data.
- Using the query parameter **h**, you can specify the columns to display as a comma-separated list.
- Please display: index, shard, state and unassigned.reason.
- Using the query parameter **s**, you can sort according to comma separated fields (you can specify ‘:asc’ or ‘:desc’ for every one).
- Please sort according to *index* and *shard*.

Updating Index Setting

- Let's update an index to specify 'no-replicas':

```
PUT /logstash-2015.05.18/_settings
```

```
{  
  "index" : {  
    "number_of_replicas" : 0  
  }  
}
```

Not for use

Task

- Execute a single command to update the settings of all of our *logs* indices to have **0** replicas.
- Check the status of these indices.
- Make sure everything is green.

Index Patterns

- The first thing you need to define when starting to work with Kibana is the index patterns!
- An index pattern specifies which indices in ES will be analyzed by Kibana.
- You can specify an exact index name, or you can use wildcards (“*”).

Task

- Specify an index pattern for our log data.
- Select `@timestamp` as the ‘time filter field name’.
- In the *Discover* tab, specify ‘Last 5 year’ (upper right corner).
- Zoom on the relevant section (containing data) in the time-series chart.



Demo

- Filtering for *error* @tags.
- Filtering for security @tags.
- Doing it in a query (note ‘and’ vs. ‘AND’).
- Saving the query.
- View the query in the Management tab.

Task

- Using UI operations, filter response status of 503.
- Filter geo.src from the US.
- Now, do that as a query.
- Change the query to filter all response codes representing errors (i.e., 4XX-5XX). Use range syntax (fieldname:[A to B]).
- Save your query.

Task

- Insert the accounts.json into a *bank* index using:

```
curl -H 'Content-Type: application/x-ndjson' -XPOST  
'localhost:9200/bank/account/_bulk?pretty' --data-binary  
@accounts.json
```

- Update index settings to have a green status.
- Define the index pattern.
- Filter results having balance over 10000 (fieldname:>N) of females.
- Save your query.

Highlighting

- Sometimes you don't care about highlighting in the Discovery table.
- You can change that in the 'Advanced Settings' tab.
- Look for: "doc_table:highlight".

Visualization Theory

- A good visualization should communicate “the greatest number of ideas, in the shortest time, using the least amount of ink, in the smallest space” (Edward Tufte).
- Let's see a master-piece...



Visualization Demo

- Create a 'vertical bar' visualization for only females.
- Y-axis is average balance.
- X-axis is age bucketed at 10 years intervals.
- Saving the visualization.

Task

- For the accounts data, create a vertical bar visualization:
- Y-axis is avg. balance.
- Use split-chart by gender with sub-aggregation by age with interval of 5.
- Save the visualization.

Task

- For the logs data, create a heat-map visualization over the ‘security errors’ query.
- Use geo.src for the X-Axis.
- Which country has most originating events?

Task

- Create an area visualization for the *accounts* data.
- Count by State for the top 10 states.
- Create a pie visualization for the *logs* data.
 - Slices should be host names.
- Create a coordinate-map for the *logs* data based on **geo.coordinates** field.

Task

- Create a new index for *accounts*.
- Specify that the *address* field should be analyzed (hint: not a *keyword* type).
- Examine the results in the *Discover* tab.
- Try a search for partial addresses.

Lucene

- So, we've already talked about the inverted index.
- Remember that a document is split into **terms** which are the keys in the inverted index.
- Now, which of the following operations, in your opinion, will perform efficiently with a basic index?
 - Search for term.
 - Search for prefix.
 - Search for suffix.
 - Search for *contains*.



Efficient Index (for search)

- Term – out of the box.
- Prefix – out of the box.
- Suffix – index reverse terms.
- Contains – index ngrams (shingles).
- Numerics – special handling.
- Phonetics – converts terms by formula.
- Geo – convert to strings.
- Suggestions – Levenshtein machine.

Lucence Indexes

- In order for indexes to perform fast, they are immutable.
- I.e., no updates, appends or deletes on the file level.
- Deletion is handled by a special file and documents are only marked as deleted (doesn't reduce the actual index size).
- Update is basically delete and insert.
- An insert? What about immutability?

Index Segments

- Actually, Lucence index is composed of immutable index segments.
- Like mini-indexes.
- A search operation targeting an index will go over all the index segments.
- A merge operation is performed periodically to merge segments and remove deleted documents.
- Thus, adding a new document may actually decrease your index size.

NRT

- ElasticSearch has a **Near Real Time** indexing.
- In order to understand this feature, let's discuss Lucene's commits.

Not for Commercial Use

Lucence Commit

- The actual safe operation that creates new index segments **on disk**.
- An expensive operation which usually can't happen on every document index.

ES translog

- When documents are indexed, they are added to a transaction log (per shard).
 - Also known as **translog**.
 - A journaling solution.
-
- However, not immediately available for search.

ElasticSearch Refresh

- A **refresh** operation will create a new index segment in memory and will allow searching the new data.
- Note that the translog is not cleared and the new segment is only in memory.
- Happens by default on every second.

ElasticSearch Flush

- A flush operation is a commit to Lucence.
- Can be configured in the translog settings.

Not for Commercial Use

Shards

- A shard is an indivisible unit in ES.
- An index is composed of one or more shards.
- Each shard is a full Lucence index.
- Conceptually, there is no difference between a single ES index with two shards and two indexes with one shard.

of Shards

- Note that there is no hard limit on a shard size (up to the hardware).
- Using too many shards may have a high overhead (duplication of internal structure).
- Possibly, search performance will be slower.

Search

- A node receiving a search request becomes the coordinator of the request.
- All nodes are aware of the cluster state.
- Note that the metadata for each node contains also the indexes' mappings (explosion of fields may be a problem).

Search and Index

- A search request will access replicas (if able).
- An index operation will always access the primary (and maybe some replicas (e.g., *quorum*)).
- Routing is hash-based over document-id by default.
- You can change that.
- Beware of unbalance shards.

Scoring

- Search API calculates a score for each document.
- We'll go later over the fine details.
- Documents with higher score are returned first.

Not for Commercial Use

Boosting

- Boosting alters the weights that fields get when computing the relevance score.
- Boosting can (and should) be specified on the query.
- It can also be specified in the mapping (index time).
- Index-time boosting is deprecated (since 5.x) and is generally a bad idea (can't be changed without a reindex).

Fuzziness

- Some queries support fuzziness.
- I.e., similarity matching.
- Calculation is based on Levenshtein Edit Distance.
- Measures the number of one character changes needed to transform one term into the other.

Fuzziness Values

- The fuzziness parameter can be either set to an absolute value or to AUTO (the default).
- Auto fuzziness value depends on the term's length:
 - 0..2 – 0 (exact match).
 - 3..5 – 1.
 - > 5 – 2.

Analyzers

- Analyzers are responsible for the text processing flow of converting text into terms.
- Analyzers are actually composed of:
 - Character filters.
 - Tokenizers.
 - Token filters.

Testing Analyzers

- The *analyze* API can be used to test the analyzers.
- Example:

Console

```
1 | POST _analyze
2 | {
3 |   "tokenizer": "standard",
4 |   "filter": [ "lowercase", "asciifolding" ],
5 |   "text": "Is this déjà vu?"
6 |
7 | }
8 |
9 | {
10|   "tokens": [
11|     {
12|       "token": "is",
13|       "start_offset": 0,
14|       "end_offset": 2,
15|       "type": "<ALPHANUM>",
16|       "position": 0
17|     },
18|     {
19|       "token": "this",
20|       "start_offset": 3,
21|       "end_offset": 7,
22|       "type": "<ALPHANUM>",
23|       "position": 1
24|     },
25|     {
26|       "token": "deja",
27|       "start_offset": 8,
28|       "end_offset": 12,
29|       "type": "<ALPHANUM>",
30|       "position": 2
31|     },
32|     {
33|       "token": "vu",
34|       "start_offset": 13,
35|       "end_offset": 15,
36|       "type": "<ALPHANUM>",
37|       "position": 3
38|     }
39|   ]
40| }
```

Discussion

- Here you see an example using the `_analyze` API.
- Also note the response from the analyzer.
- It's not only terms.
- You also get offsets and ordinal values!

Character Filters

- An analyzer can have zero or more character filters.
- A character filter can transform the text stream before tokenization.
- Let's discuss an example (HTML stripping)...

Tokenizers

- An analyzer must have exactly one tokenizer.
- Responsible for "splitting" text into tokens.
- Let's go over some tokenizers...

Not for Commercial Use

Word Oriented Tokenizers

- standard – tokenizes according to Unicode text segmentation algorithm. Works well for most languages.
- letter – splits according to non-letters.
- lowecase – works like ‘letter’ + ‘lowercase token filter’ (we’ll discuss that filter later).
- uax_url_email -- like the ‘standard’ tokenizer that also recognizes URLs and email addresses.
- classic – English oriented tokenizer. See documentation for more details.

Query & Filter Contexts

- There are two types of ‘contexts’ in the query DSL:
 - Query Context.
 - Filter Context.
- A query clause in a filter context will return a Boolean result regarding a document match check.
- A query clause in a query context will return a **score** representing how well a document matches the query clause.
- Let’s see some examples for query DSL...

Simple Query Clauses

- match_all – matches all document with a constant score of: **1.0**. You can also specify a *boost* parameter to customize the scoring.
- match_none – doesn't match any document.

Match

- match – the common full-text search. Accepts a field name and a value. Analyzes the value and constructs a Boolean query.
- Let's discuss an example...

Example

Console

```
1 GET /_search
2 {
3   "query": {
4     "match_all": {
5       |
6     }
7   }
8 }
9 }
```

▶ 🔍

```
19 _source": {
20   "buildNum": 15629
21 }
22 },
23 {
24   "_index": "megacorp",
25   "_type": "employee",
26   "_id": "2",
27   "_score": 1,
28   "_source": {
29     "first_name": "Johnny",
30     "last_name": "Smith",
31     "age": 27,
32     "about": "I love to go to rock concerts",
33     "interests": [
34       "mtg",
35       "music"
36     ]
37 },
38 },
39 {
40   "_index": "megacorp",
41   "_type": "employee",
42   "_id": "1",
43   "_score": 1,
44   "_source": {
45     "first_name": "John",
46     "last_name": "Smith",
47     "age": 25,
48     "about": "I love to go rock climbing",
49     "interests": [
50       "sports",
51       "music"
52     ]
53 }
```

Example

Console

```
1 GET /_search
2 {
3   "query": {
4     "match": {
5       "about": "rock concerts"
6     }
7   }
8 }
```

```
10  "hits": [
11    "total": 2,
12    "max_score": 0.5716521,
13    "hits": [
14      {
15        "_index": "megacorp",
16        "_type": "employee",
17        "_id": "2",
18        "_score": 0.5716521,
19        "_source": {
20          "first_name": "Johnny",
21          "last_name": "Smith",
22          "age": 27,
23          "about": "I love to go to rock concerts",
24          "interests": [
25            "mtg",
26            "music"
27          ]
28        }
29      },
30      {
31        "_index": "megacorp",
32        "_type": "employee",
33        "_id": "1",
34        "_score": 0.26742277,
35        "_source": {
36          "first_name": "John",
37          "last_name": "Smith",
38          "age": 25,
39          "about": "I love to go rock climbing",
40          "interests": [
41            "sports",
42            "music"
43          ]
44        }
45      }
46    ]
47  }
```

Example

Console

```
1 | GET /_search
2 | {
3 |   "query": {
4 |     "match": {
5 |       "about": {
6 |         "query": "rock concerts",
7 |         "operator": "and"
8 |       }
9 |     }
10|   }
11| }
```

```
1 | {
2 |   "took": 0,
3 |   "timed_out": false,
4 |   "_shards": {
5 |     "total": 6,
6 |     "successful": 6,
7 |     "skipped": 0,
8 |     "failed": 0
9 |   },
10|   "hits": {
11|     "total": 1,
12|     "max_score": 0.5716521,
13|     "hits": [
14|       {
15|         "_index": "megacorp",
16|         "_type": "employee",
17|         "_id": "2",
18|         "_score": 0.5716521,
19|         "_source": {
20|           "first_name": "Johnny",
21|           "last_name": "Smith",
22|           "age": 27,
23|           "about": "I love to go to rock concerts",
24|           "interests": [
25|             "mtg",
26|             "music"
27|           ]
28|         }
29|       }
30|     ]
31|   }
32| }
```

Example

Console

```
1 GET /_search
2 {
3   "query": {
4     "match": {
5       "about": {
6         "query": "rock concerts",
7         "operator": "or"
8       }
9     }
10  }
11 }
```

```
10 "hits": [
11   "total": 2,
12   "max_score": 0.5716521,
13   "hits": [
14     {
15       "_index": "megacorp",
16       "_type": "employee",
17       "_id": "2",
18       "_score": 0.5716521,
19       "_source": {
20         "first_name": "Johnny",
21         "last_name": "Smith",
22         "age": 27,
23         "about": "I love to go to rock concerts",
24         "interests": [
25           "mtg",
26           "music"
27         ]
28       }
29     },
30     {
31       "_index": "megacorp",
32       "_type": "employee",
33       "_id": "1",
34       "_score": 0.26742277,
35       "_source": {
36         "first_name": "John",
37         "last_name": "Smith",
38         "age": 25,
39         "about": "I love to go rock climbing",
40         "interests": [
41           "sports",
42           "music"
43         ]
44       }
45     }
46   ]
47 }
```

Explain

Console

```
1 GET /_search
2 {
3   "explain": true,
4   "query": {
5     "match": {
6       "about": {
7         "query": "rock concerts",
8         "operator": "or"
9       }
10    }
11  }
12 }
```

```
10 "hits": [
11   "total": 2,
12   "max_score": 0.5716521,
13   "hits": [
14     {
15       "_shard": "[megacorp][2]",
16       "_node": "LgNEouTCQ9ScQCcb4MuDQ",
17       "_index": "megacorp",
18       "_type": "employee",
19       "_id": "2",
20       "_score": 0.5716521,
21       "_source": {
22         "first_name": "Johnny",
23         "last_name": "Smith",
24         "age": 27,
25         "about": "I love to go to rock concerts",
26         "interests": [
27           "mtg",
28           "music"
29         ]
30       },
31       "_explanation": {
32         "value": 0.5716521,
33         "description": "sum of:",
34         "details": [
35           {
36             "value": 0.28582606,
37             "description": "weight(about:rock in 0) [PerFieldSimilarity], result of:",
38             "details": [
39               {
40                 "value": 0.28582606,
41                 "description": "score(doc=0,freq=1.0 = termFreq=1.0\\n), product of:",
42                 "details": [
43                   {
44                     "value": 0.2876821,
45                     "description": "idf, computed as log(1 + (docCount - docFreq + 0.5) / (docFreq + 0.5)) from:",
46                   }
47                 ]
48               }
49             ]
50           }
51         ]
52       }
53     }
54   ]
55 }
```

Match Phrase

- match phrase – analyzes and creates a phrase query.
- It cares about the order.
- Let's discuss an example...

Example

Console

```
1 GET /_search
2 {
3   "query": {
4     "match_phrase": {
5       "about": "rock concerts"
6     }
7   }
8 }
```

```
1 {
2   "took": 0,
3   "timed_out": false,
4   "_shards": {
5     "total": 6,
6     "successful": 6,
7     "skipped": 0,
8     "failed": 0
9   },
10  "hits": {
11    "total": 1,
12    "max_score": 0.5716521,
13    "hits": [
14      {
15        "_index": "megacorp",
16        "_type": "employee",
17        "_id": "2",
18        "_score": 0.5716521,
19        "_source": {
20          "first_name": "Johnny",
21          "last_name": "Smith",
22          "age": 27,
23          "about": "I love to go to rock concerts",
24          "interests": [
25            "mtg",
26            "music"
27          ]
28        }
29      }
30    ]
31  }
32 }
```

Example

Console

```
1 GET /_search
2 {
3   "query": {
4     "match_phrase": {
5       "about": "concerts rock"
6     }
7   }
8 }
```

```
1 {  
2   "took": 0,  
3   "timed_out": false,  
4   "_shards": {  
5     "total": 6,  
6     "successful": 6,  
7     "skipped": 0,  
8     "failed": 0  
9   },  
10  "hits": {  
11    "total": 0,  
12    "max_score": null,  
13    "hits": []  
14  }  
15 }
```

Example

Console

```
1 GET /_search
2 {
3   "query": {
4     "match": {
5       "about": "concerts rock"
6     }
7   }
8 }
```

```
1 {
2   "took": 0,
3   "timed_out": false,
4   "_shards": {
5     "total": 6,
6     "successful": 6,
7     "skipped": 0,
8     "failed": 0
9   },
10  "hits": {
11    "total": 2,
12    "max_score": 0.5716521,
13    "hits": [
14      {
15        "_index": "megacorp",
16        "_type": "employee",
17        "_id": "2",
18        "_score": 0.5716521,
19        "_source": {
20          "first_name": "Johnny",
21          "last_name": "Smith",
22          "age": 27,
23          "about": "I love to go to rock concerts",
24          "interests": [
25            "mtg",
26            "music"
27          ]
28        }
29      },
30      {
31        "_index": "megacorp",
32        "_type": "employee",
33        "_id": "1",
34        "_score": 0.26742277,
35        "_source": {
36          "first_name": "John",
37          "last_name": "Smith",
38          "age": 25,
39          "about": "I love to go rock climbing",
40        }
41      }
42    ]
43  }
44}
```

Match Phrase

- match_phrase_prefix – analyzes and creates a phrase query.
- Allows for expansions for the last term.
- By default, *max_expansions* is 50.
- Consider lowering it if you plan to use it.
- Not the best solution for ‘search-as-you-type’.

Multi Match

- multi-match – match query for multiple fields.

Console

```
1 GET /_search
2 {
3   "query": {
4     "multi_match": {
5       "query" : "music climbing",
6       "fields": ["about", "interests"]
7     }
8   }
9 }
```

Multi Match

- You can boost fields and do wildcards selection:

Console

```
1 GET /_search
2 {
3   "query": {
4     "multi_match": {
5       "query" : "sports concerts",
6       "fields": ["about", "interests"]
7     }
8   }
9 }
```

```
1 {
2   "took": 0,
3   "timed_out": false,
4   "_shards": {
5     "total": 6,
6     "successful": 6,
7     "skipped": 0,
8     "failed": 0
9   },
10  "hits": {
11    "total": 2,
12    "max_score": 0.28582606,
13    "hits": [
14      {
15        "_index": "megacorp",
16        "_type": "employee",
17        "_id": "2",
18        "_score": 0.28582606,
19        "_source": {
20          "first_name": "Johnny",
21          "last_name": "Smith",
22          "age": 27,
23          "about": "I love to go to rock concerts",
24          "interests": [
25            "mtg",
26            "music"
27          ]
28        }
29      },
30      {
31        "_index": "megacorp",
32        "_type": "employee",
33        "_id": "1",
34        "_score": 0.25811607,
35        "_source": {
36          "first_name": "John",
37          "last_name": "Smith",
38          "age": 25,
39          "about": "I love to go rock climbing",
40        }
41      }
42    ]
43  }
44 }
```

Multi Match

Console

```
1 GET /_search
2 {
3     "query": {
4         "multi_match": {
5             "query" : "sports concerts",
6             "fields": ["about", "interests^2"]
7         }
8     }
9 }
```

```
1 {
2     "took": 0,
3     "timed_out": false,
4     "_shards": {
5         "total": 6,
6         "successful": 6,
7         "skipped": 0,
8         "failed": 0
9     },
10    "hits": {
11        "total": 2,
12        "max_score": 0.51623213,
13        "hits": [
14            {
15                "_index": "megacorp",
16                "_type": "employee",
17                "_id": "1",
18                "_score": 0.51623213,
19                "_source": {
20                    "first_name": "John",
21                    "last_name": "Smith",
22                    "age": 25,
23                    "about": "I love to go rock climbing",
24                    "interests": [
25                        "sports",
26                        "music"
27                    ]
28                }
29            },
30            {
31                "_index": "megacorp",
32                "_type": "employee",
33                "_id": "2",
34                "_score": 0.28582606,
35                "_source": {
36                    "first_name": "Johnny",
37                    "last_name": "Smith",
38                    "age": 27,
39                    "about": "I love to go to rock concerts",

```

Multi Match

Console

```
1 GET /_search
2 {
3     "query": {
4         "multi_match": {
5             "query" : "sports concerts",
6             "fields": ["ab*^3", "interests^2"]
7         }
8     }
9 }
```

```
1 {
2     "took": 0,
3     "timed_out": false,
4     "_shards": {
5         "total": 6,
6         "successful": 6,
7         "skipped": 0,
8         "failed": 0
9     },
10    "hits": {
11        "total": 2,
12        "max_score": 0.8574782,
13        "hits": [
14            {
15                "_index": "megacorp",
16                "_type": "employee",
17                "_id": "2",
18                "_score": 0.8574782,
19                "_source": {
20                    "first_name": "Johnny",
21                    "last_name": "Smith",
22                    "age": 27,
23                    "about": "I love to go to rock concerts",
24                    "interests": [
25                        "mtg",
26                        "music"
27                    ]
28                }
29            },
30            {
31                "_index": "megacorp",
32                "_type": "employee",
33                "_id": "1",
34                "_score": 0.51623213,
35                "_source": {
36                    "first_name": "John",
37                    "last_name": "Smith",
38                    "age": 25,
39                    "about": "I love to go rock climbing",
40                }
41            }
42        ]
43    }
44 }
```

tools/console

Multi Match

- Multi-field match queries operate according to the **type** field.
- Possible values:
 - best fields (default).
 - most fields.
 - cross fields.
 - phrase.
 - phrase prefix.

Multi Match

- **best_fields** – uses the best field's score.
- **most_fields** – combining the scores from all the fields (useful when the same value is handled with different analyzers).
- **phrase, phrase_prefix** – like best_fields but using a phrase/phrase_prefix query.
- **cross_fields** – conceptually combines fields with the same analyzer into a single field.

Common Query Clause

- When you execute a search like: “The tall student”, the search generates three term queries (one for each word).
- Of-course, querying for “*the*” will generate much more hits than the other two.
- A common solution for this problem was: stop words.
- I.e., ignore term with high frequency (don’t even store these).
- The problem is that sometimes you still need these words (you hurt yours precision and recall).

Common Query Clause

- Another solution is to use the **common term** query.
- The common term query assign each term to either the low frequency group (important) or the high frequency group (less important).
- First, documents are searched for the low-frequency terms.
- Second, the less important terms are only searched for documents matching the first group!
- Thus, we are not paying a huge performance for the high-frequency terms.

Common Query Clause

- Group division is controlled by cutoff frequency.
- Can be specified as a relative value (0..1) or an absolute value (>1).
- If the query contains only high-frequency words, it will be executed as an “*AND*” query.

Term Queries

- Term queries are not analyzed.
- Usually used for: numeric, dates and other fields with structure (non textual).
- term – checks for an **exact** value in the inverted index!
- Note that it should be used for keyword fields and **not** text fields.

Range Term Query

- Matches documents based on field's value being inside the specified range.
- Works for text, numerics and dates.
- Accepts the parameters: *lte*, *lt*, *gte*, *gt*.

Exists Query

- Returns all documents that have at least one **non null** value in the specified field.
- Example:

No *exists*

```
Console
1 GET /_search
2 {
3   "query": {
4     "exists": {
5       "field": "about"
6     }
7   }
8 }
```

```
1 {
2   "took": 0,
3   "timed_out": false,
4   "_shards": {
5     "total": 6,
6     "successful": 6,
7     "skipped": 0,
8     "failed": 0
9   },
10  "hits": {
11    "total": 2,
12    "max_score": 1,
13    "hits": [
14      {
15        "_index": "megacorp",
16        "_type": "employee",
17        "_id": "2",
18        "_score": 1,
19        "_source": {
20          "first_name": "Johnny",
21          "last_name": "Smith",
22          "age": 27,
23          "about": "I love to go to rock concerts",
24          "interests": [
25            "mtg",
26            "music"
27          ]
28        }
29      },
30      {
31        "_index": "megacorp",
32        "_type": "employee",
33        "_id": "1",
34        "_score": 1,
35        "_source": {
36          "first_name": "John",
37          "last_name": "Smith",
38          "age": 25,
39        }
40      }
41    ]
42  }
43 }
```

More Term Queries

- prefix – matching terms starting with the specified prefix.
- wildcard – supports '*' and '?'. Has significant performance cost.
Don't start with a wildcard value.
- regexp – the usual suspect.
- type – filters according to the mapping type.
- ids – filters according to the `_uid` field.

Bool Query

- Combines multiple queries using the following typed occurrences:
- must – query must appear in matching documents and adds to the score.
- filter – just like 'must', but uses a filter context which doesn't contribute to the score and is cached.
- must not – query must not appear in matching documents. Doesn't contribute to the score (filter context).
- should – discussed on the next page...

should

- If this is a query context, then:
 - If it has a *must* or *filter*, then the *should* part will only affect the score.
- For all other scenarios:
 - At least one *should* query must match.

Not for Commercial Use

XPack

- The commercial package by elastic.
- We'll mainly focus on Machine Learning.
- Note that security is automatically added.
- Default username is: *elastic*.
- Default password is: *changeme*.



XPack ML

- Anomaly detection and trend predictions.
- Let's do it..

Not for Commercial Use

Task

- Add the data files to your logstash bin folder.
- Use `cat/type stock.csv | ./logstash -f stocks.csv`
- Repeat for other data files as well (with the respective conf files).

Task

- Examine your data and create the relevant index patterns.
- Create a Single Metric ML job over stocks.
- What should be the bucket span ?
- Check Count agg vs. Mean.
- Which is more relevant?



Task

- Create a job and view results.
- Explore the anomalies.

Not for Commercial Use

Task

- Explore *farequote*.
- Where is the data on the timeline? (use Discover tab).
- Create a Visualization (data table) to find the number of unique airlines.

Not for Commercial Use

Task

- Create a Single Metric job to find Count anomalies.
- What should be the bucket span?
- When the anomalies happened?

Not for Commercial Use

Task

- Create a Multi Metric job to find mean response time anomalies by which airline (a no-split job).
- This is each anomaly compared to the whole dataset.
- Create a new job to do the same but each airline data by itself?
- Compare the results.



Task

- Create and compare a Count by airline multi-metric job to the single job you've created previously.
- Why are they different?

Not for Commercial Use



Task

- Create an advanced Job over the cloudwatch indices.
- Select *mean* detector for all metrics partitioned by instance with bucket span of 1 hour.
- Explore the results.

Task

- Clone the job and add to the datafeed tab the following query:
- `{"term": {"region.keyword": {"value": "eu-central-1","boost": 1}}}`
- Compare the results with previous job.

Timelion

- Timelion is a powerful tool allowing you to explore, aggregate and compare time-series data from different sources.
- Of course ElasticSearch is included.

Timelion

- Supports a sophisticated query language to aggregate data from the datasources.
- Let's see it in action...

Not for Commercial Use

REST API

- The nodes in the cluster communicate with each other using the ‘transport interface’ (TCP-based).
- Nevertheless, the nodes also expose an HTTP-based REST API.
- Let’s see an example...

REST API Examples

Console

```
1  {
2    "_nodes": {
3      "total": 1,
4      "successful": 1,
5      "failed": 0
6    },
7    "cluster_name": "elasticsearch",
8    "nodes": {
9      "LgNEouTCQ9ScQCby4MuDQ": {
10        "name": "LgNEouT",
11        "transport_address": "127.0.0.1:9300",
12        "host": "127.0.0.1",
13        "ip": "127.0.0.1",
14        "version": "5.6.8",
15        "build_hash": "688ecce",
16        "total_indexing_buffer": 207775334,
17        "roles": [
18          "master",
19          "data",
20          "ingest"
21        ],
22        "settings": {
23          "client": {
24            "type": "node"
25          },
26          "cluster": {
27            "name": "elasticsearch"
28          },
29          "http": {
30            "type": {
31              "default": "netty4"
32            }
33          },
34          "node": {
35            "name": "LgNEouT"
36          }
37        }
38      }
39    }
40  }
```

REST API Examples

Console

1

2 GET _cluster/health



1 {

```
2   "cluster_name": "elasticsearch",
3   "status": "yellow",
4   "timed_out": false,
5   "number_of_nodes": 1,
6   "number_of_data_nodes": 1,
7   "active_primary_shards": 1,
8   "active_shards": 1,
9   "relocating_shards": 0,
10  "initializing_shards": 0,
11  "unassigned_shards": 1,
12  "delayed_unassigned_shards": 0,
13  "number_of_pending_tasks": 0,
14  "number_of_in_flight_fetch": 0,
15  "task_max_waiting_in_queue_millis": 0,
16  "active_shards_percent_as_number": 50
```

17 }

: