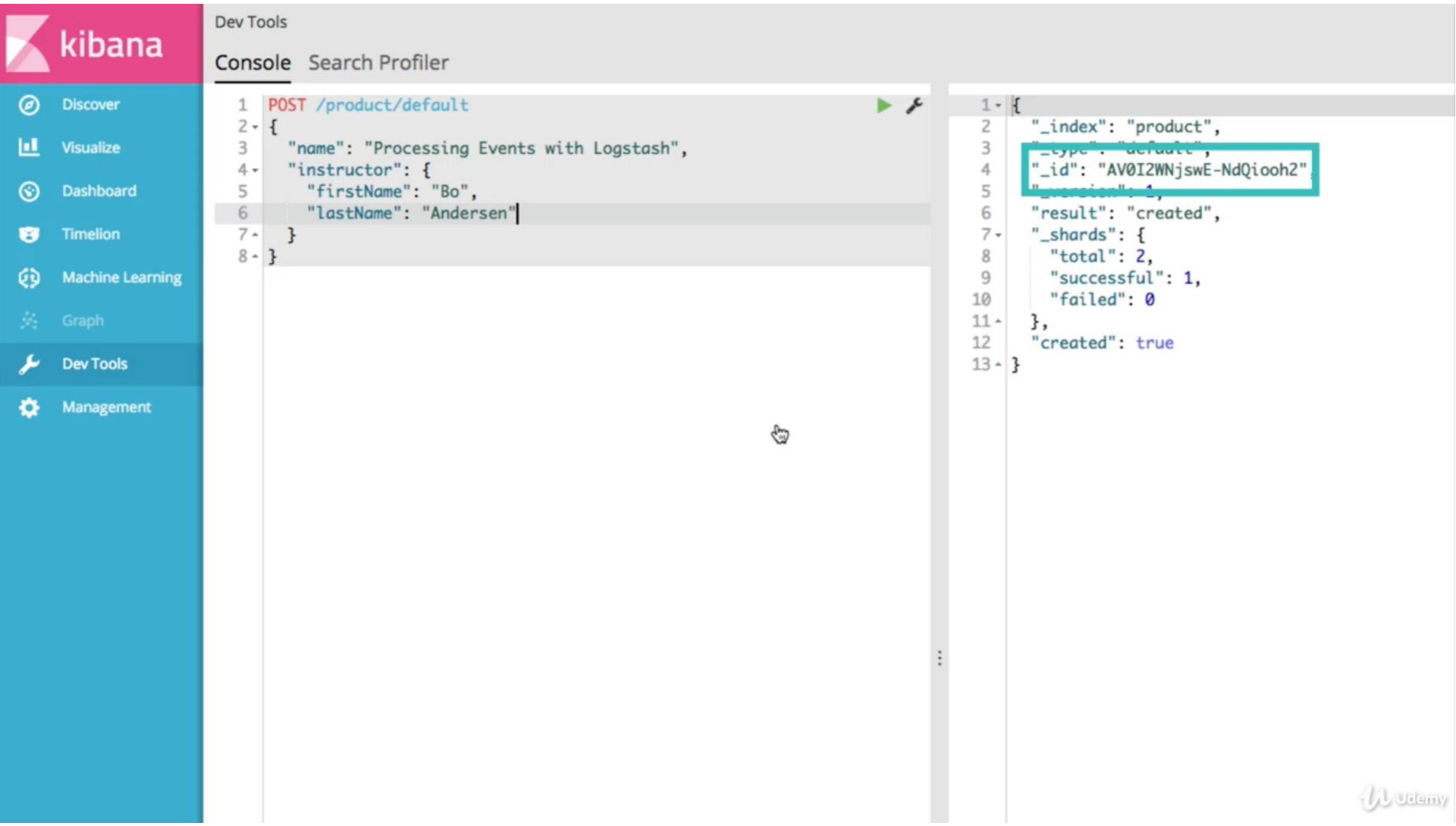
Documents are Immutable

- Elasticsearch documents are immutable (!)
- We actually *replaced* documents in this lecture
- The Update API did some things for us, making it *look* like we updated documents
- The Update API is simpler and saves some network traffic

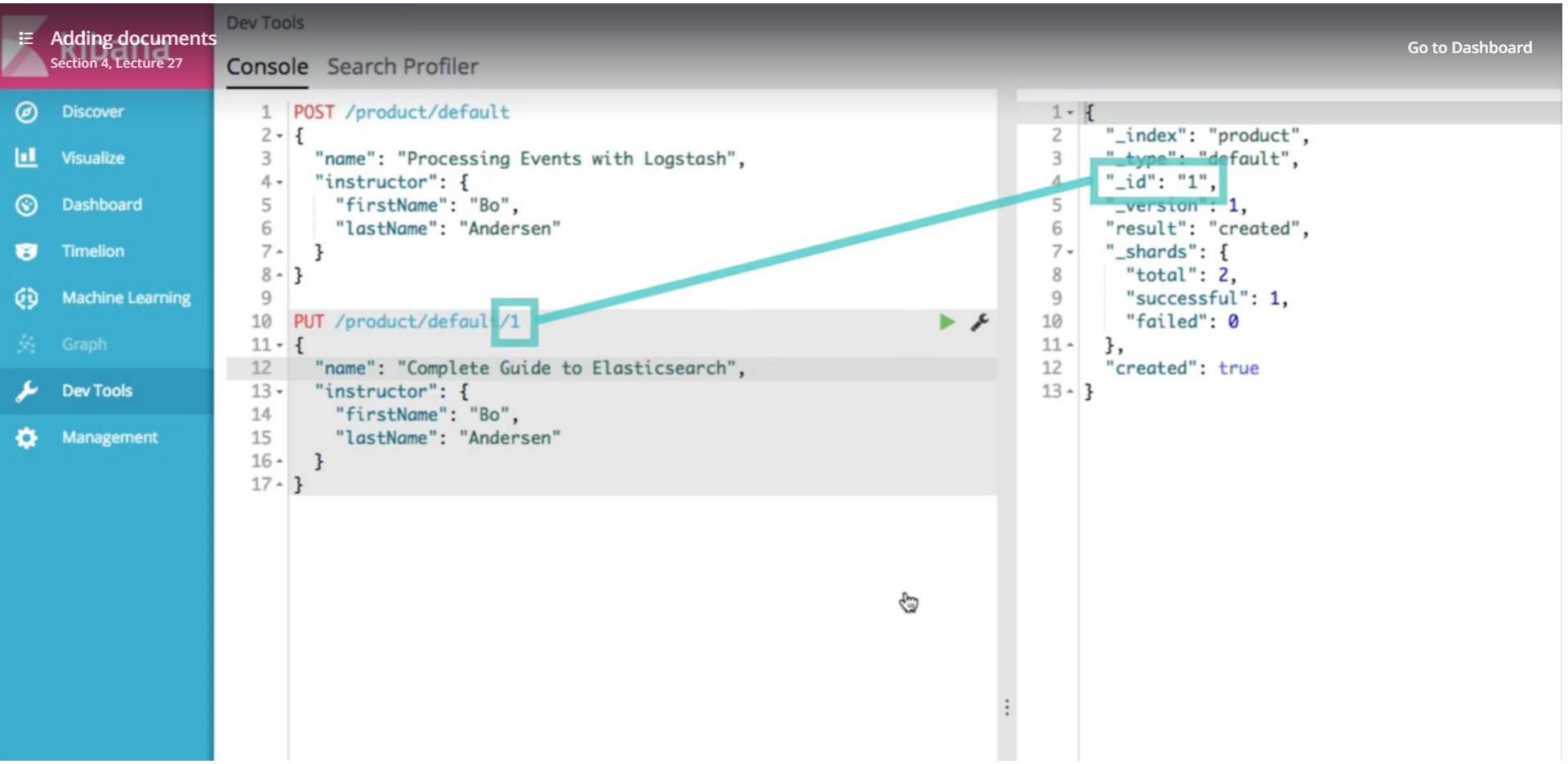
How update API Works

- The current document is retrieved
- The field values are changed
- The existing document is *replaced* with the modified document
- We *could* do the exact same thing at the application level

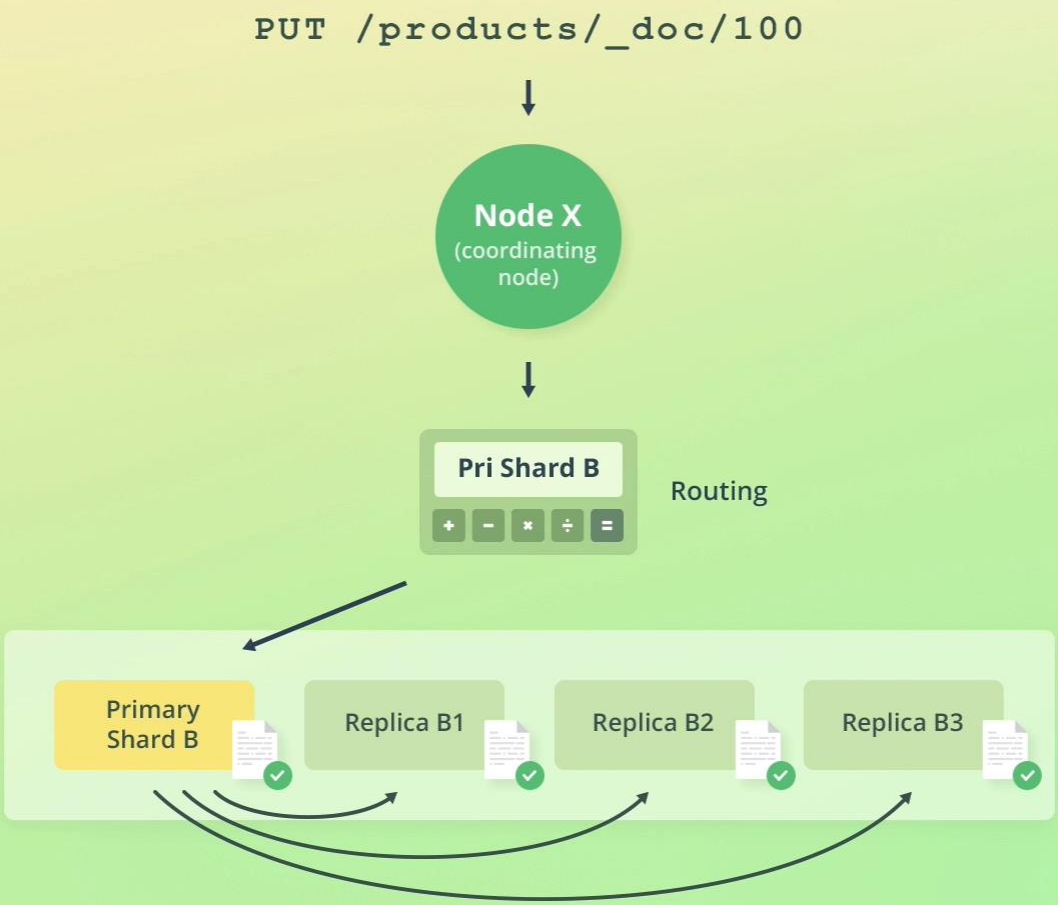
Managing Documents – Adding Documents

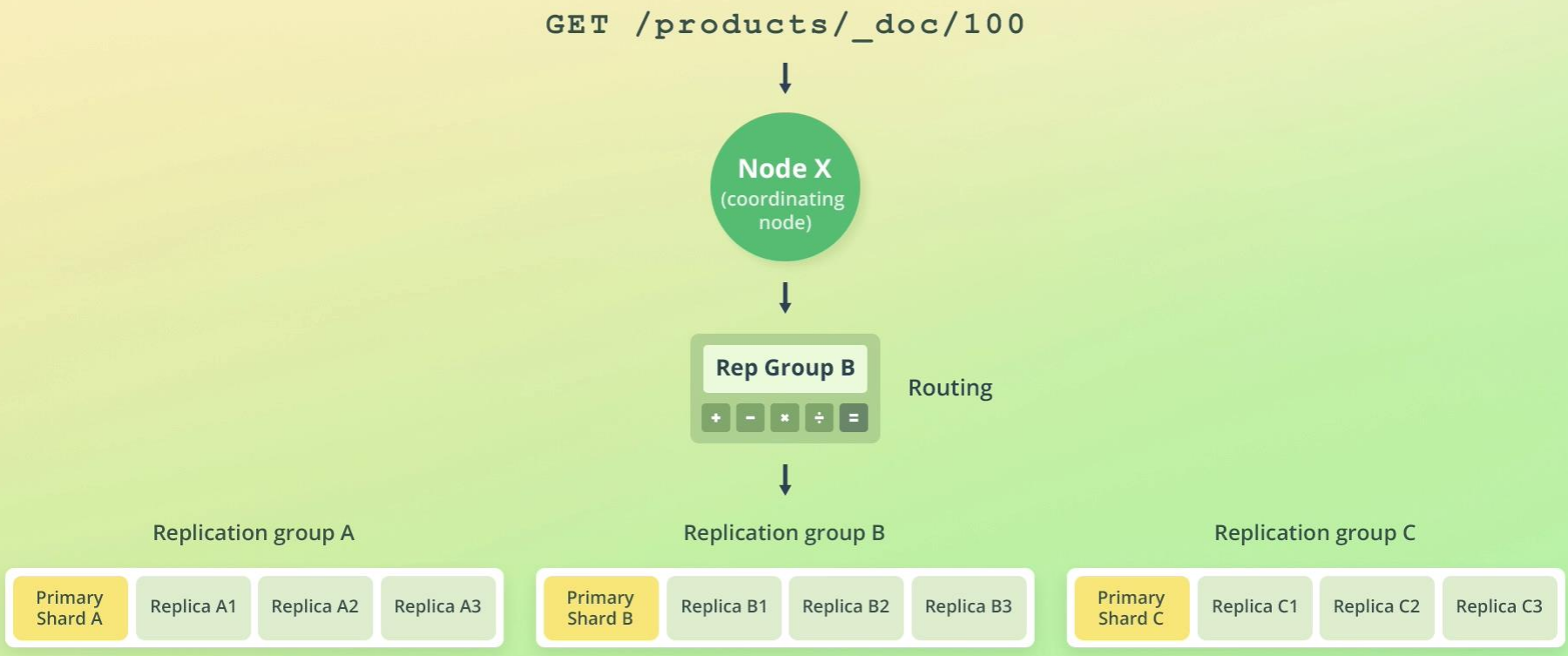


Managing Documents – Adding Documents

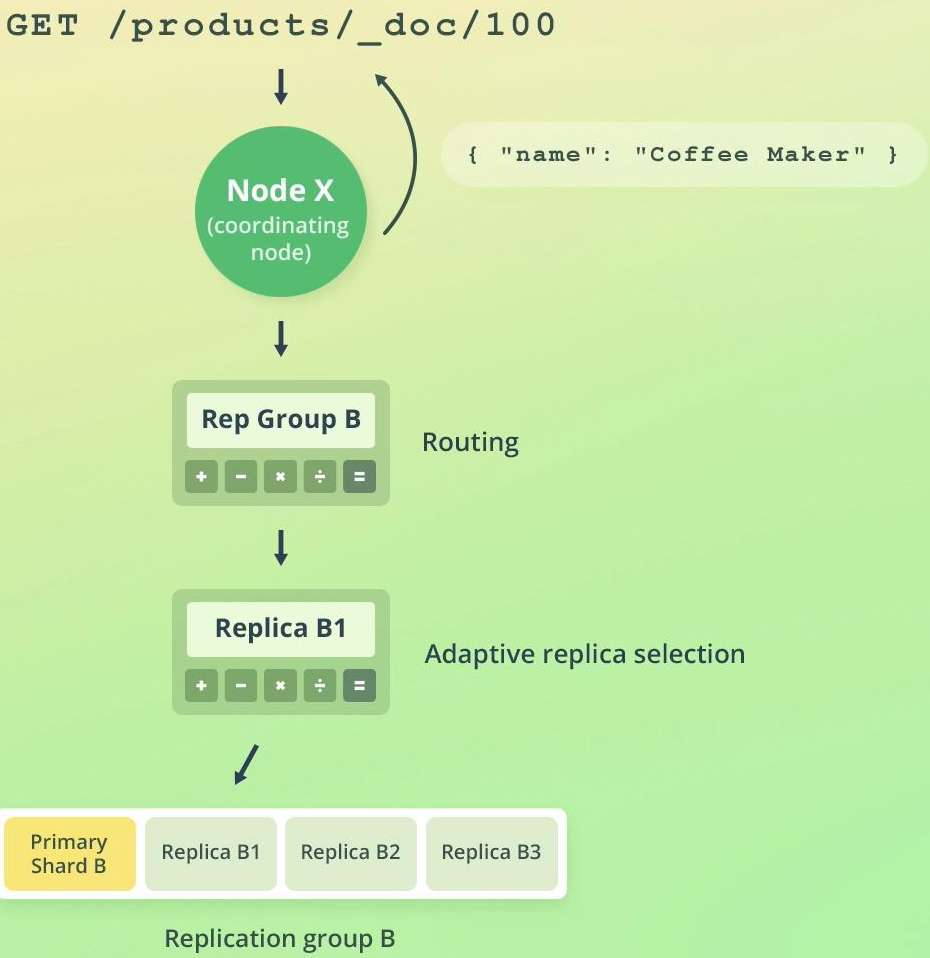


Writing Documents

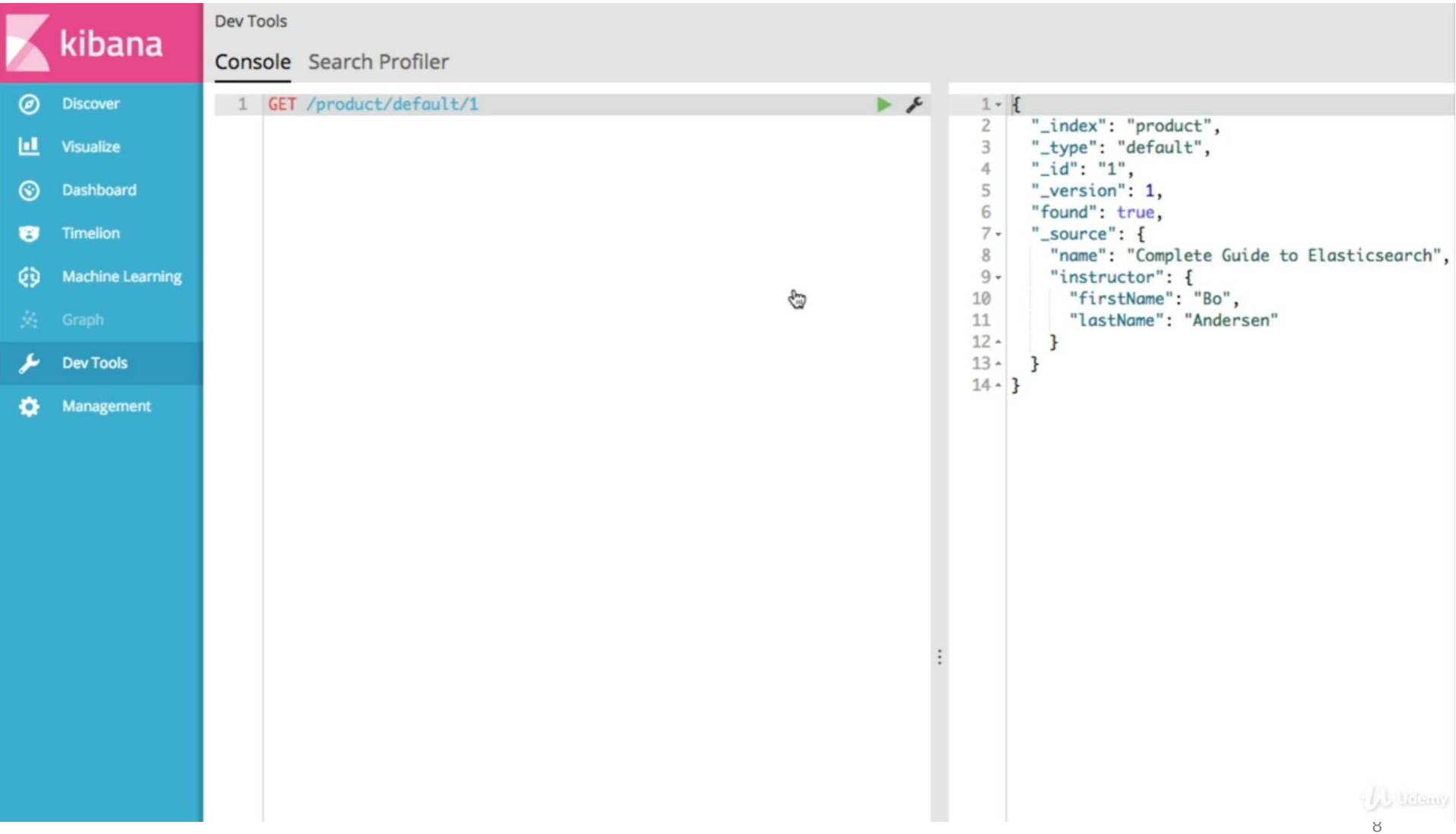



GET /products/_doc/100
Node X
(coordinating node)
Rep Group B
Routing
Replication group A
Primary Shard A
Replica A1
Replica A2
Replica A3
Replication group B
Primary Shard B
Replica B1
Replica B2
Replica B3
Replication group C
Primary Shard C
Replica C1
Replica C2
Replica C3

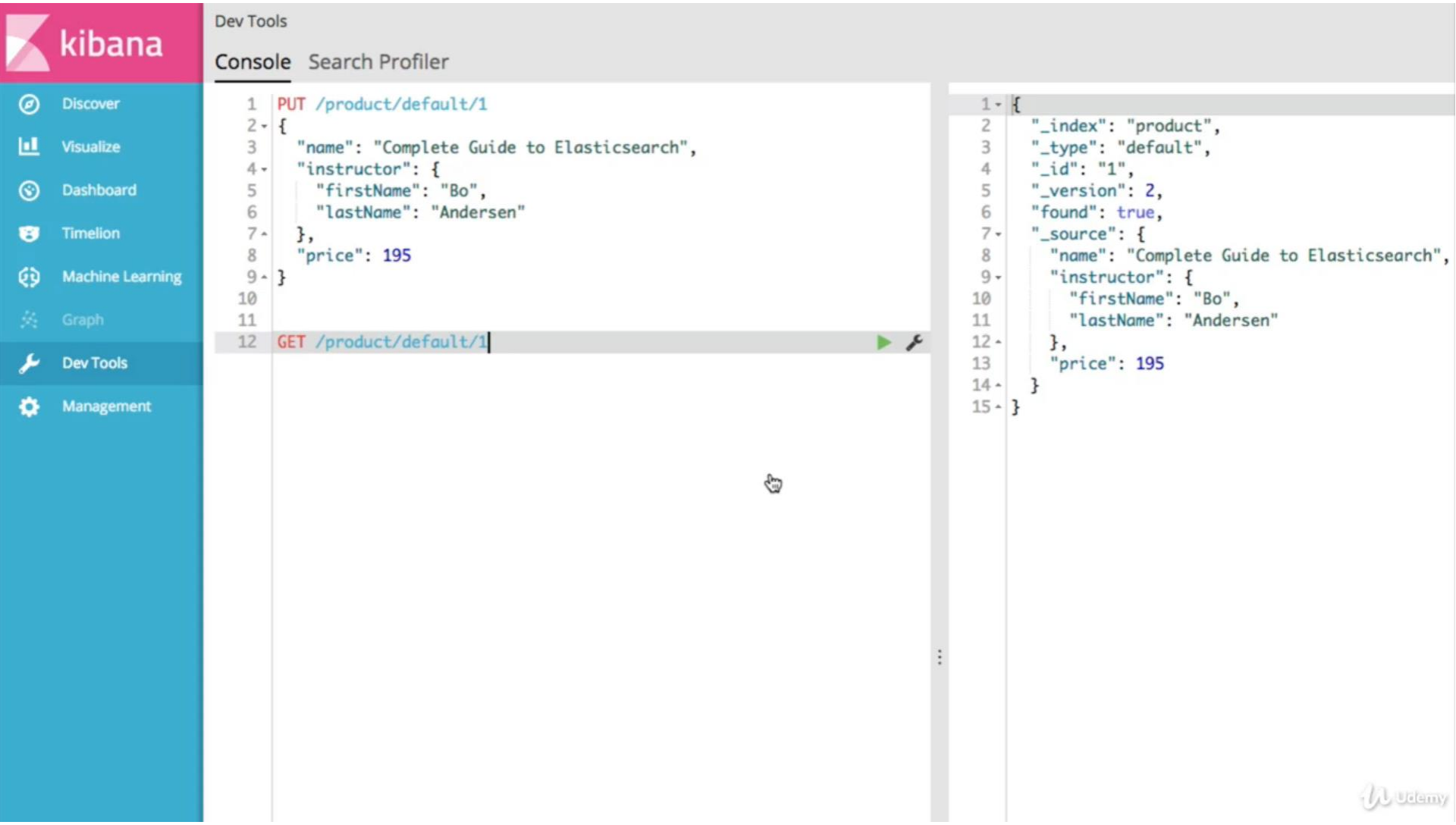
Reading Documents



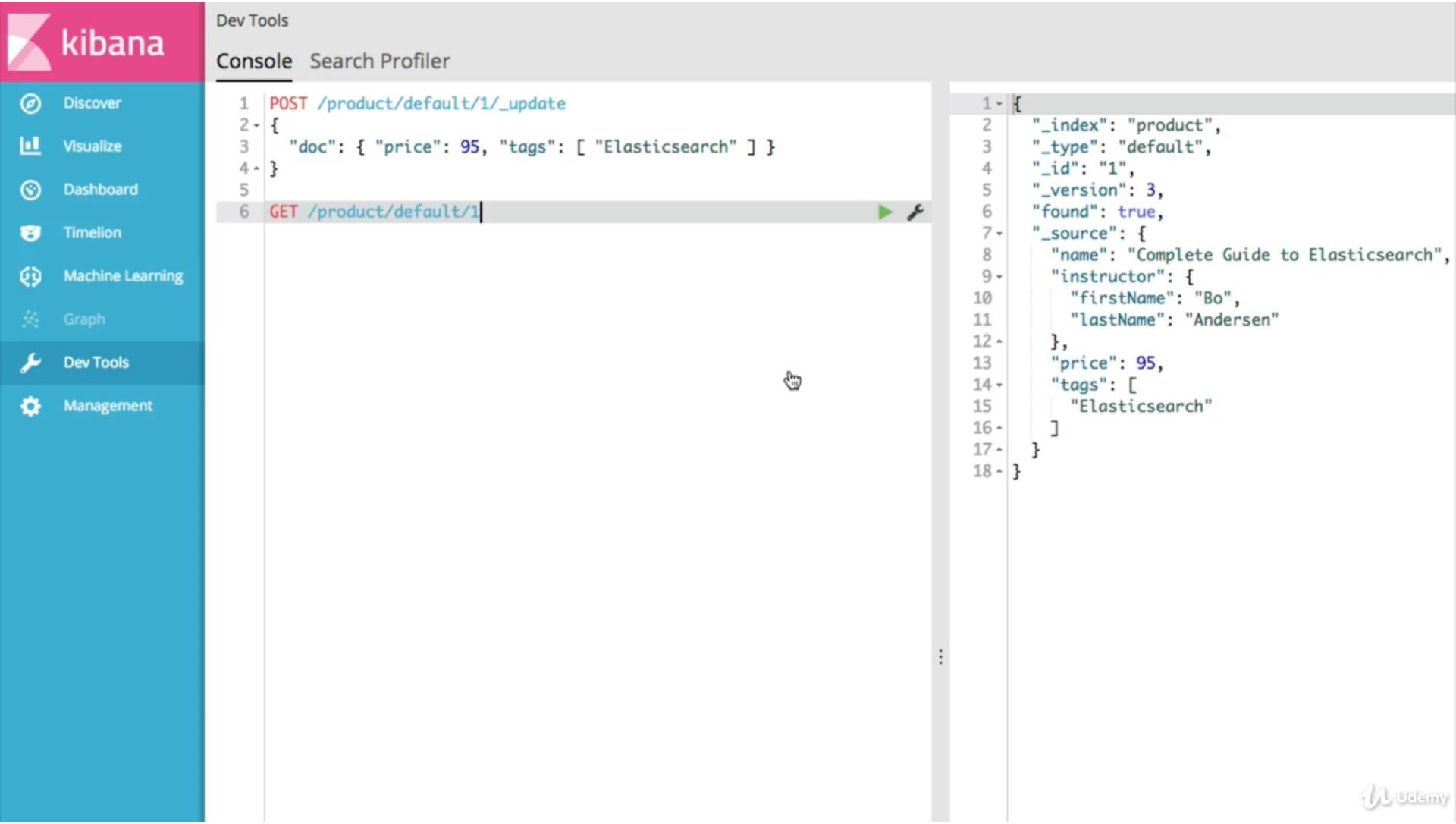
Managing Documents – Retrieving Documents



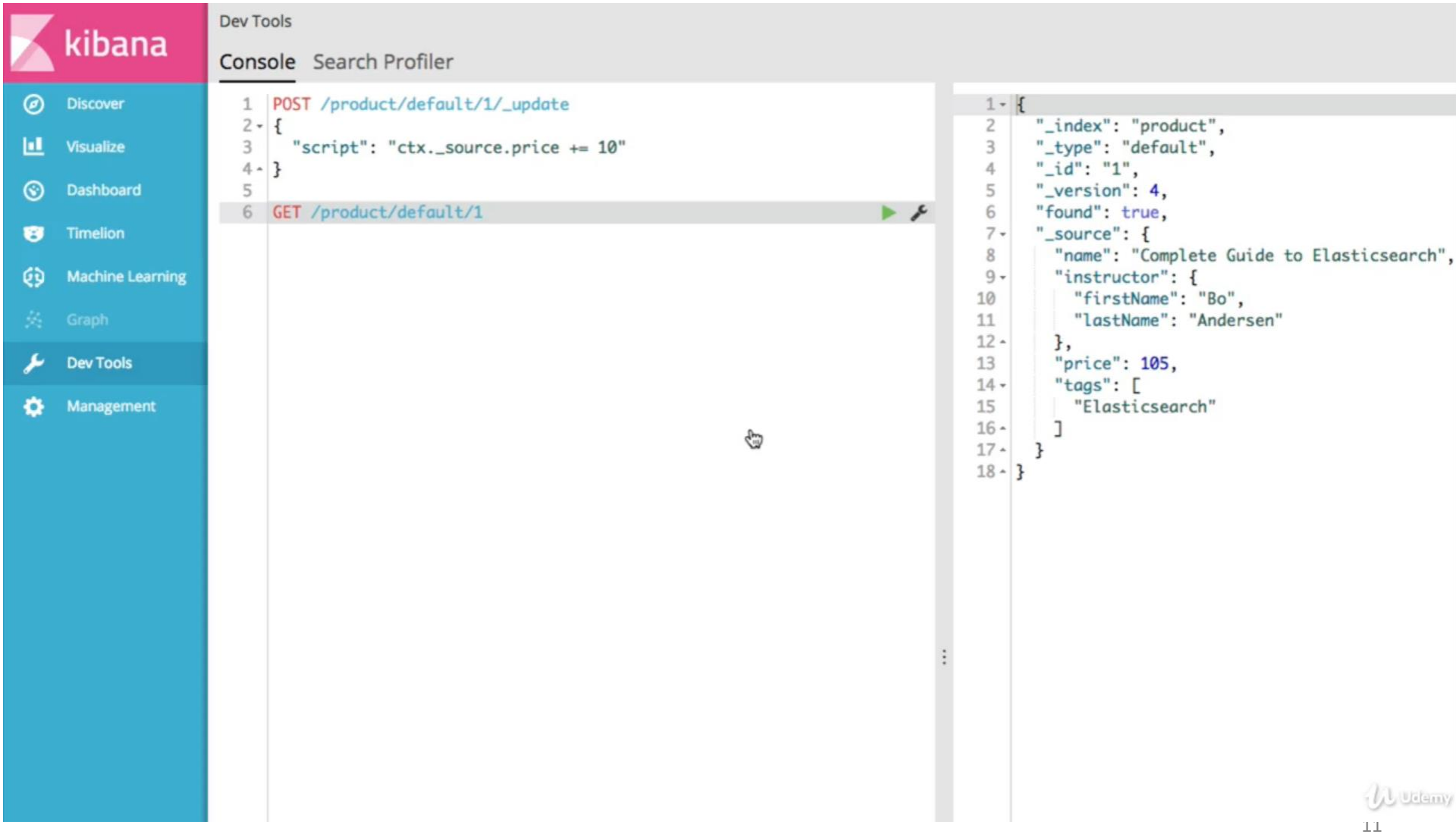
Managing Documents – Replace Documents



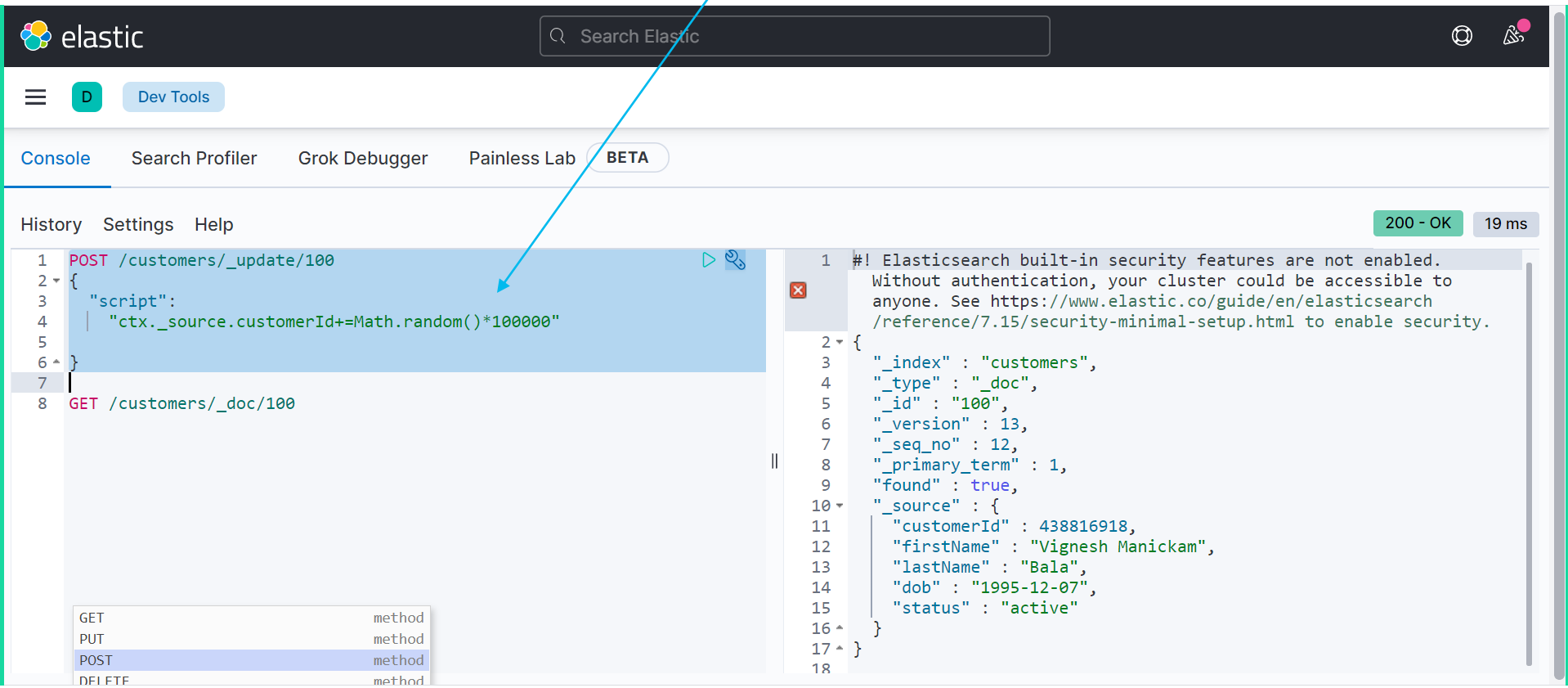
Managing Documents – Update Documents



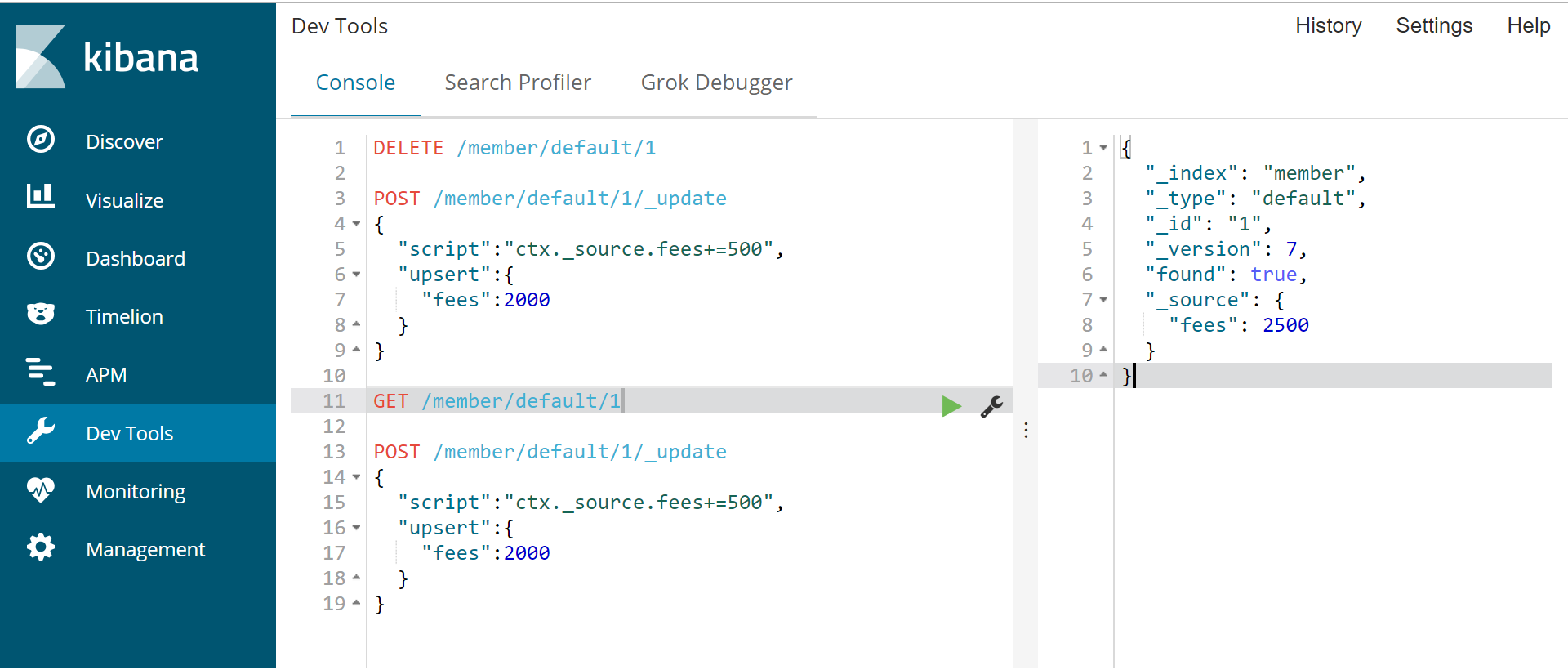
Managing Documents – Scripting Documents



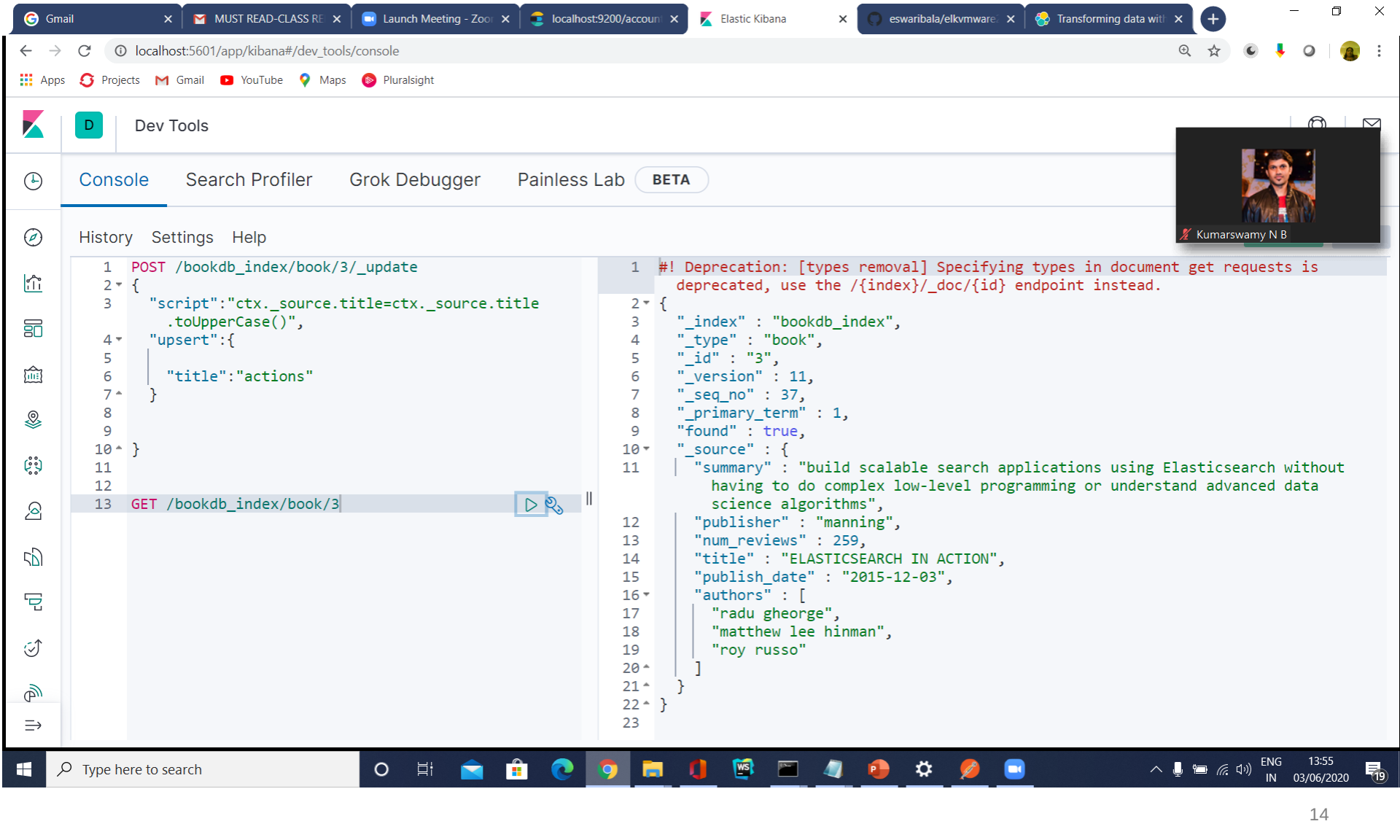
Managing Documents – Scripting Documents



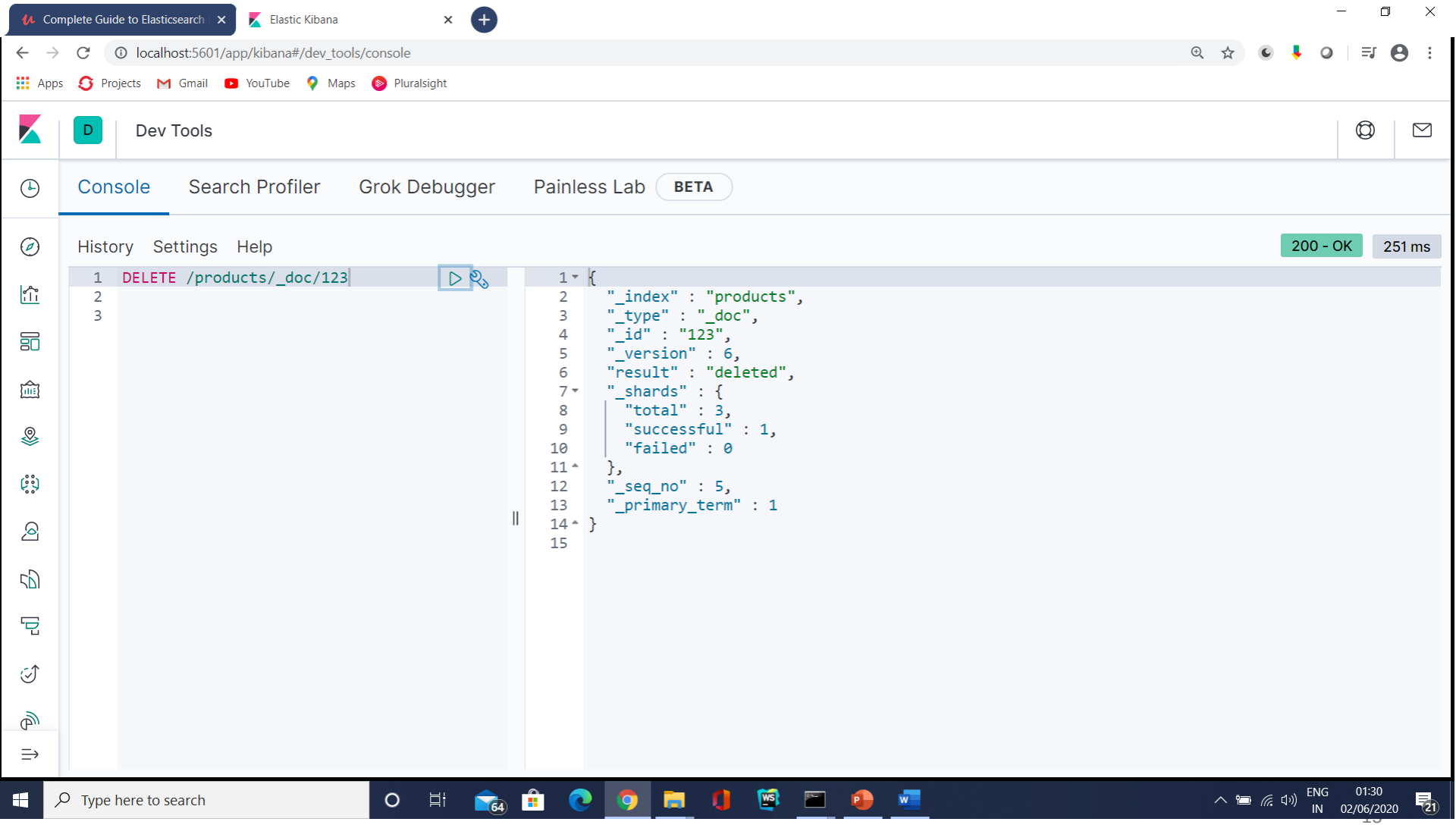
Managing Documents – Upserting Documents



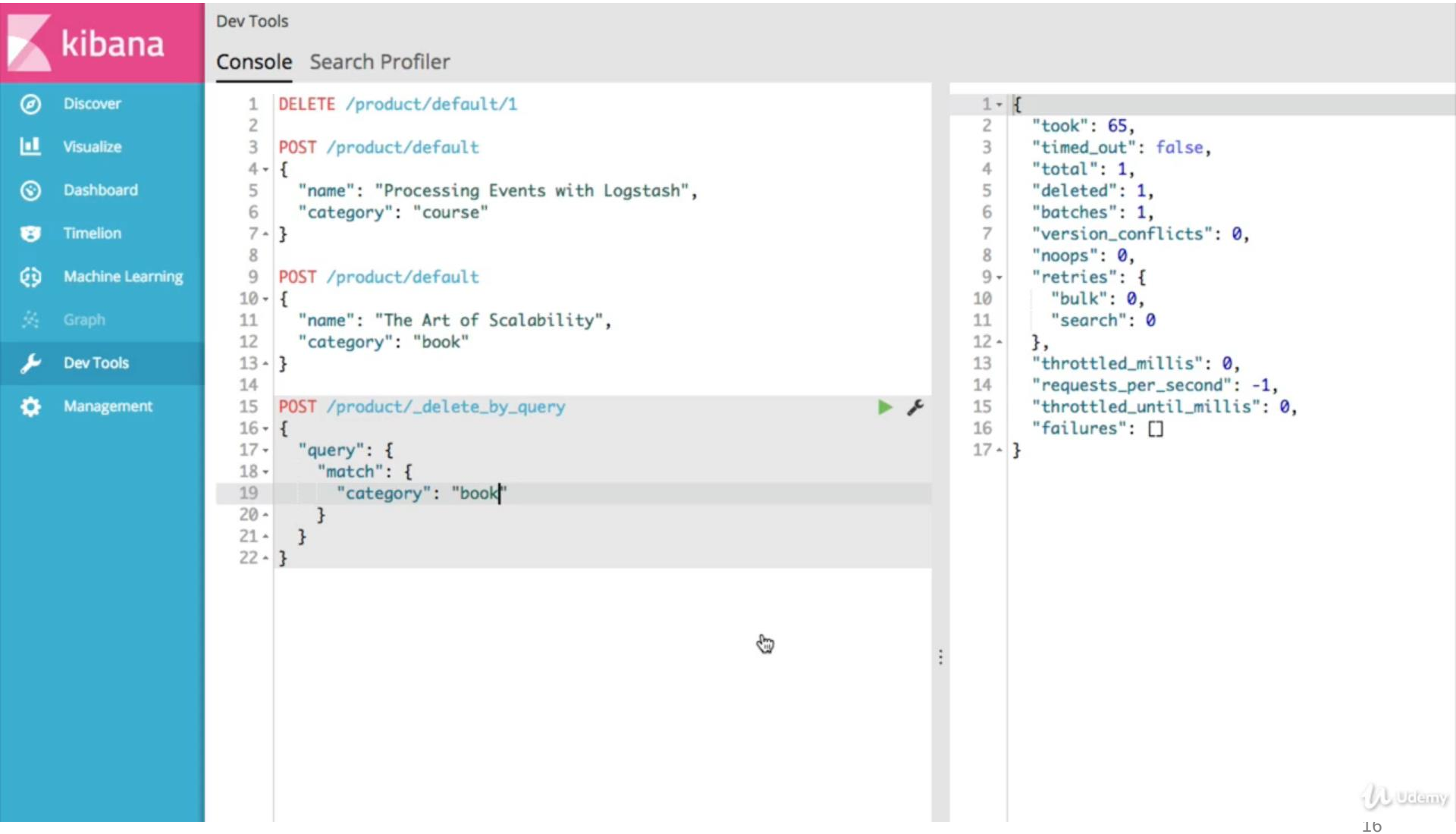
Managing Documents – Upserting Documents



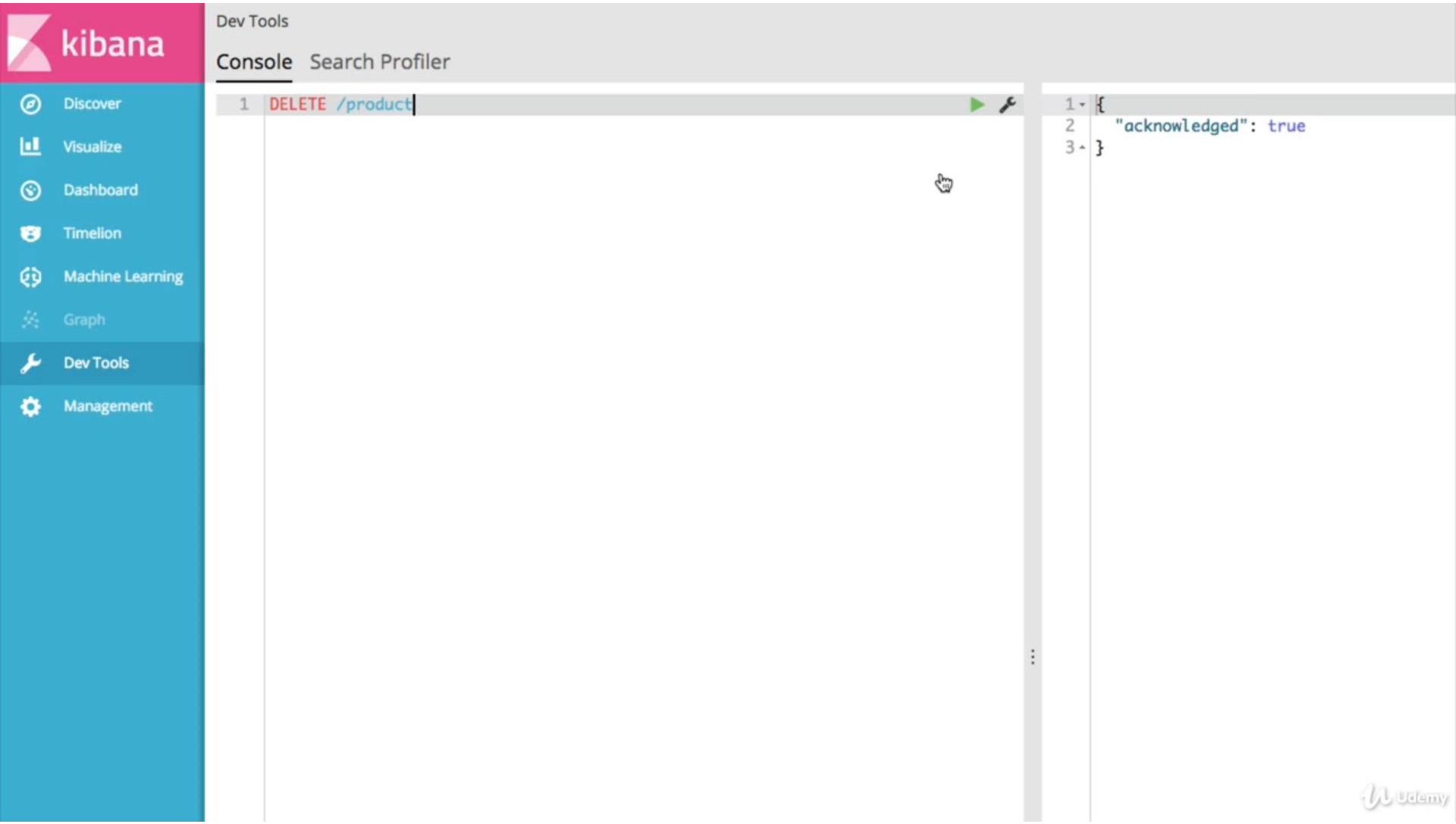
Managing Documents – Deleting Documents



Managing Documents – Deleting Documents



Managing Documents – Delete By Index



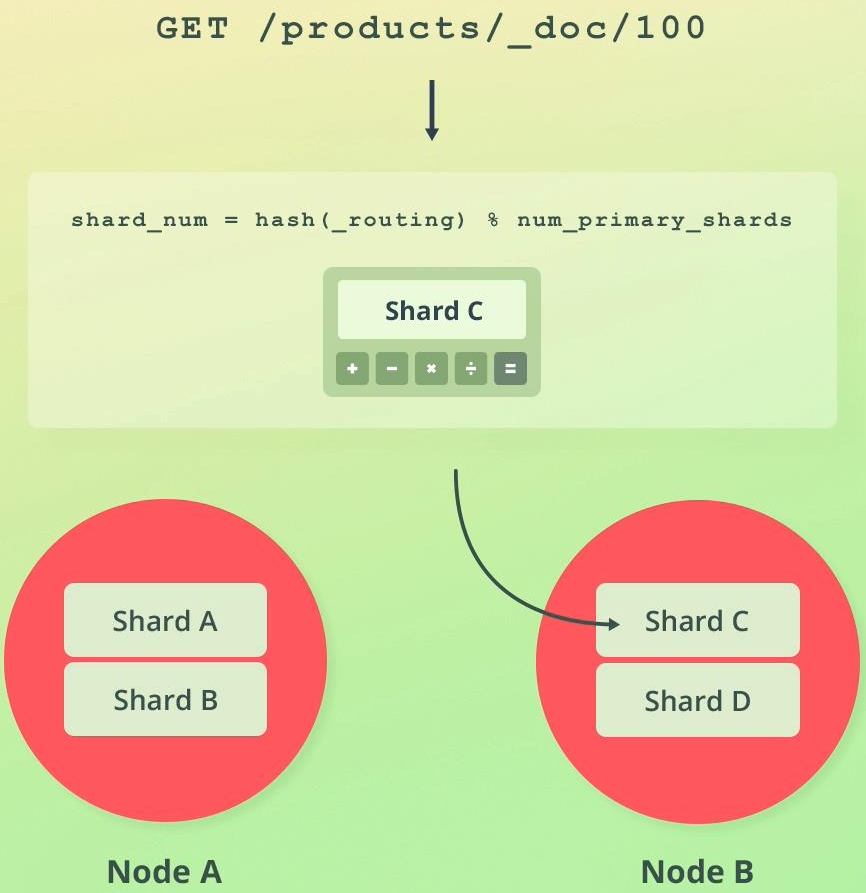
Introduction to Routing

- How does Elasticsearch know where to store documents?
- How are documents found once they have been indexed?
- The answer is *routing*
- Routing is the process of resolving a shard for a document

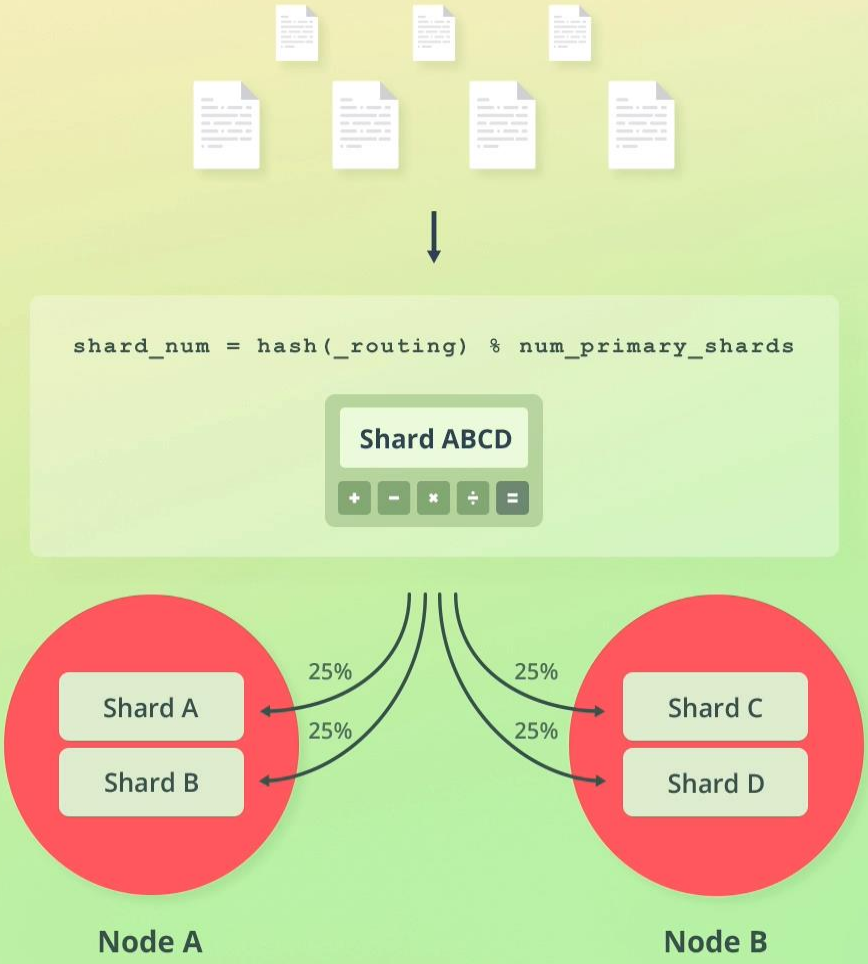
Introduction to Routing

```
shard_num = hash(_routing) % num_primary_shards
```

Introduction to Routing



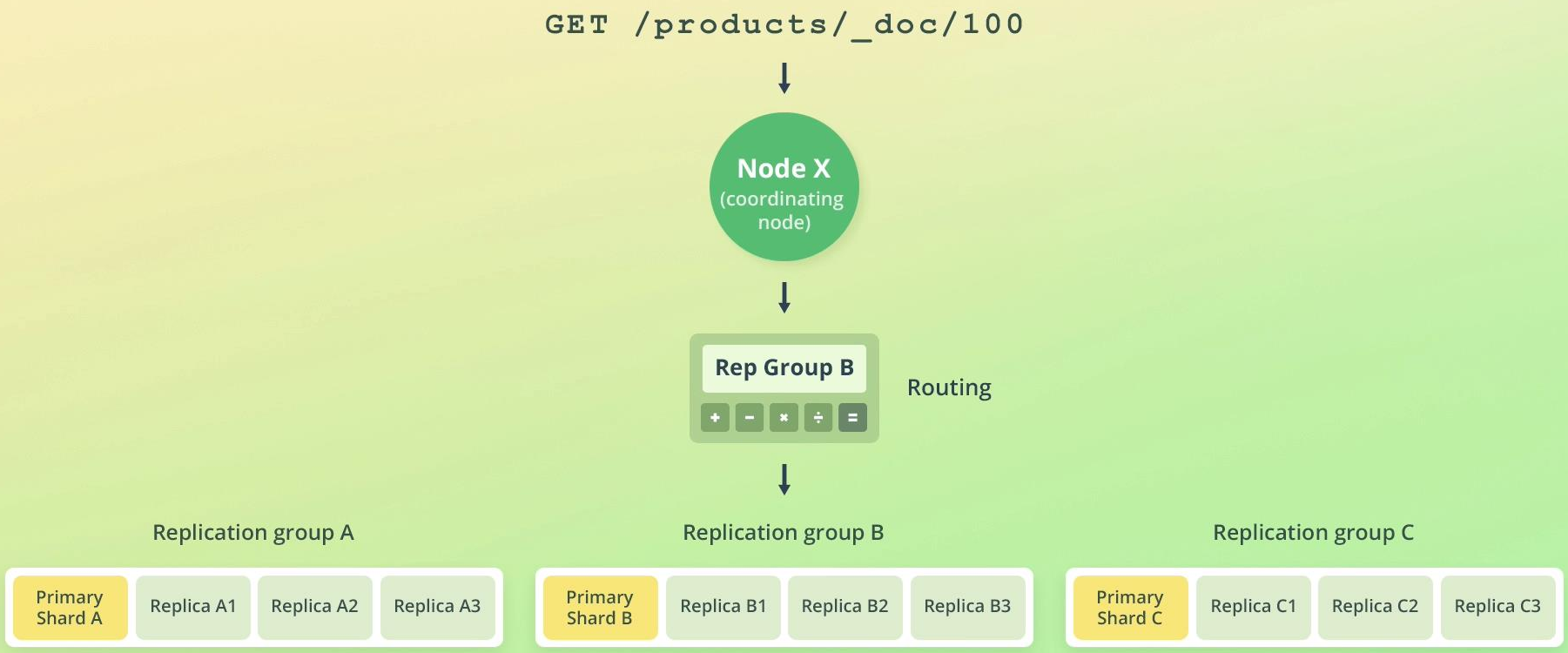
Introduction to Routing



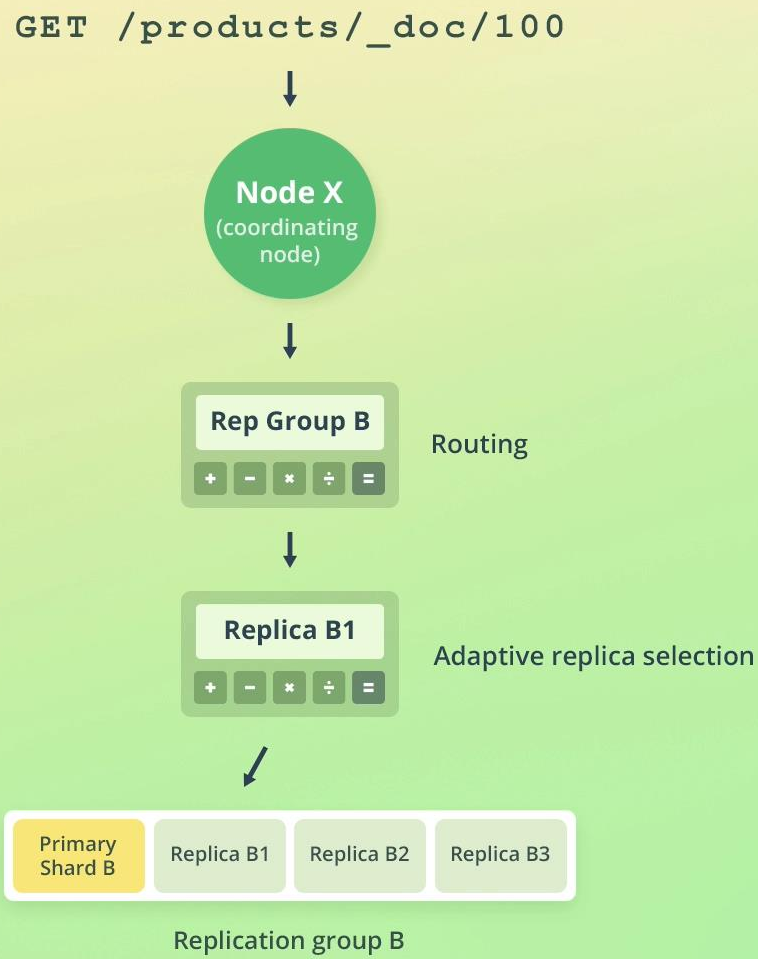
Introduction to Routing

Introduction to Routing

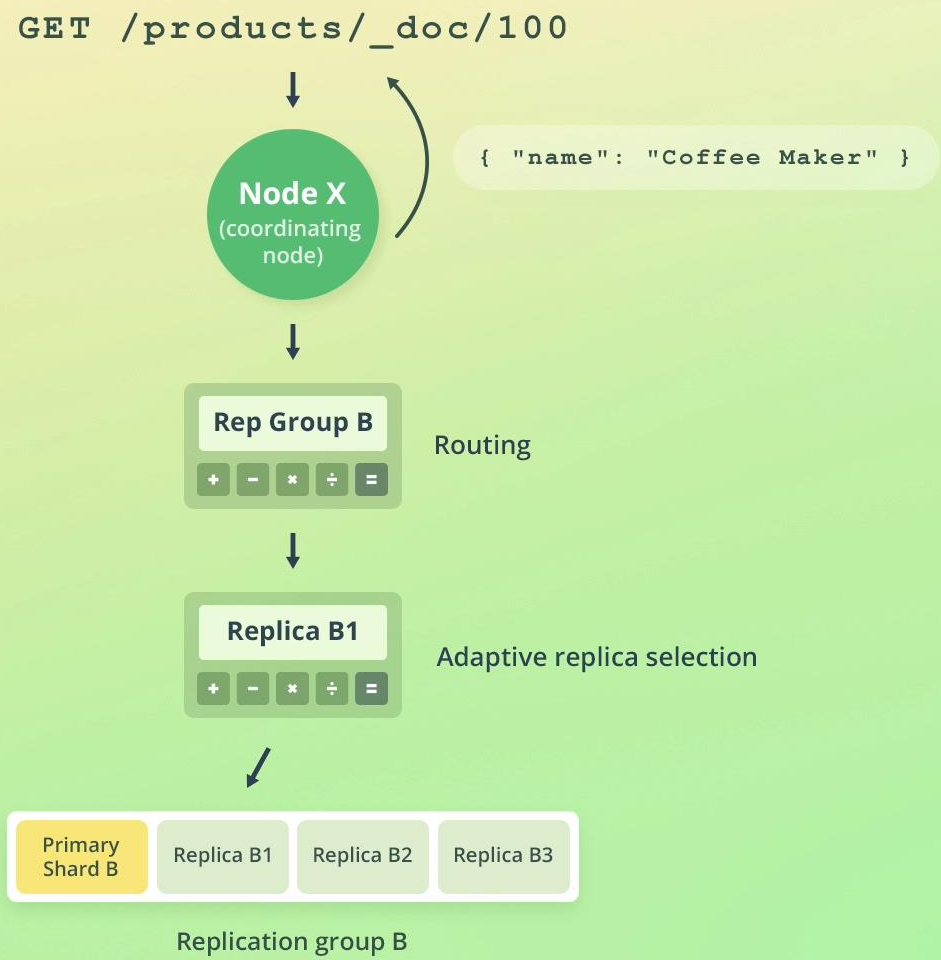
Introduction to Read Document



Introduction to Read Document



Introduction to Read Document

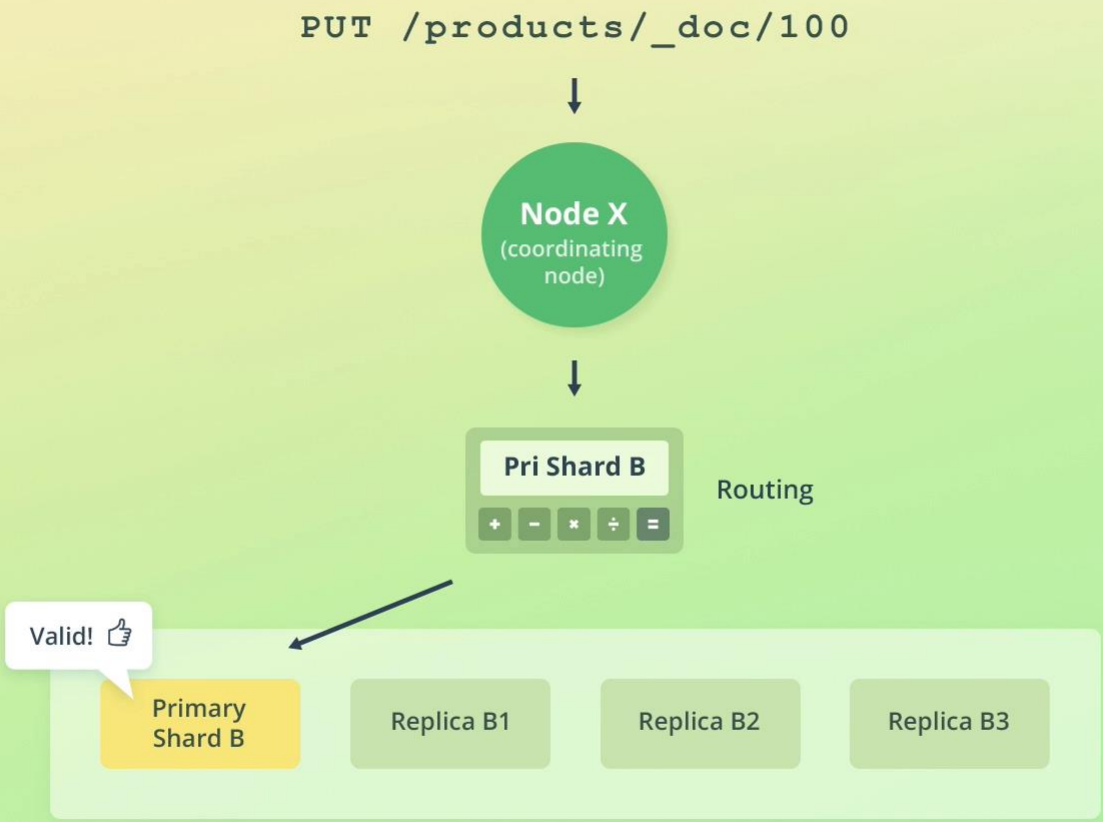


Introduction to Routing

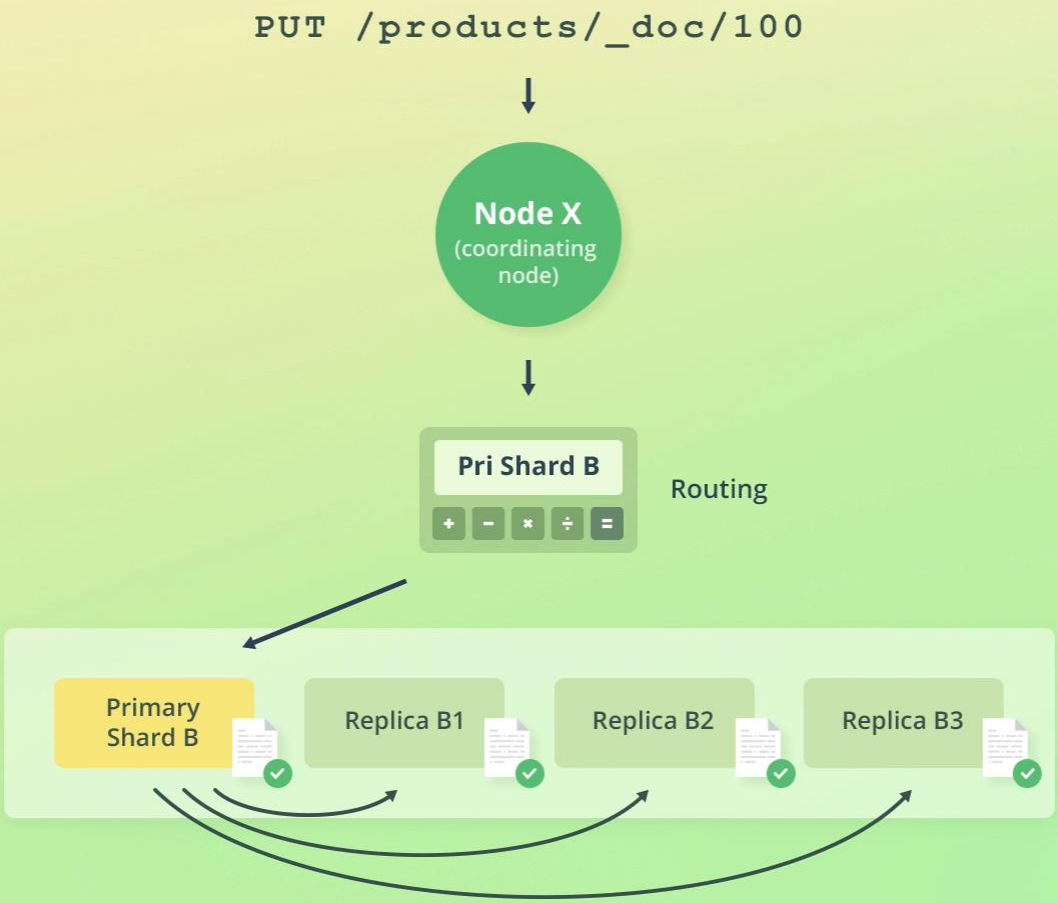
Lecture summary

- A read request is received and handled by a *coordinating node*
- Routing is used to resolve the document's replication group
- ARS is used to send the query to the best available shard
  - ARS is short for *Adaptive Replica Selection*
  - ARS helps reduce query response times
  - ARS is essentially an intelligent load balancer
- The coordinating node collects the response and sends it to the client

Writing Documents



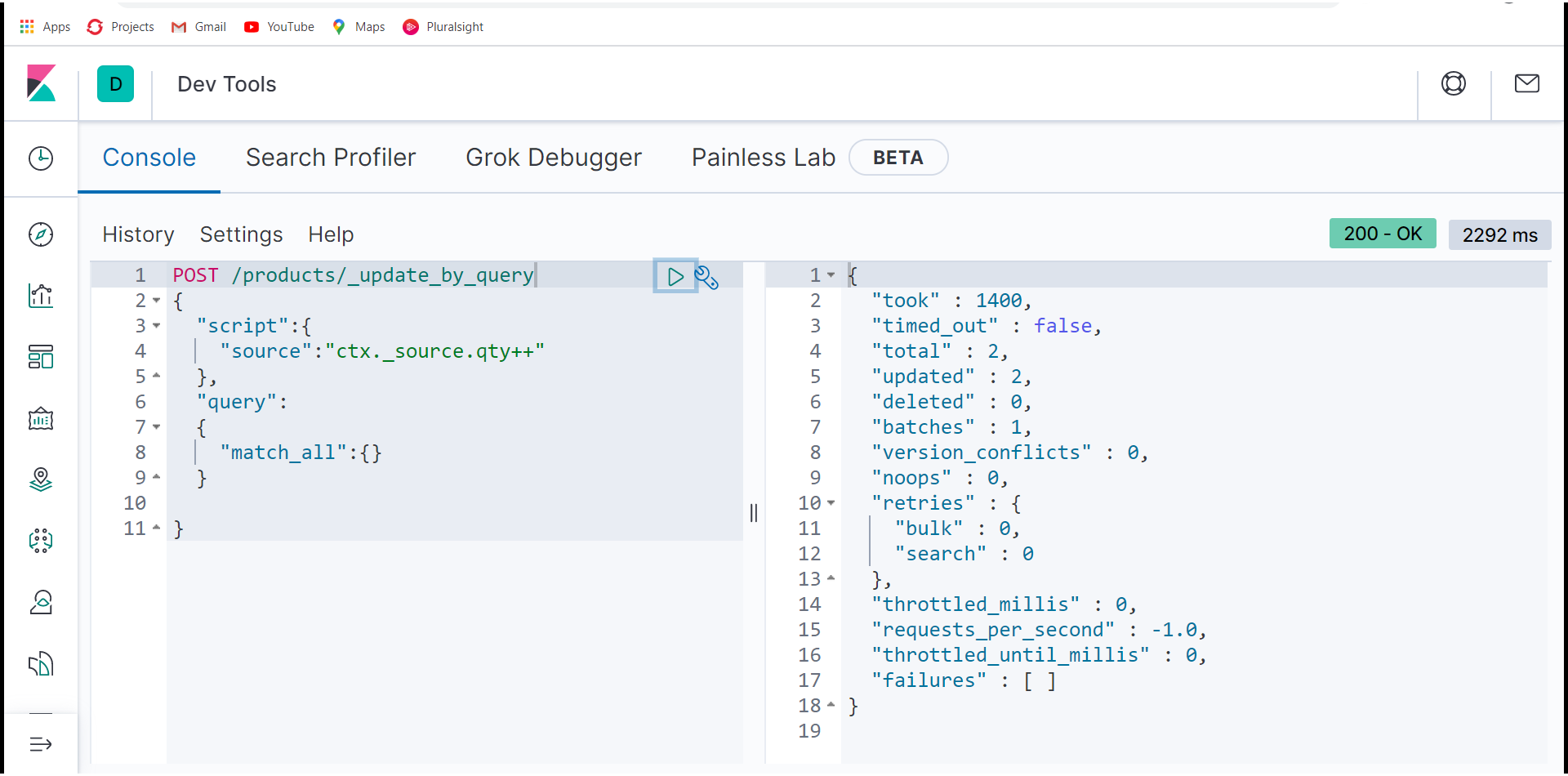
Writing Documents



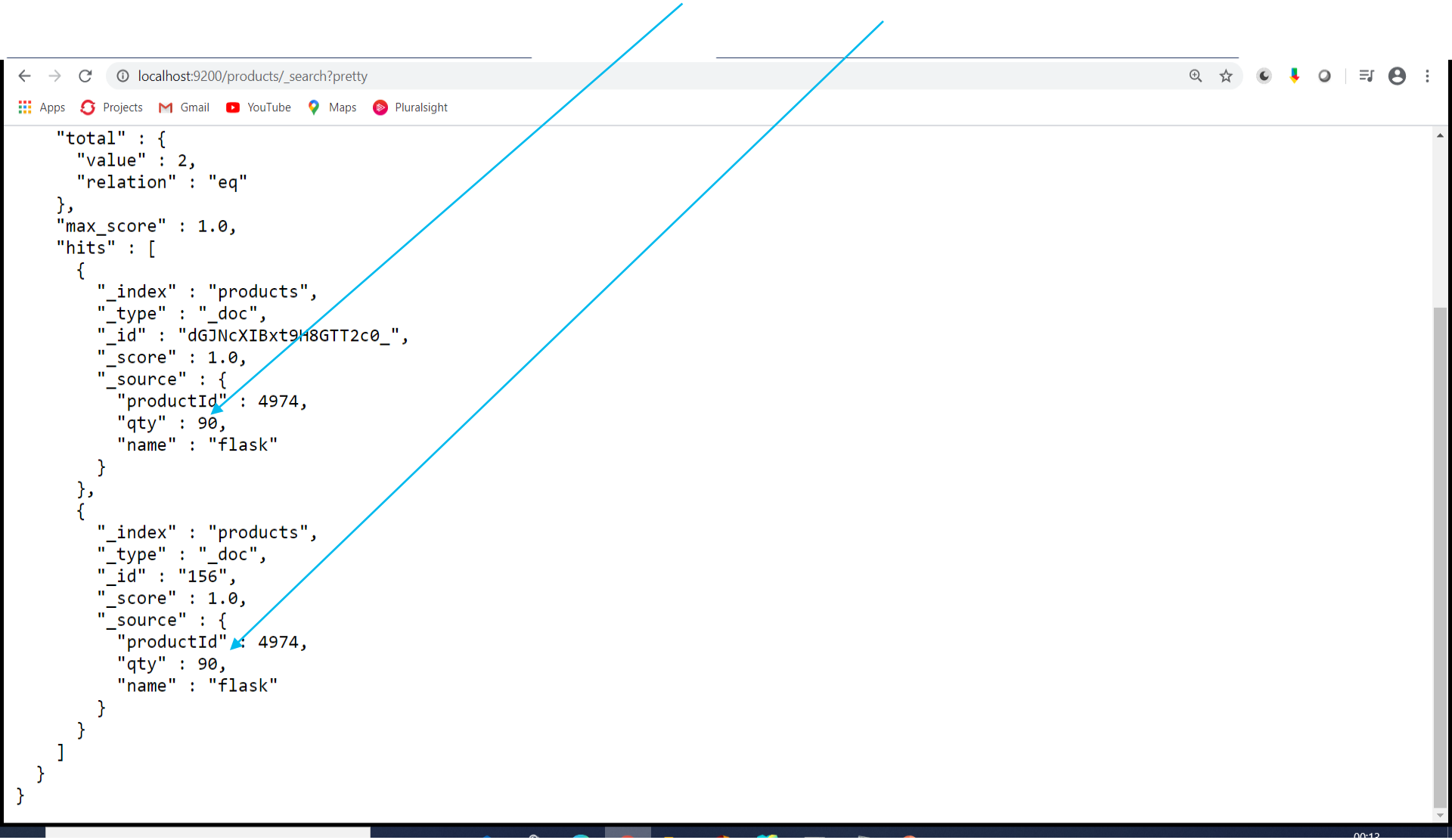
Multiple Update

- We already covered how to update *one* document at a time
- Let's now update *multiple* documents within a single query
  - Similar to an `UPDATE WHERE` query in a RDBMS
- The query uses three concepts that we have just covered
  - Primary terms
  - Sequence numbers
  - Optimistic concurrency control

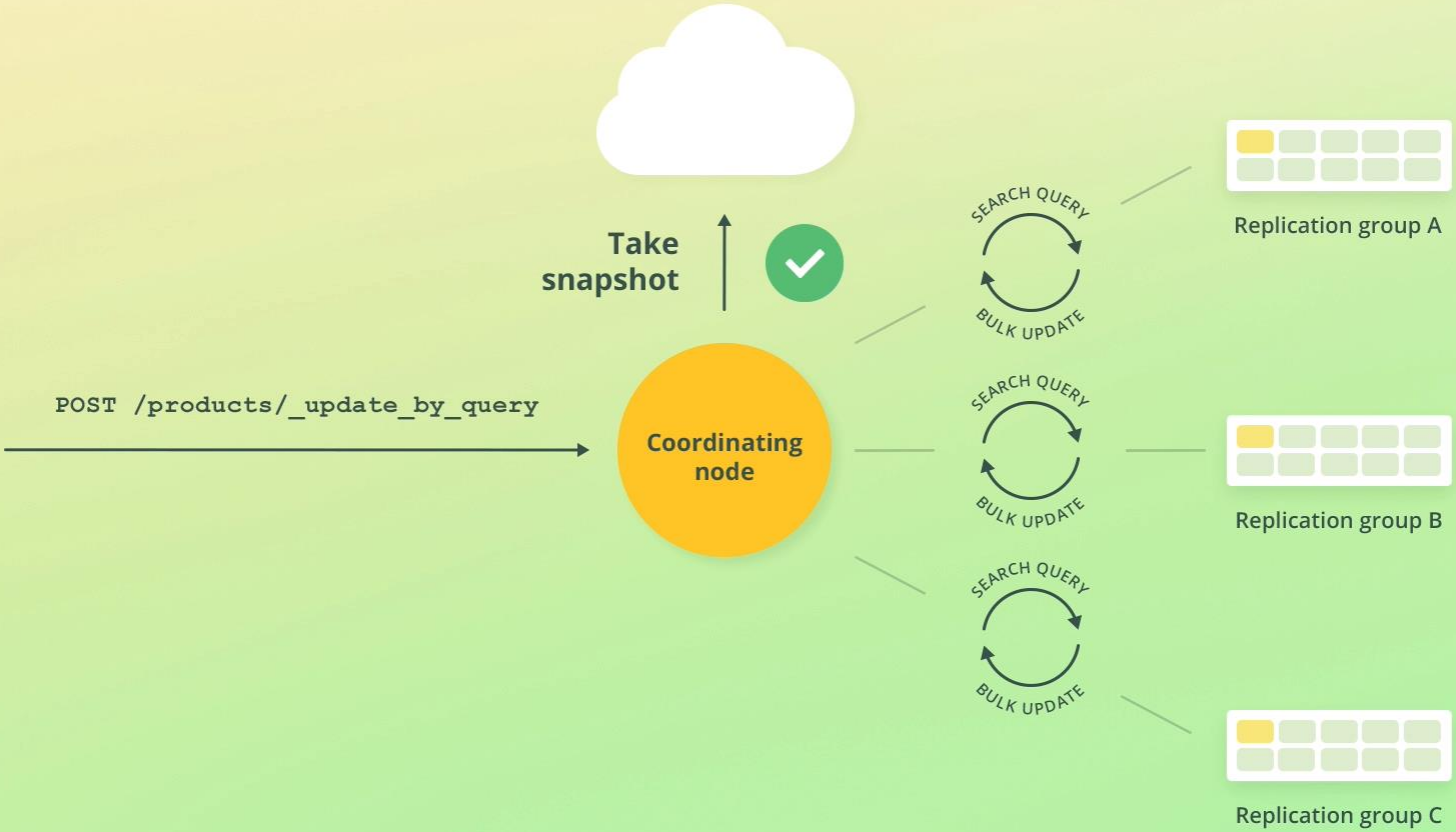
Multiple Update



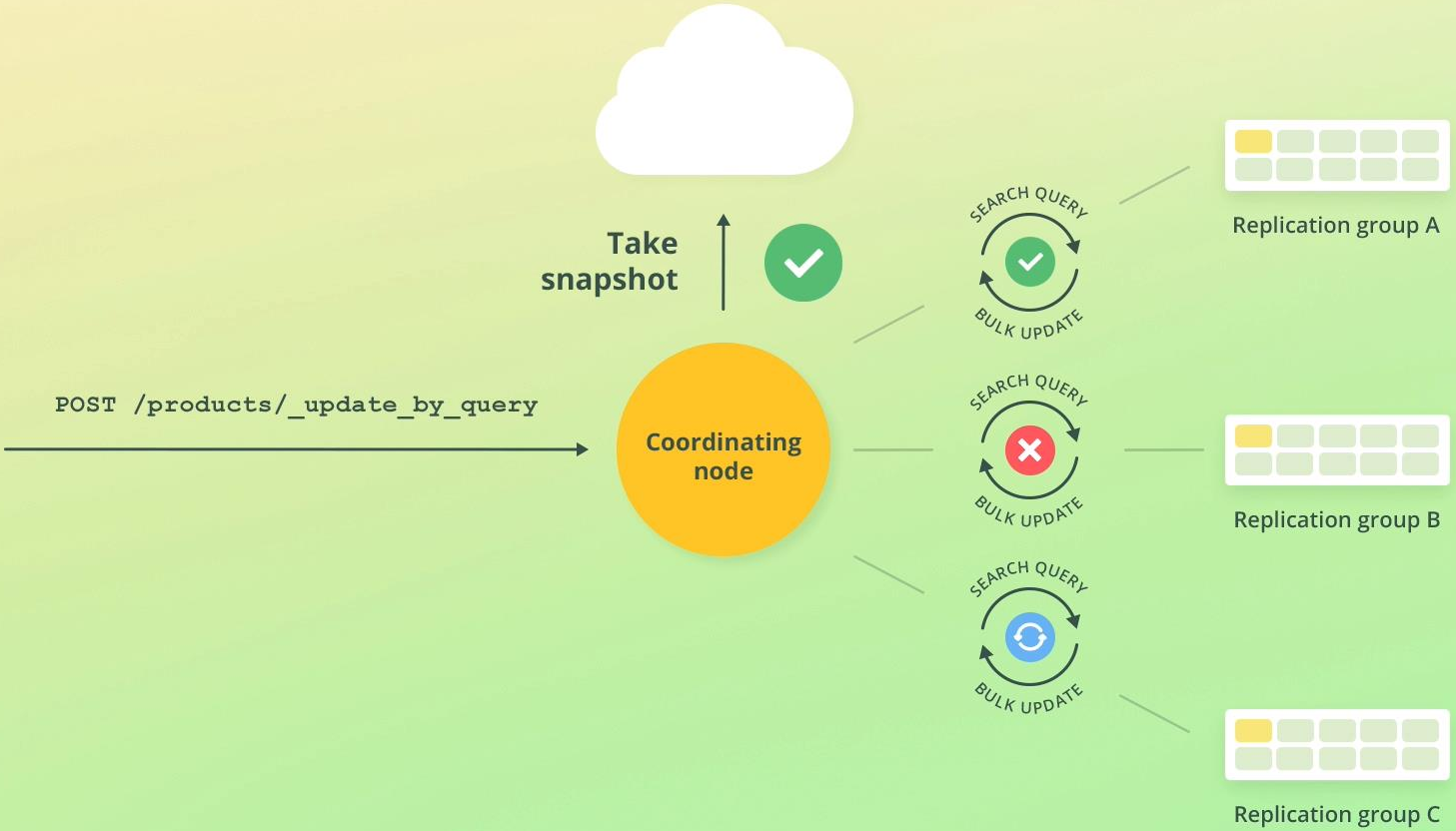
Multiple Update



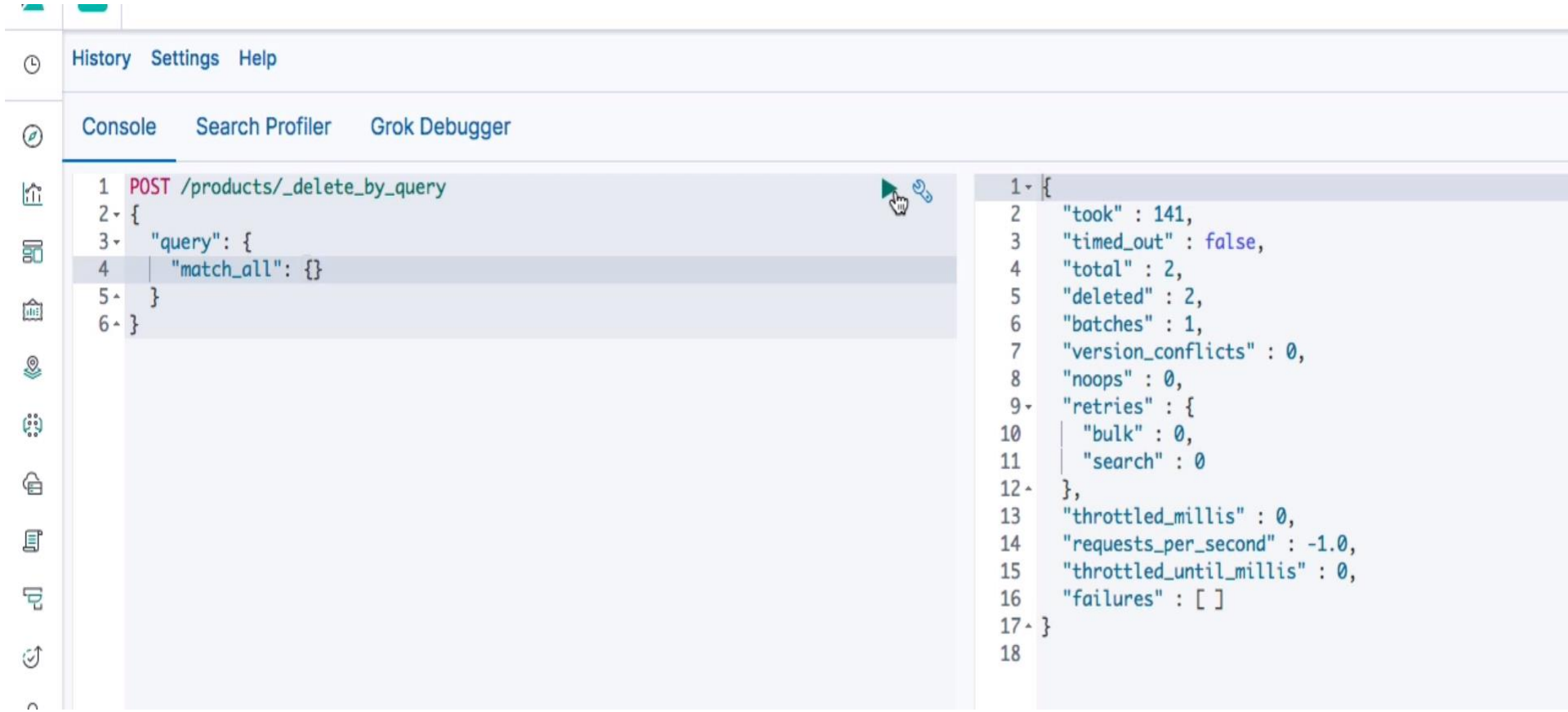
Multiple update(Scroll API)



Multiple update(Scroll API)



Delete Multiple Documents

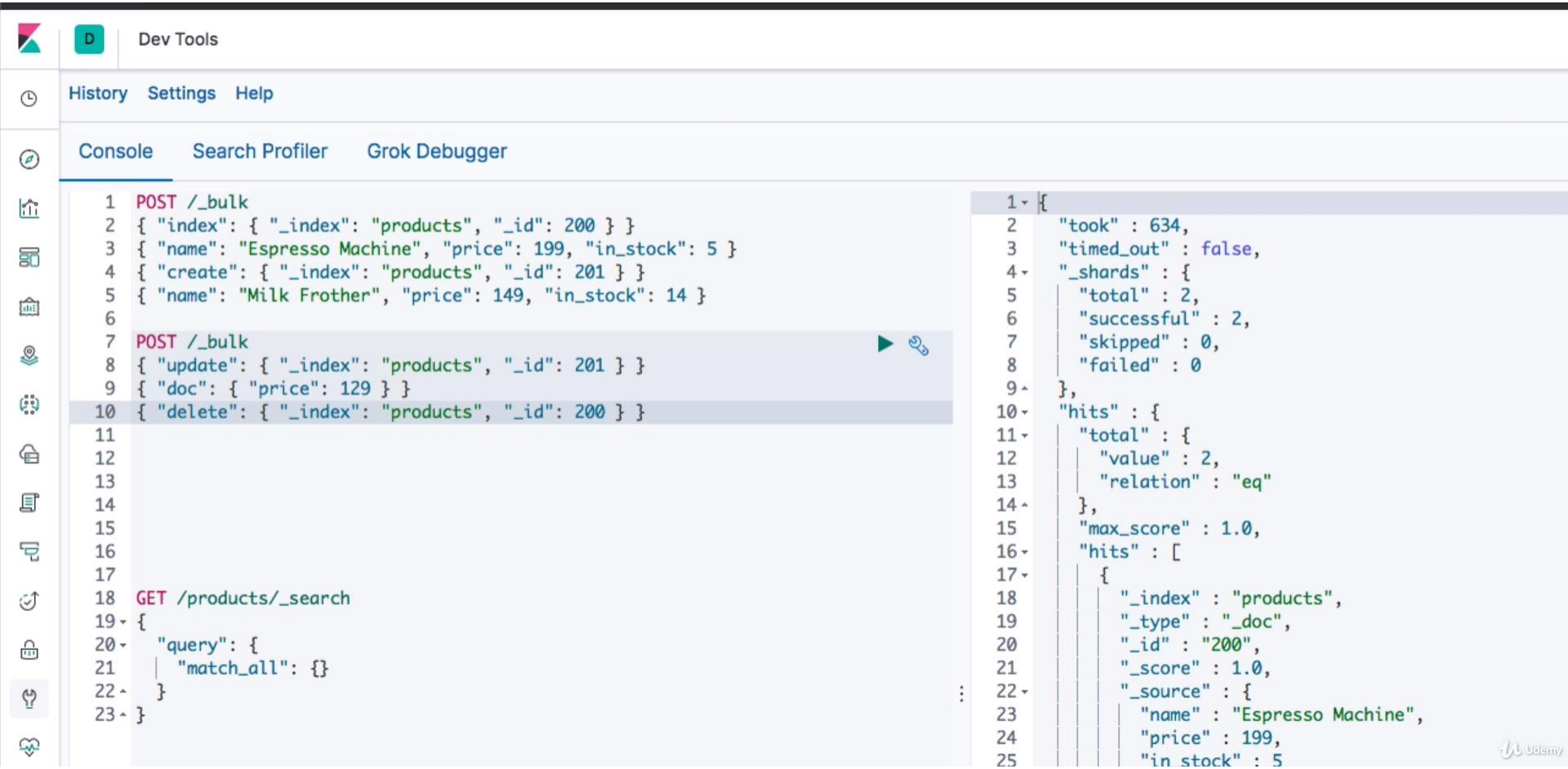


Bulk API

- You learned how to index, update, and delete documents
- Let's see how we can perform these actions on *many* documents with a *single* query
  - That's done with the Bulk API
- The Bulk API expects data formatted using the NDJSON specification

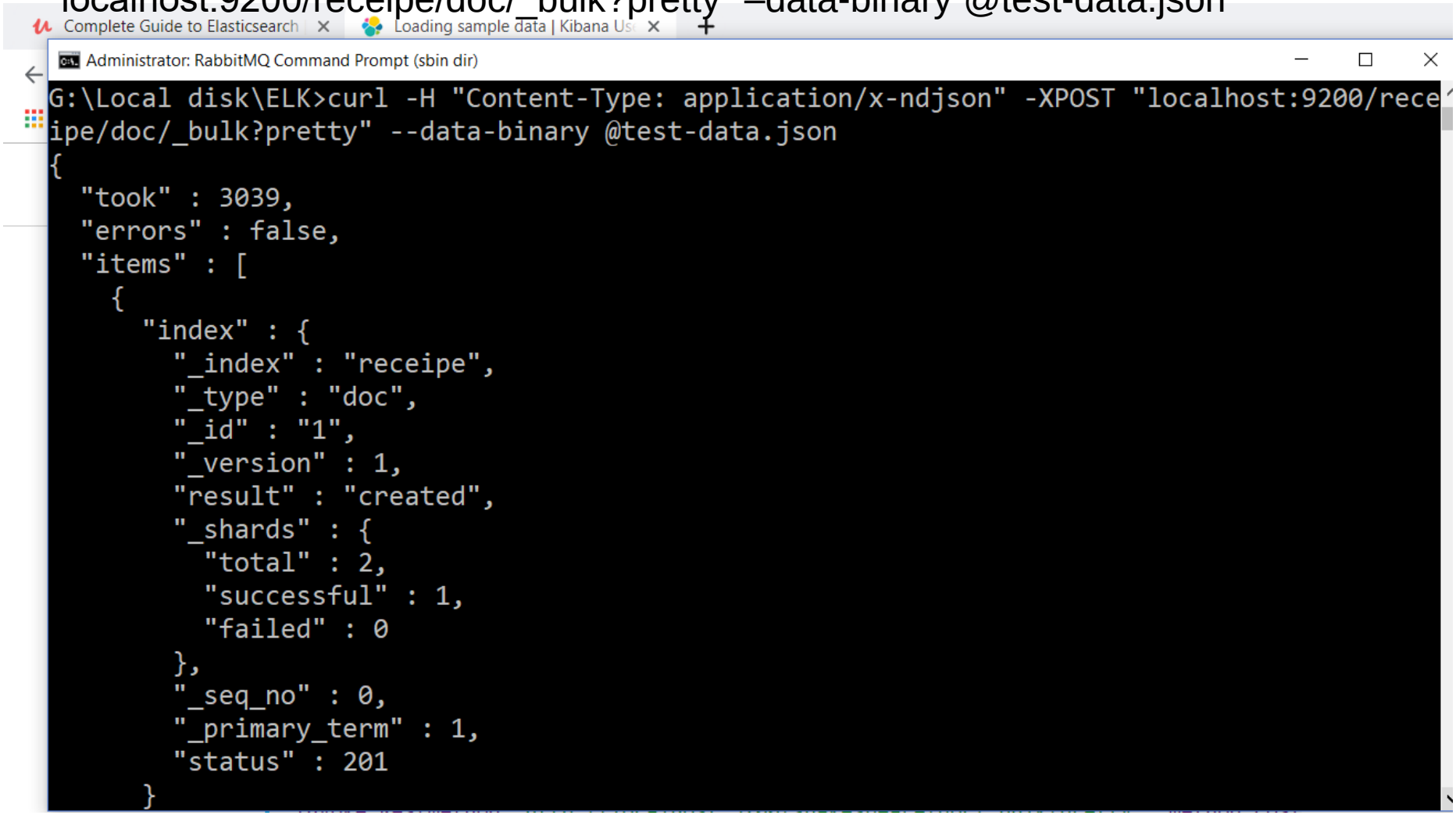
```
action_and_metadata\n
  optional_source\n
action_and_metadata\n
  optional_source\n
```

Bulk API


Dev Tools
History
Settings
Help
Console
Search Profiler
Grok Debugger
POST /_bulk
{ "index": { "_index": "products", "_id": 200 } }
{ "name": "Espresso Machine", "price": 199, "in_stock": 5 }
{ "create": { "_index": "products", "_id": 201 } }
{ "name": "Milk Frother", "price": 149, "in_stock": 14 }
POST /_bulk
{ "update": { "_index": "products", "_id": 201 } }
{ "doc": { "price": 129 } }
{ "delete": { "_index": "products", "_id": 200 } }
GET /products/_search
{
"query": {
"match_all": {}
}
}
{
"took" : 634,
"timed_out" : false,
"_shards" : {
"total" : 2,
"successful" : 2,
"skipped" : 0,
"failed" : 0
},
"hits" : {
"total" : {
"value" : 2,
"relation" : "eq"
},
"max_score" : 1.0,
"hits" : [
{
"_index" : "products",
"_type" : "_doc",
"_id" : "200",
"_score" : 1.0,
"_source" : {
"name" : "Espresso Machine",
"price" : 199,
"in_stock" : 5

Bulk Insert

Curl –H “Content-Type: application/x-ndjson” –XPOST “localhost:9200/receipe/doc/_bulk?pretty” –data-binary @test-data.json



Bulk API

New line data json -- ndjson

Things to be aware of (1/3)

- The HTTP `Content-Type` header should be set as follows
  - `Content-Type: application/x-ndjson`
  - `application/json` is accepted, but that's not the correct way
- The Console tool handles this for us
  - The Elasticsearch SDKs also handle this for us
  - Using HTTP clients, we need to handle this ourselves
- You will see how to do this in the next lecture

Bulk API

Things to be aware of (2/3)

- Each line **must** end with a newline character (`\n` or `\r\n`)
  - This **includes** the last line
    - In a text editor, this means that the last line should be empty
  - Automatically handled with the Console tool
  - Typically a script will generate the bulk file, in which case you need to handle this
  - Don't type out `\n` or `\r\n` in a text editor 🙂

Bulk API

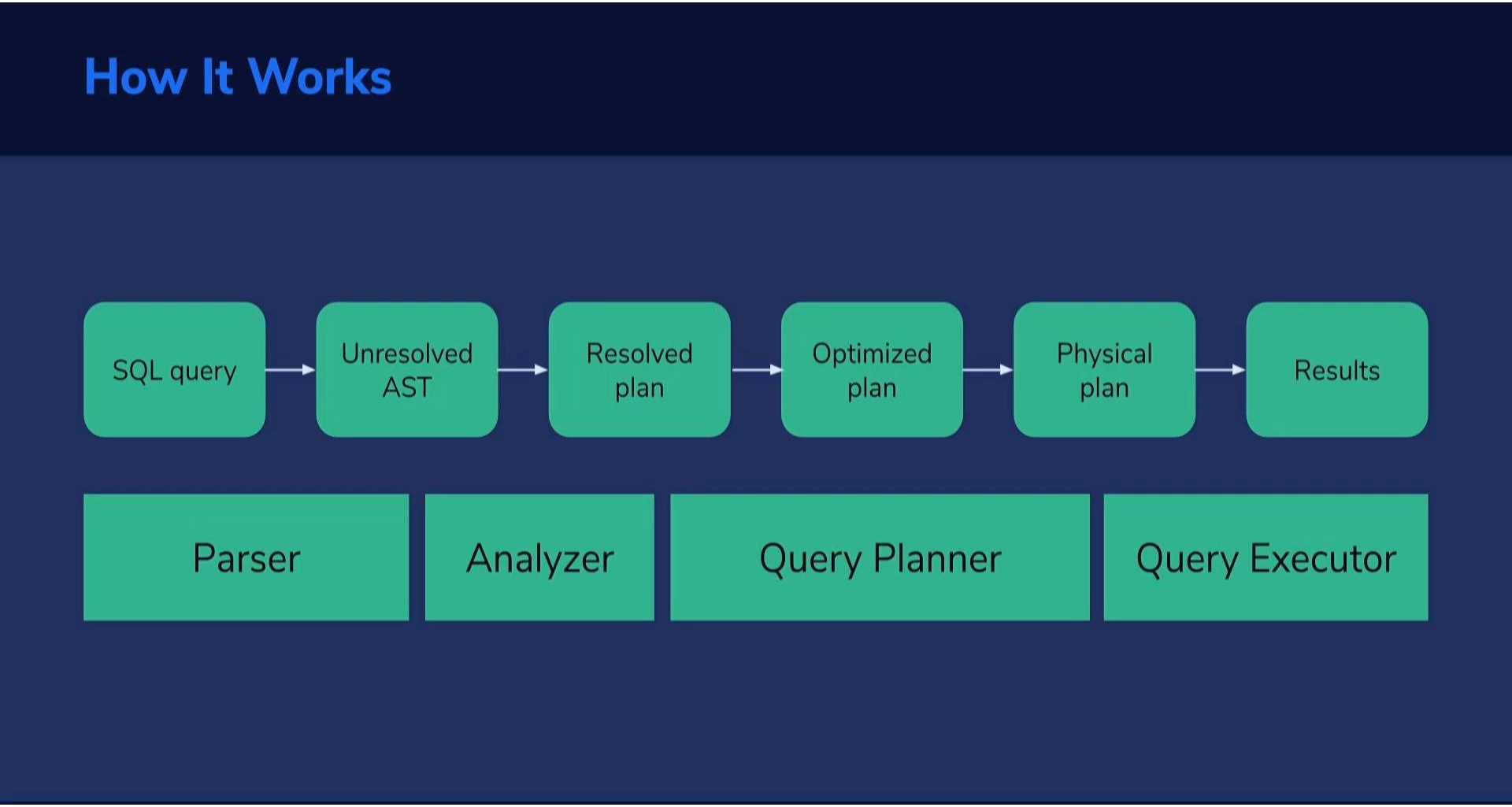
Things to be aware of (3/3)

- A failed action will **not** affect other actions
  - Neither will the bulk request as a whole be aborted
- The Bulk API returns detailed information about each action
  - Inspect the `items` key to see if a given action succeeded
    - The order is the same as the actions within the request
  - The `errors` key conveniently tells us if any errors occurred

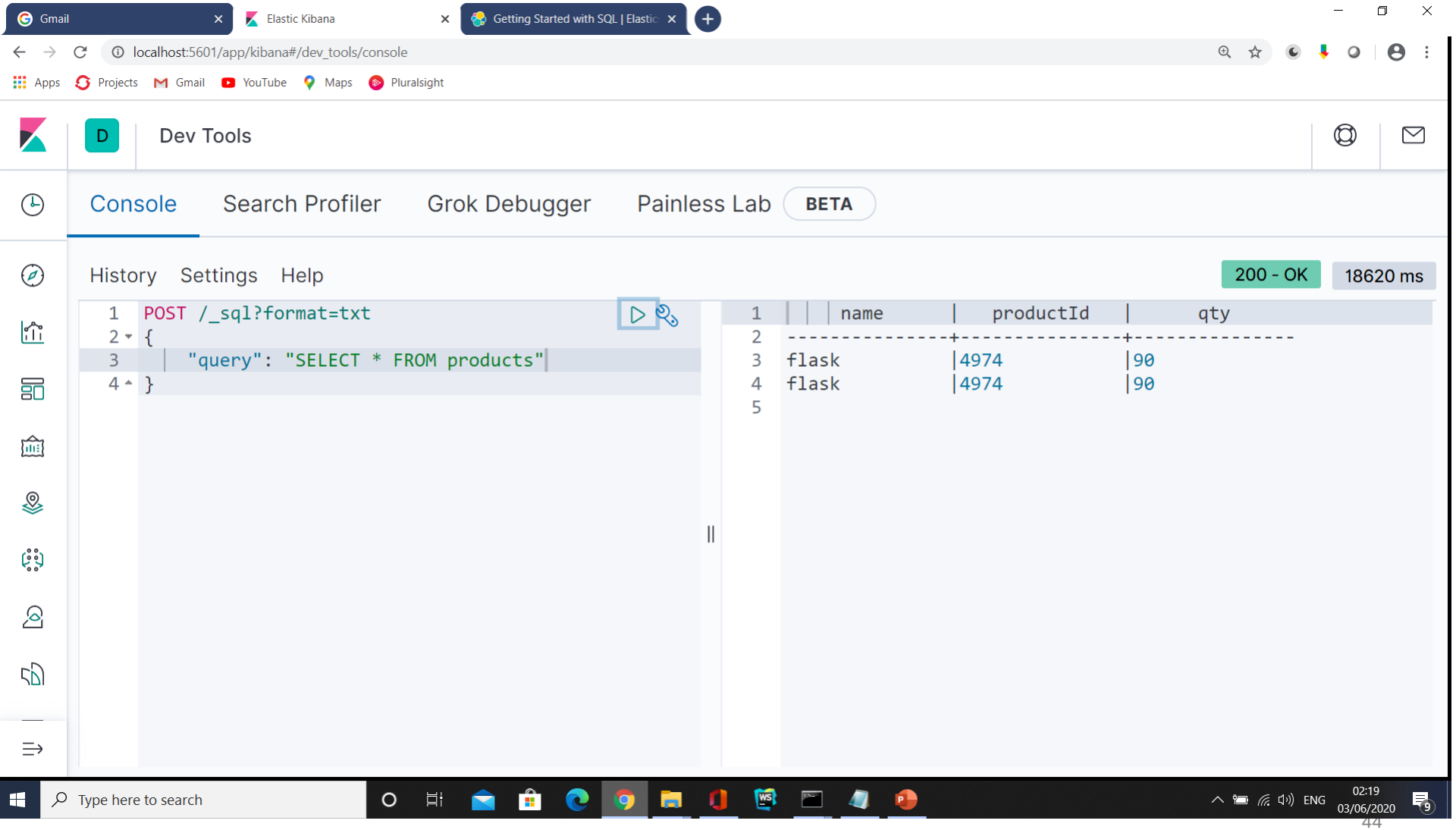
Why Bulk API

- When you need to perform lots of write operations at the same time
  - E.g. when importing data or modifying lots of data
- The Bulk API is more efficient than sending individual write requests
  - A lot of network round trips are avoided

Elastic Sql query



Elastic SQL Query



Elastic SQL Query

- POST /_sql?format=txt
- {
- "query": "SELECT * FROM library WHERE release_date < '2000-01-01’”
- }

Elastic SQL(elastic-sql-cli) from elastic bin

Administrator: Command Prompt - elasticsearch-sql-cli

```
                    asticElasticE
                  ElasticE  sticEla
         sticEl  ticEl           Elast
       lasti Elasti                tic
     cEl       ast                 icE
    icE        as                   cEl
    icE        as                   cEl
    icEla     las                    El
  sticElasticElast                  icElas
 las          last                 ticElast
El             asti               asti    stic
El             asticEla          Elas       icE
El           Elas  cElasticE   ticEl          cE
Ela       ticEl        ticElasti             cE
 las     astic             last             icE
   sticElas                 asti          stic
     icEl                   sticElasticElast
     icE                    sticE   ticEla
     icE                    sti       cEla
     icEl                   sti        Ela
      cEl                   sti       cEl
       Ela                 astic    ticE
         asti             ElasticElasti
           ticElasti  lasticElas
              ElasticElast

                    SQL
                   7.7.0

sql>
```

Bulk Insert

Curl –H “Content-Type: application/x-ndjson” –XPOST “localhost:9200/receipe/doc/_bulk?pretty” - -data-binary @test-data.json

Complete Guide to Elasticsearch | Loading sample data | Kibana Us

Administrator: RabbitMQ Command Prompt (sbin dir)

```
G:\Local disk\ELK>curl -H "Content-Type: application/x-ndjson" -XPOST "localhost:9200/rece
ipe/doc/_bulk?pretty" --data-binary @test-data.json
{
  "took" : 3039,
  "errors" : false,
  "items" : [
    {
      "index" : {
        "_index" : "receipe",
        "_type" : "doc",
        "_id" : "1",
        "_version" : 1,
        "result" : "created",
        "_shards" : {
          "total" : 2,
          "successful" : 1,
          "failed" : 0
        },
        "_seq_no" : 0,
        "_primary_term" : 1,
        "status" : 201
      }
```

Mapping

```
CREATE TABLE employees (
  id INT AUTO_INCREMENT PRIMARY KEY,
  first_name VARCHAR(255) NOT NULL,
  last_name VARCHAR(255) NOT NULL,
  dob DATE,
  description TEXT,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

MySQL

```
PUT /employees
{
  "mappings": {
    "properties": {
      "id": { "type": "integer" },
      "first_name": { "type": "text" },
      "last_name": { "type": "text" },
      "dob": { "type": "date" },
      "description": { "type": "text" },
      "created_at": { "type": "date" }
    }
  }
}
```

Elasticsearch

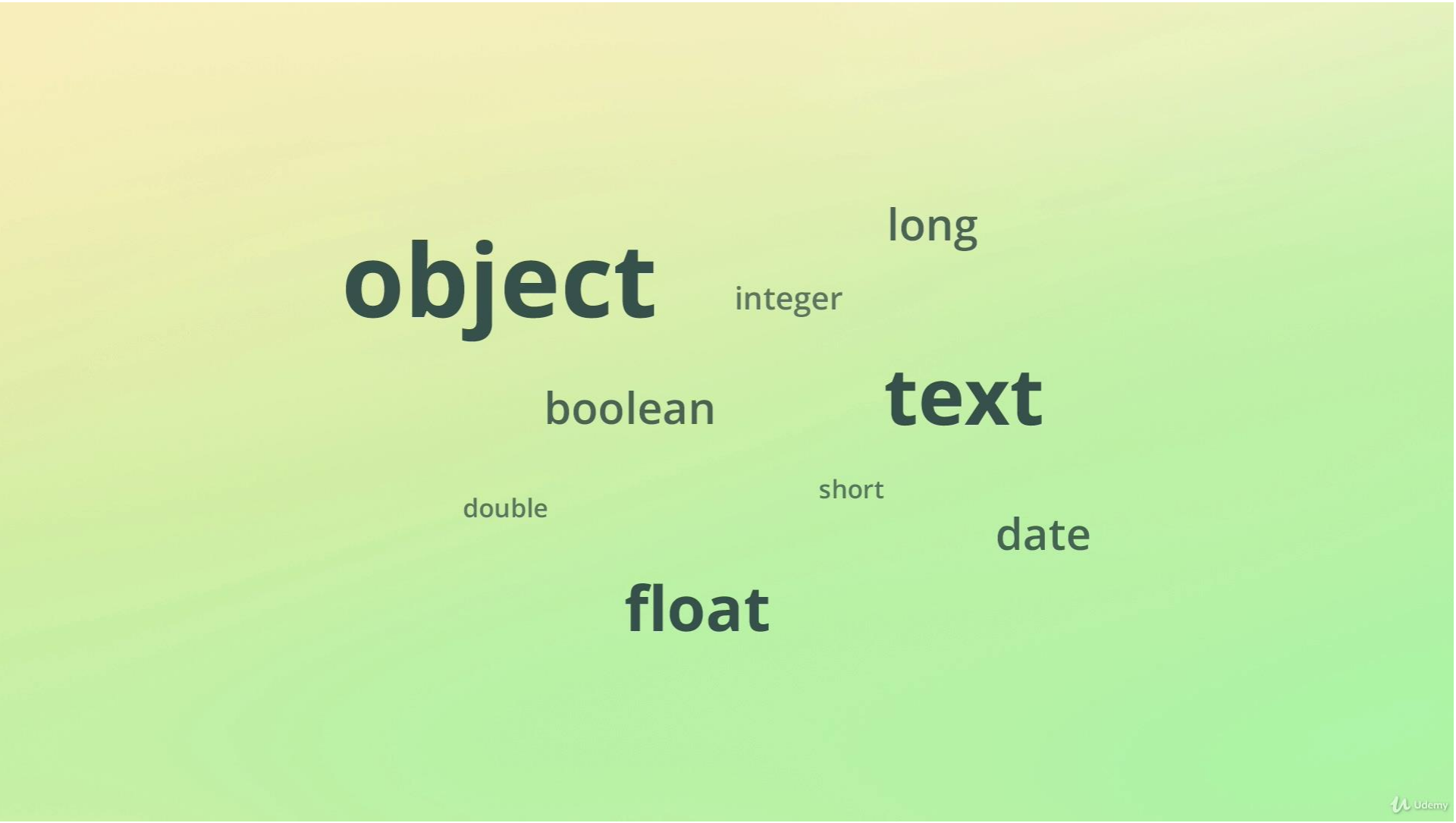
What is Mapping

- Defines the structure of documents (e.g. fields and their data types)
  - Also used to configure how values are indexed
- Similar to a table's schema in a relational database
- Explicit mapping
  - We define field mappings ourselves
- Dynamic mapping
  - Elasticsearch generates field mappings for us

Keyword Data Type

- Used for exact matching of values
- Typically used for filtering, aggregations, and sorting
- E.g. searching for articles with a status of `PUBLISHED`
- For full-text searches, use the `text` data type instead
    - E.g. searching the body text of an article

Keyword Data Type



Data Types

- Core datatypes
- string
- text and keyword
- Numeric
- long, integer, short, byte, double, float, half_float, scaled_float
- Date
- date
- Date nanoseconds

Data Types

- date_nanos
- Boolean
- boolean
- Binary
- binary
- Range
- integer_range, float_range, long_range, double_range, date_range, ip_range

Data Types

- Complex datatypes
- Object
- object for single JSON objects
- Nested
- nested for arrays of JSON objects
- Geo datatypes
- Geo-point
- geo_point for lat/lon points
- Geo-shape
- geo_shape for complex shapes like polygons

Data Types

- Specialised datatypese
- IP
- ip for IPv4 and IPv6 addresses
- Completion datatype
- completion to provide auto-complete suggestions
- Token count
- token_count to count the number of tokens in a string
- mapper-murmur3

Data Types

- Specialised datatypese
- IP
- ip for IPv4 and IPv6 addresses
- Completion datatype
- completion to provide auto-complete suggestions
- Token count
- token_count to count the number of tokens in a string
- mapper-murmur3

Data Types

- Join
- Defines parent/child relation for documents within the same index
- Rank feature
- Record numeric feature to boost hits at query time.
- Rank features
- Record numeric features to boost hits at query time.
- Dense vector
- Record dense vectors of float values.
- Sparse vector
- Record sparse vectors of float values.
- Search-as-you-type

Data Types

- A text-like field optimized for queries to implement as-you-type completion
- Alias
- Defines an alias to an existing field.
- Flattened
- Allows an entire JSON object to be indexed as a single field.
- Shape
- shape for arbitrary cartesian geometries.
- Histogram
- histogram for pre-aggregated numerical values for percentiles aggregations.
- Constant keyword
- Specialization of keyword for the case when all documents have the same value.
- Arrays

Meta Fields

- Meta fields customize how a document's associated metadata is treated.
- Each document has associated metadata such as the _index, mapping _type, and _id meta-fields.
- The behavior of some of these meta-fields could be custom when a mapping type was created.

Meta Fields

- Identity meta-fields
  - _index: The index to which the document belongs.
  - _uid: A composite field consisting of the _type and the _id.
  - _type: The document's mapping type.
  - _id: The document's ID.

- Document source meta-fields
  - _source: The original JSON representing the body of the document.
  - _size:The size of the _source field in bytes, provided by the mapper-size plugin.

Meta Fields

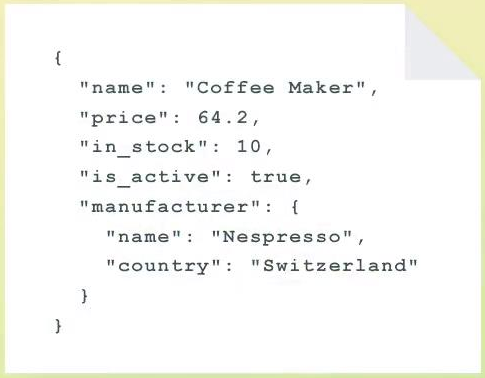
- Indexing meta-fields
  - _all: A catch-all field that indexes the values of all other fields.
  - _field_names: All fields in the document that contain non-null values.
  - _timestamp: A timestamp associated with the document, either specified manually or auto-generated.
  - _ttl: How long a document should live before it is automatically deleted.
- Routing meta-fields
  - _parent: Used to create a parent-child relationship between two mapping types.
  - _routing: A custom routing value that routes a document to a particular shard.

Mapping

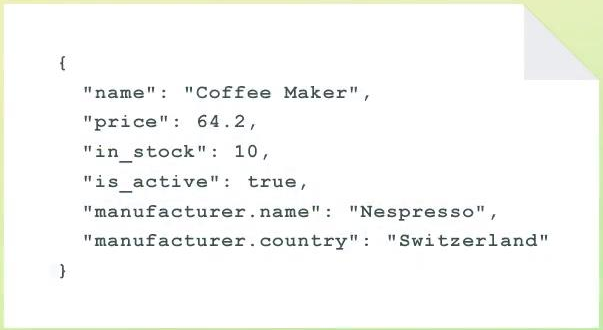
```
{
  "name": "Coffee Maker",
  "price": 64.2,
  "in_stock": 10,
  "is_active": true,
  "manufacturer": {
    "name": "Nespresso",
    "country": "Switzerland",
    "owned_by": {
      "name": "Nestlé Group",
      "country": "Switzerland"
    }
  }
}
```

```
PUT /products
{
  "mappings": {
    "properties": {
      ...
      "manufacturer": {
        "properties": {
          "name": { "type": "text" },
          "country": { "type": "text" },
          "owned_by": {
            "properties": {
              "name": { "type": "text" },
              "country": { "type": "text" }
            }
          }
        }
      }
    }
  }
}
```

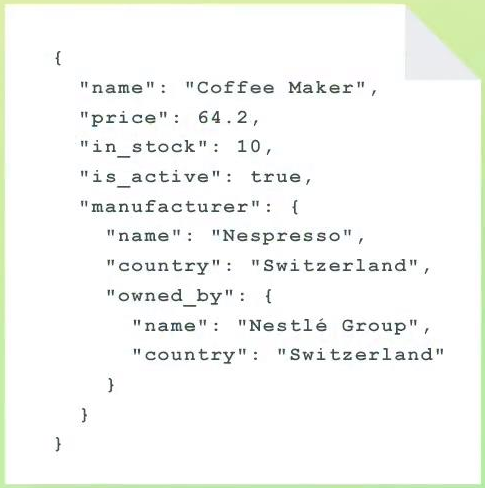
Mapping



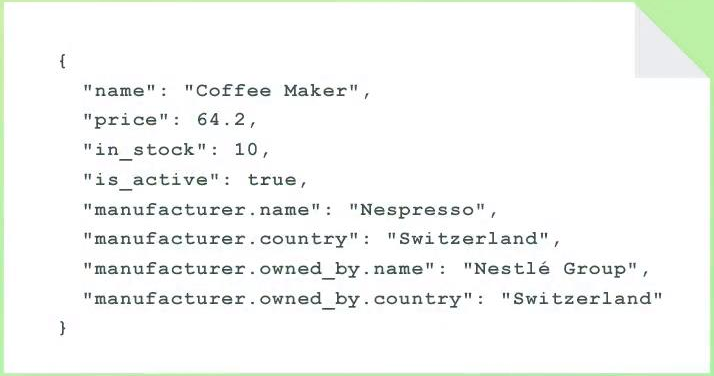
```
{
  "name": "Coffee Maker",
  "price": 64.2,
  "in_stock": 10,
  "is_active": true,
  "manufacturer": {
    "name": "Nespresso",
    "country": "Switzerland"
  }
}
```



```
{
  "name": "Coffee Maker",
  "price": 64.2,
  "in_stock": 10,
  "is_active": true,
  "manufacturer.name": "Nespresso",
  "manufacturer.country": "Switzerland"
}
```



```
{
  "name": "Coffee Maker",
  "price": 64.2,
  "in_stock": 10,
  "is_active": true,
  "manufacturer": {
    "name": "Nespresso",
    "country": "Switzerland",
    "owned_by": {
      "name": "Nestlé Group",
      "country": "Switzerland"
    }
  }
}
```

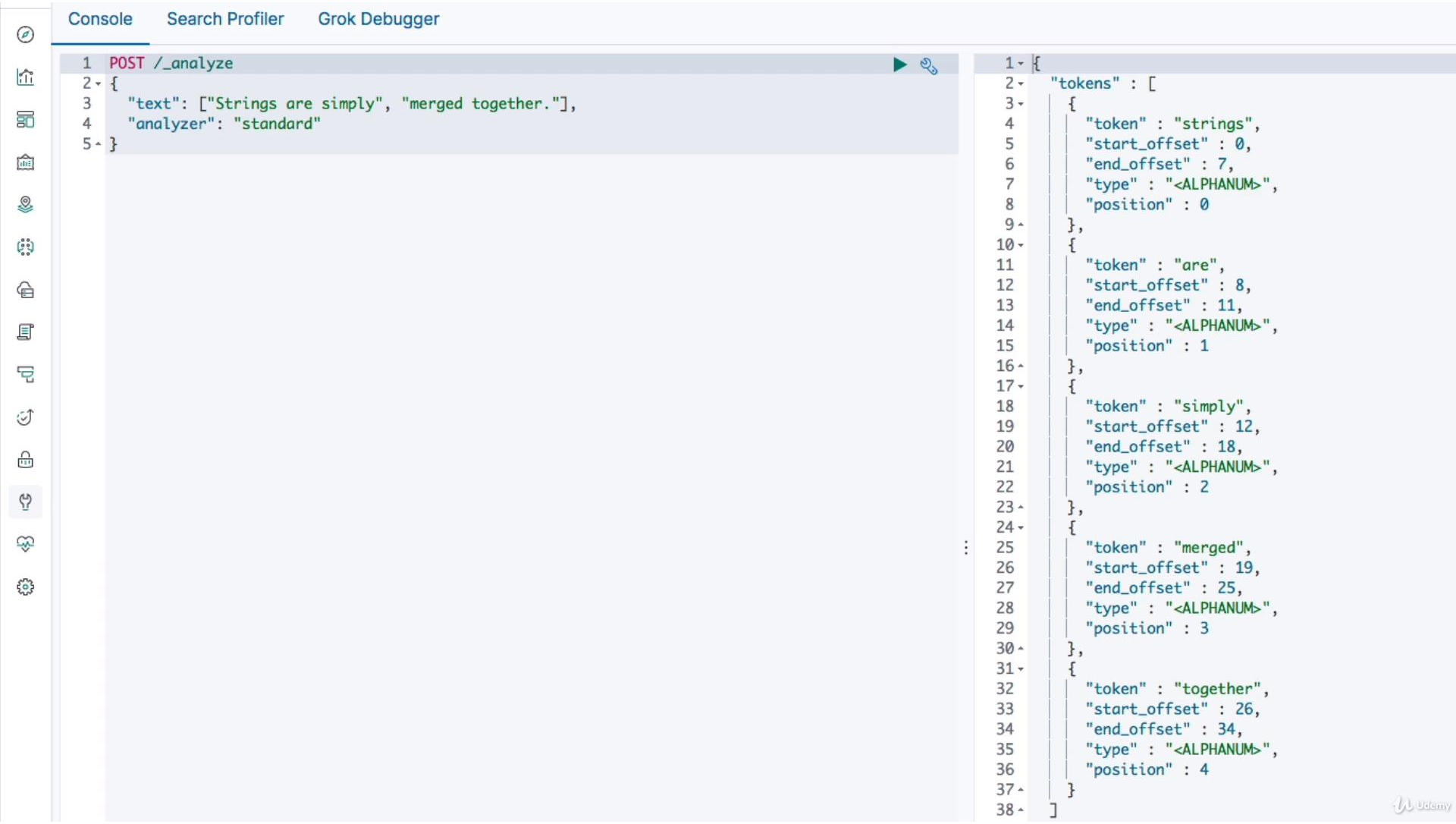


```
{
  "name": "Coffee Maker",
  "price": 64.2,
  "in_stock": 10,
  "is_active": true,
  "manufacturer.name": "Nespresso",
  "manufacturer.country": "Switzerland",
  "manufacturer.owned_by.name": "Nestlé Group",
  "manufacturer.owned_by.country": "Switzerland"
}
```

Introduction to Arrays

- There is no such thing as an `array` data type
- Any field may contain zero or more values
  - No configuration or mapping needed
  - Simply supply an array when indexing a document
- We did this for the `tags` field for the `products` index

Arrays


Console
Search Profiler
Grok Debugger
POST /_analyze
{
"text": ["Strings are simply", "merged together."],
"analyzer": "standard"
}
{
"tokens" : [
{
"token" : "strings",
"start_offset" : 0,
"end_offset" : 7,
"type" : "<ALPHANUM>",
"position" : 0
},
{
"token" : "are",
"start_offset" : 8,
"end_offset" : 11,
"type" : "<ALPHANUM>",
"position" : 1
},
{
"token" : "simply",
"start_offset" : 12,
"end_offset" : 18,
"type" : "<ALPHANUM>",
"position" : 2
},
{
"token" : "merged",
"start_offset" : 19,
"end_offset" : 25,
"type" : "<ALPHANUM>",
"position" : 3
},
{
"token" : "together",
"start_offset" : 26,
"end_offset" : 34,
"type" : "<ALPHANUM>",
"position" : 4
}
]

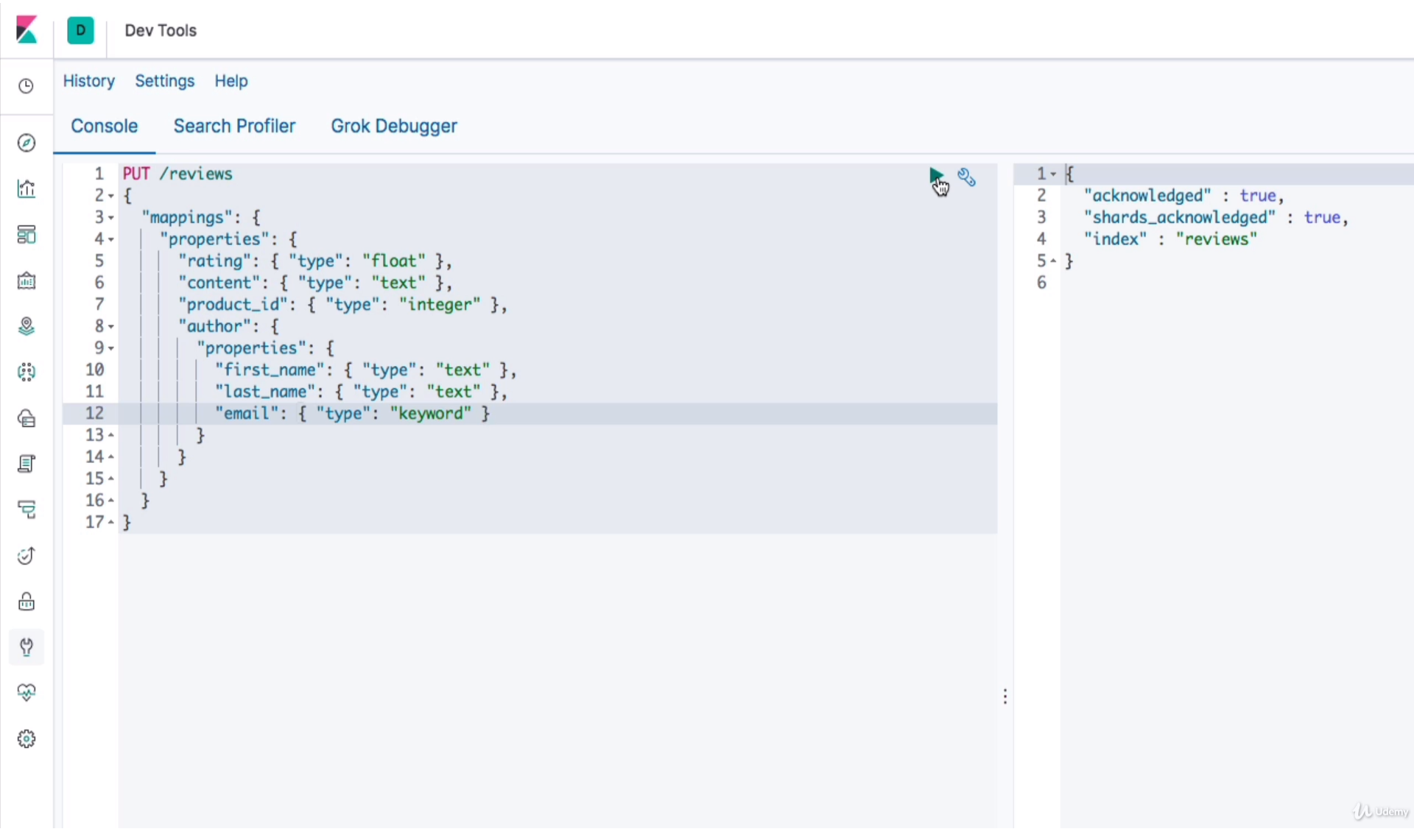
Arrays

```
// Correct data types
[ "electronics", "expensive", "popular" ]
[ 37, 45, 9 ]
[ true, false, true ]
[ { "name": "Coffee Maker" }, { "name": "Toaster" }, { "name": "Blender" } ]

// Coercion
[ true, false, "true" ]
[ "electronics", "expensive", 47 ]
[ 37, 45, "9" ]
[ true, false, "true" ]

// Cannot coerce
[ { "name": "Coffee Maker" }, { "name": "Toaster" }, false ]
```

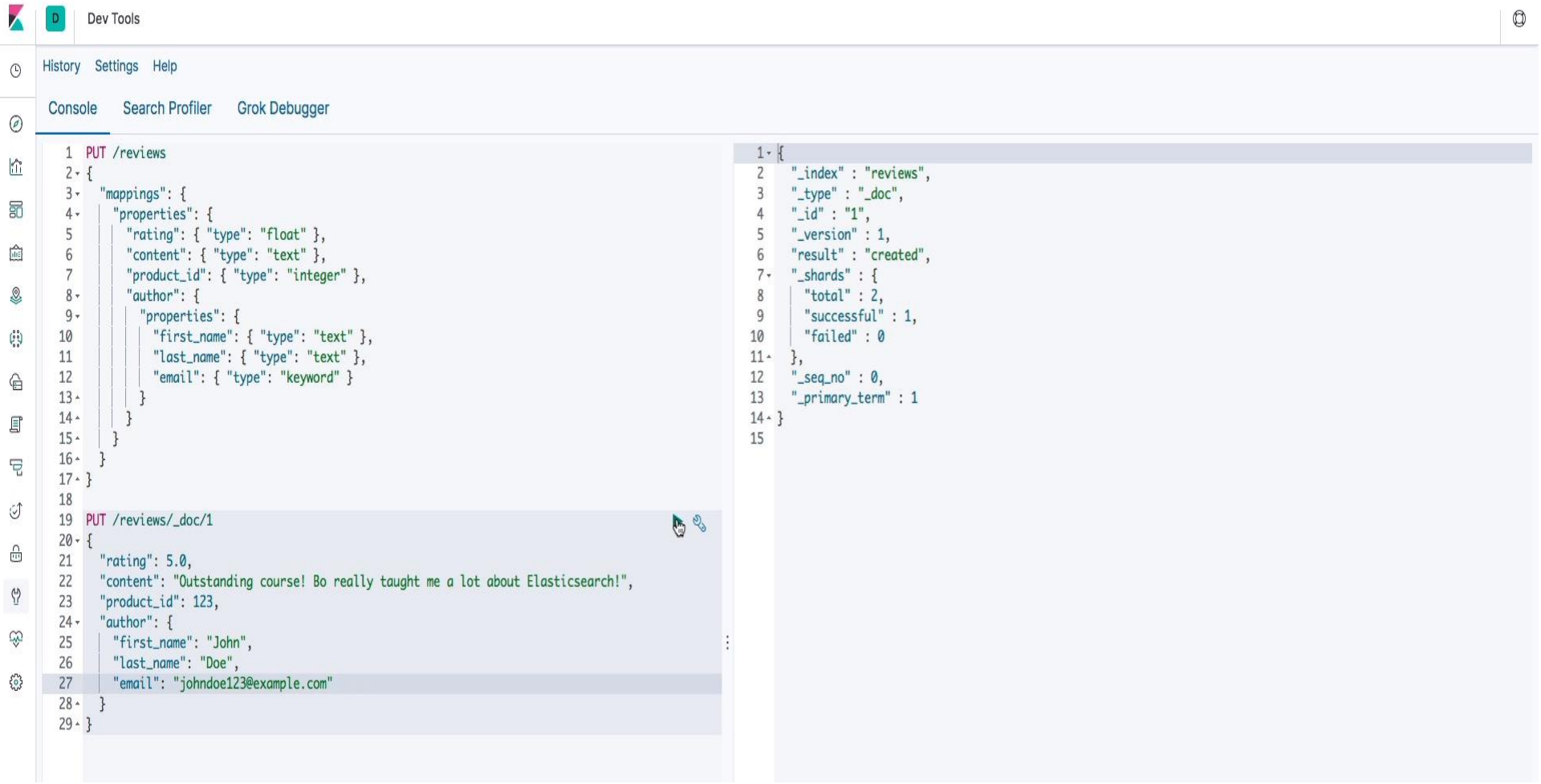
Nested Arrays

- Arrays may contain nested arrays
- Arrays are flattened during indexing
- `[ 1, [ 2, 3 ] ]` becomes `[ 1, 2, 3 ]`

Mapping by Data Type

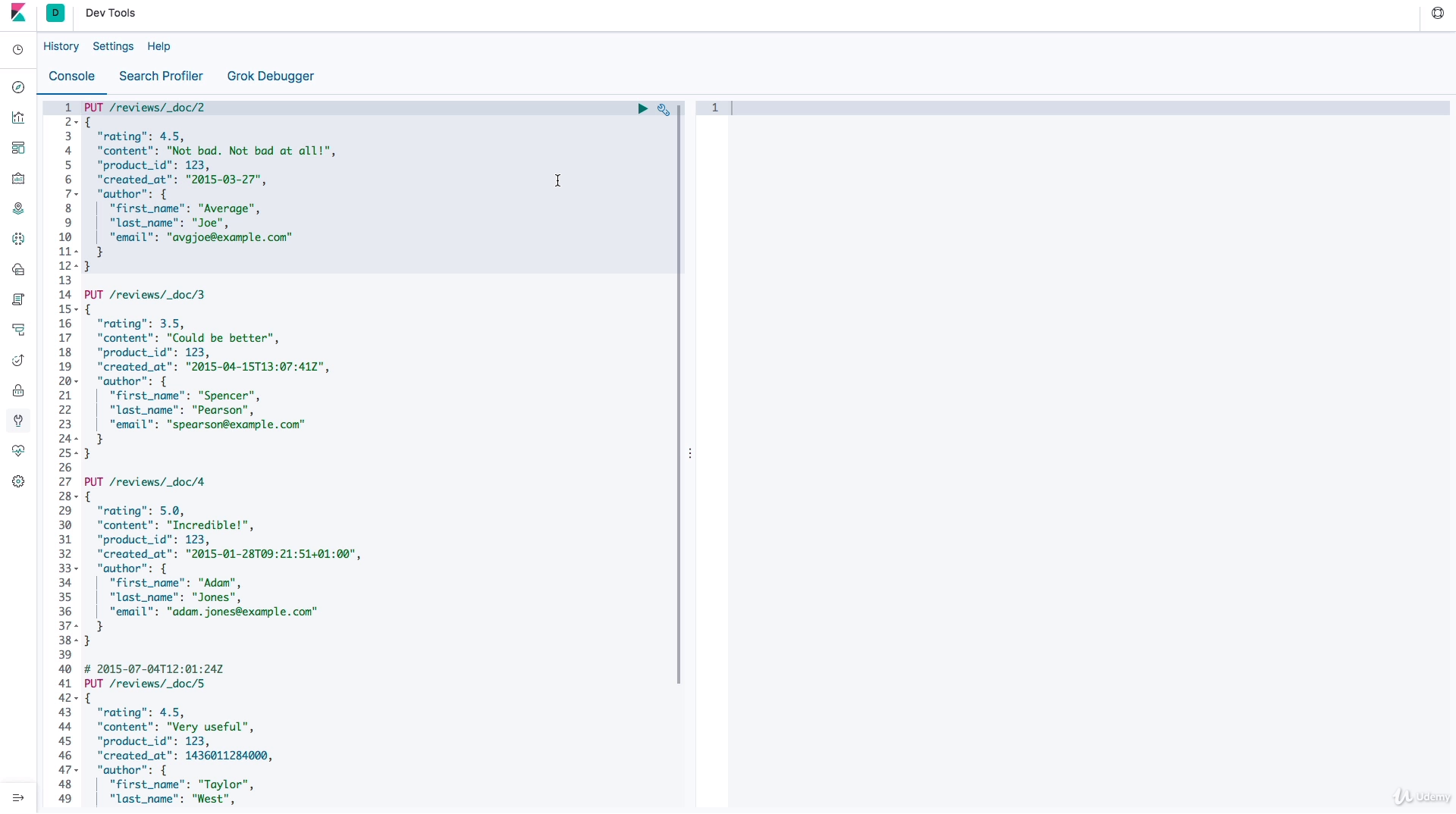


Mapping in Data Types



Mapping in Data Types

Mapping in Data Types



Add Mapping to existing index

Introduction to Date Fields

- Specified in one of three ways;
  - Specially formatted strings
  - Milliseconds since the epoch (long)
  - Seconds since the epoch (integer)
- Epoch refers to the 1st of January 1970
- Custom formats are supported

Default Behaviour of Date Fields

- Three supported formats;
  - A date *without* time
  - A date *with* time
  - Milliseconds since the epoch (long)
- UTC timezone assumed if none is specified
- Dates must be formatted according to the ISO 8601 specification

Date

```
    "email": "spearson@example.com"
  }
}

PUT /reviews/_doc/4
{
  "rating": 5.0,
  "content": "Incredible!",
  "product_id": 123,
  "created_at": "2015-01-28T09:21:51+01:00",
  "author": {
    "first_name": "Adam",
    "last_name": "Jones",
    "email": "adam.jones@example.com"
  }
}

# 2015-07-04T12:01:24Z
PUT /reviews/_doc/5
{
  "rating": 4.5,
  "content": "Very useful",
  "product_id": 123,
  "created_at": 1436011284000,
  "author": {
    "first_name": "Taylor",
    "last_name": "West",
    "email": "twest@example.com"
  }
}

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

```
  "_primary_term" : 1
}
```

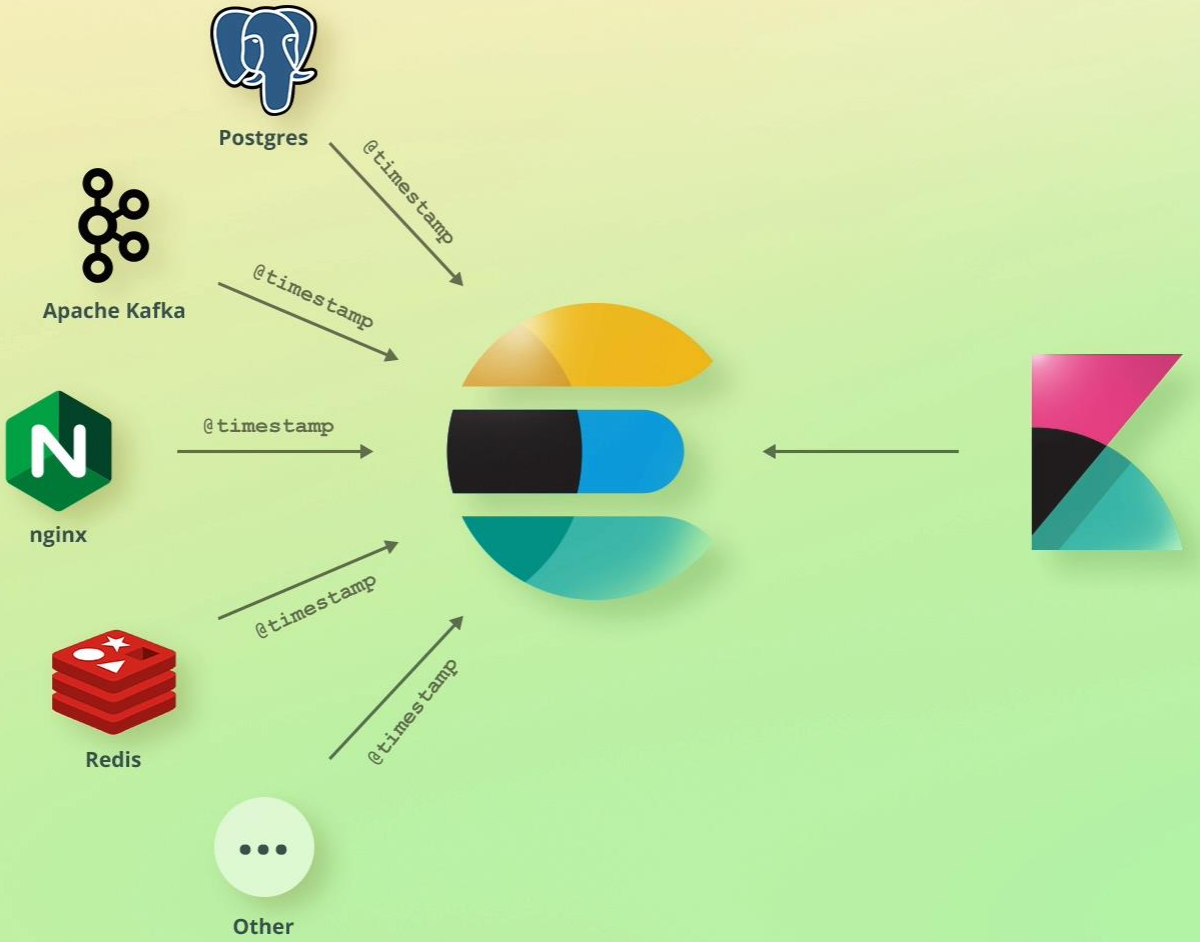
Missing Fields

- All fields in Elasticsearch are optional
- You can leave out a field when indexing documents
- E.g. unlike relational databases where you need to allow `NULL` values
- Some integrity checks need to be done at the application level
  - E.g. having required fields
- Adding a field mapping does *not* make a field required

Limitation to Update Mappings

- Being able to update mappings would be problematic for existing documents
  - Text values have already been analyzed, for instance
  - Changing between some data types would require rebuilding the whole data structure

Elastic Common Schema



Uses of ECS

- In ECS, documents are referred to as *events*
  - ECS doesn't provide fields for non-events (e.g. products)
- Mostly useful for standard events
  - E.g. web server logs, operating system metrics, etc.
- ECS is automatically handled by Elastic Stack products
  - If you use them, you often won't have to actively deal with ECS

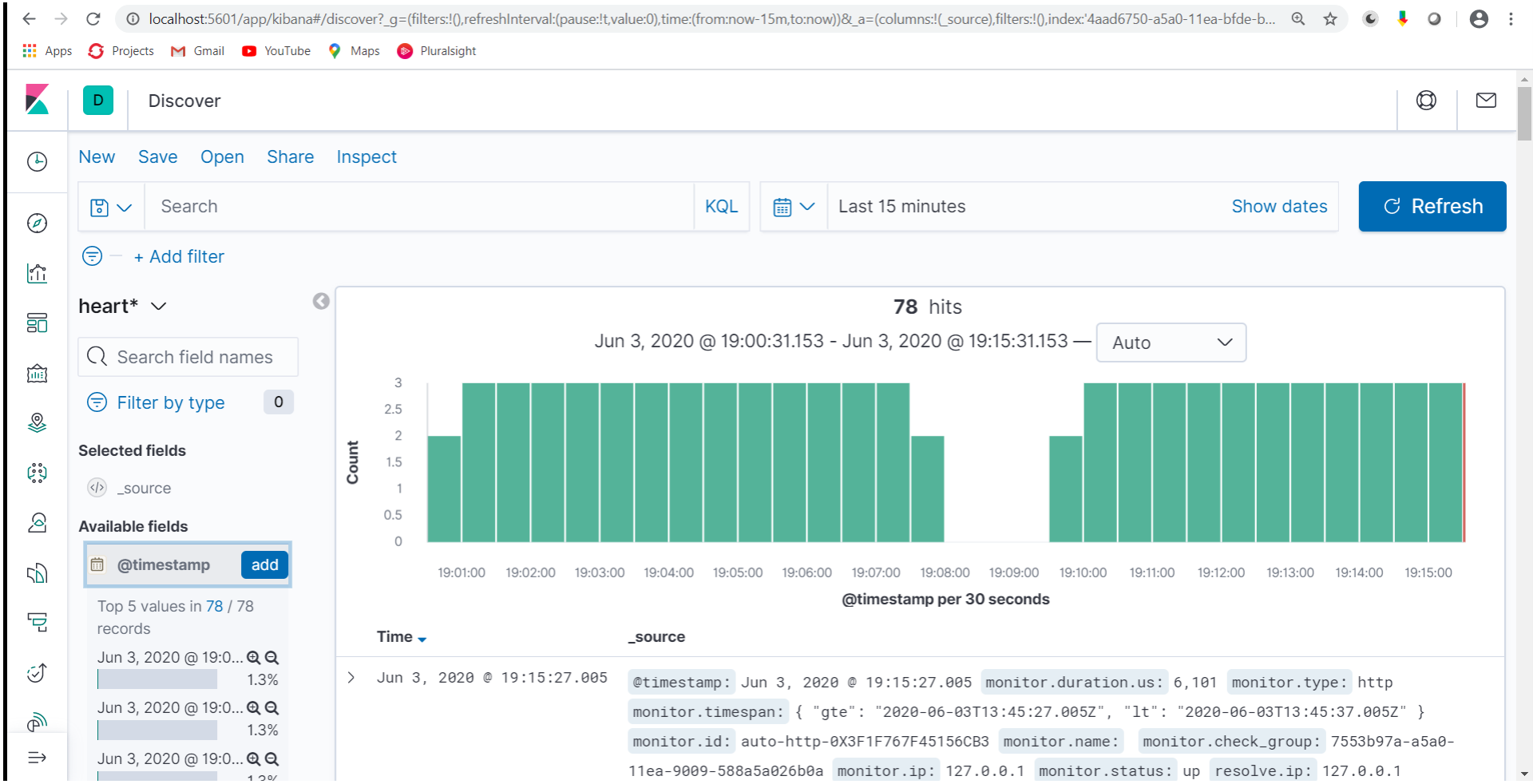
Elastic Common Schema

Heartbeat –c heartbeat.yml –e –d *

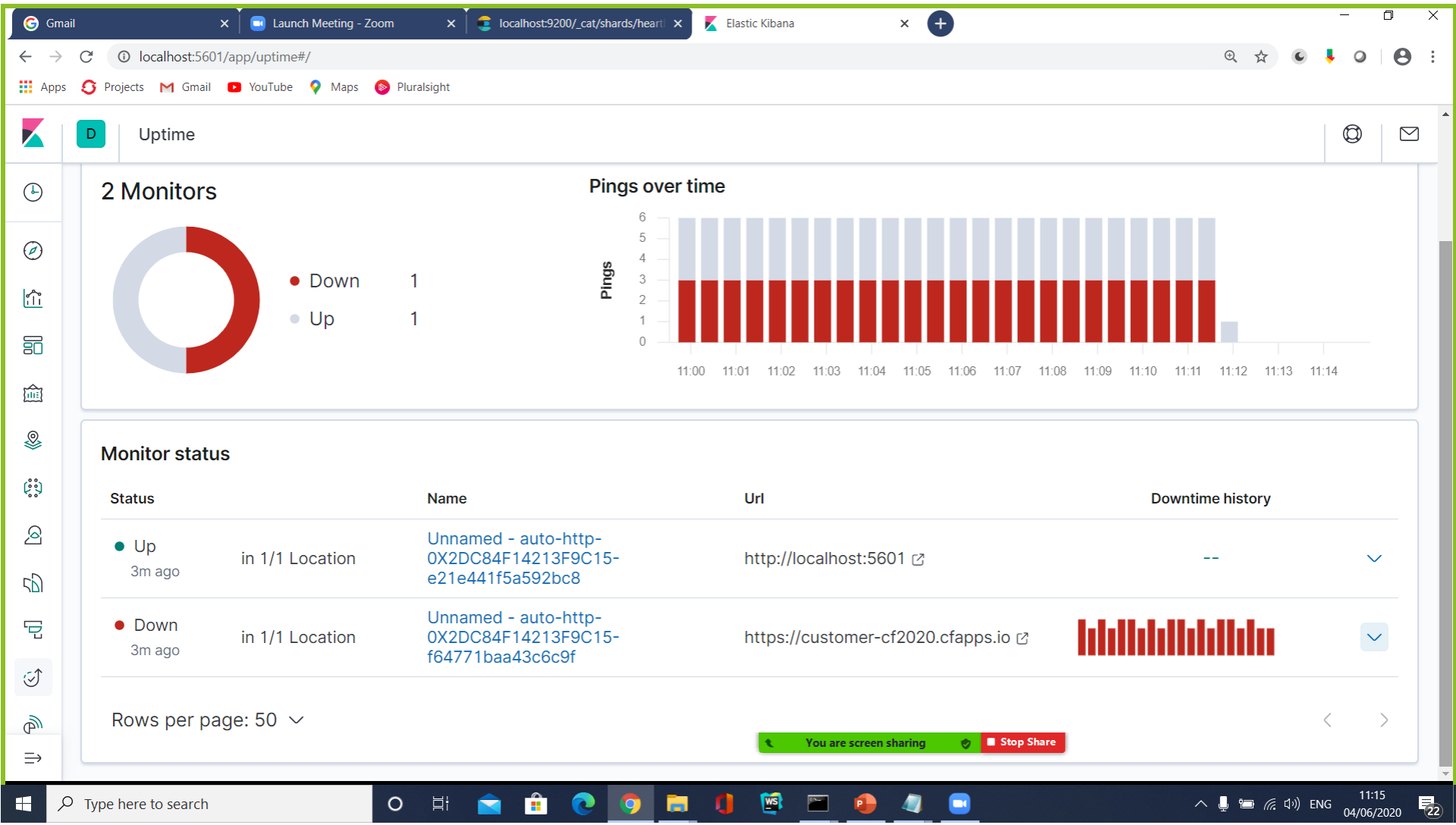
Administrator: Command Prompt - heartbeat -c heartbeat.yml -e -d *

```
      "fe80::7570:a615:c73c:c95",
      "10.102.37.150",
      "169.254.12.149",
      "fe80::bc62:acf0:4b32:24",
      "192.168.0.7",
      "fe80::a806:d6d1:284:2f3b",
      "169.254.47.59"
    ],
    "mac": [
      "58:8a:5a:02:6b:0a",
      "8e:15:9b:65:bc:9b",
      "00:15:5d:fc:f0:03",
      "02:00:4c:4f:4f:50",
      "0a:00:27:00:00:08",
      "f8:34:41:ac:d4:70",
      "fa:34:41:ac:d4:6f",
      "00:ff:28:1b:32:b7",
      "f8:34:41:ac:d4:6f",
      "f8:34:41:ac:d4:73"
    ]
  }
}
2020-06-03T19:19:47.567+0530    DEBUG   [scheduler]     scheduler/scheduler.go:195      Job 'auto-http-0X3F1F767F45156CB3' returned at
2020-06-03 19:19:47.5674444 +0530 IST m=+601.120184301
2020-06-03T19:19:47.740+0530    DEBUG   [elasticsearch] elasticsearch/client.go:217     PublishEvents: 2 events have been published to
elasticsearch in 72.9698ms.
2020-06-03T19:19:47.748+0530    DEBUG   [publisher]     memqueue/ackloop.go:160 ackloop: receive ack [59: 0, 2]
2020-06-03T19:19:47.749+0530    DEBUG   [publisher]     memqueue/eventloop.go:535       broker ACK events: count=2, start-seq=60, end-s
eq=61

2020-06-03T19:19:47.750+0530    DEBUG   [publisher]     memqueue/ackloop.go:128 ackloop: return ack to broker loop:2
2020-06-03T19:19:47.750+0530    DEBUG   [publisher]     memqueue/ackloop.go:131 ackloop:  done send ack
```

Elastic Common Schema



Elastic Common Schema



Dynamic Mapping

```
POST /my-index/_doc
{
  "tags": ["computer", "electronics"],
  "in_stock": 4,
  "created_at": "2020/01/01 00:00:00"
}
```

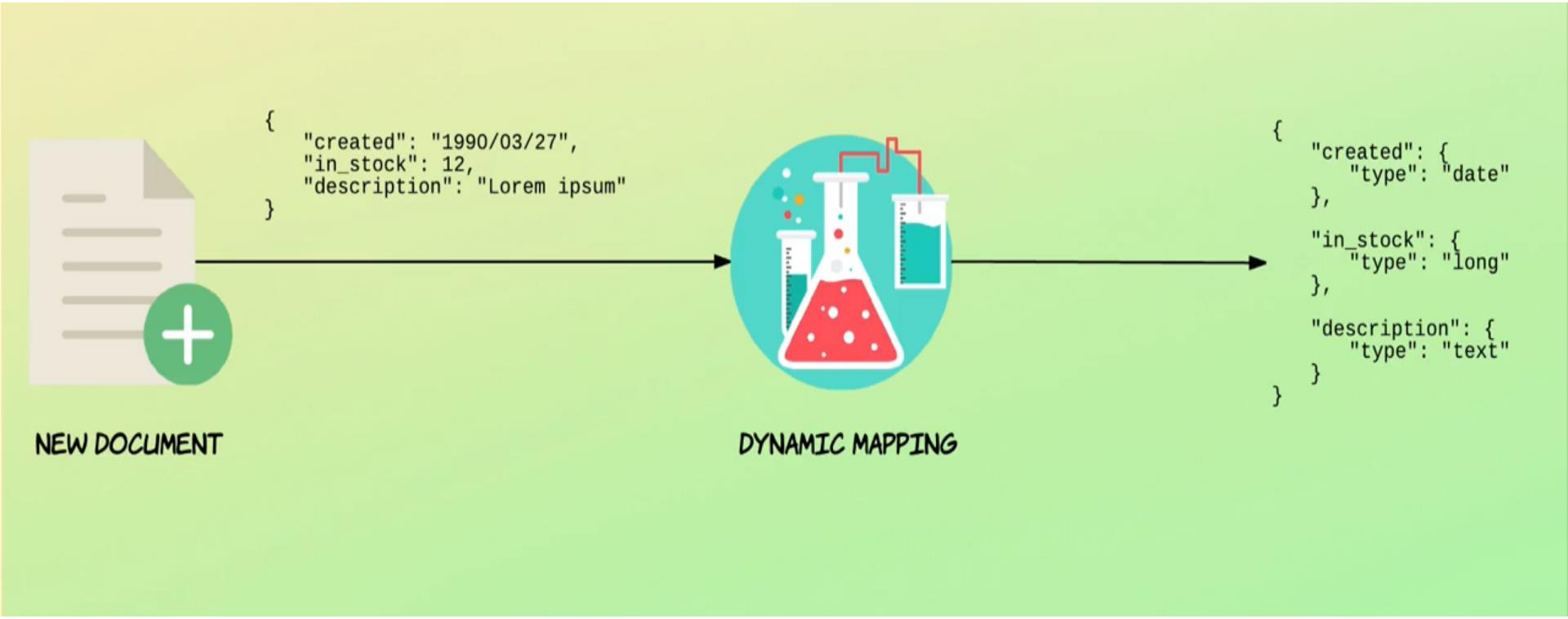
```
{
  "created_at": {
    "type": "date",
    "format": "yyyy/MM/dd HH:mm:ss||yyyy/MM/dd||epoch_millis"
  },
  "in_stock": {
    "type": "long"
  },
  "tags": {
    "type": "text",
    "fields": {
      "keyword": {
        "type": "keyword",
        "ignore_above": 256
      }
    }
  }
}
```

Dynamic Mapping

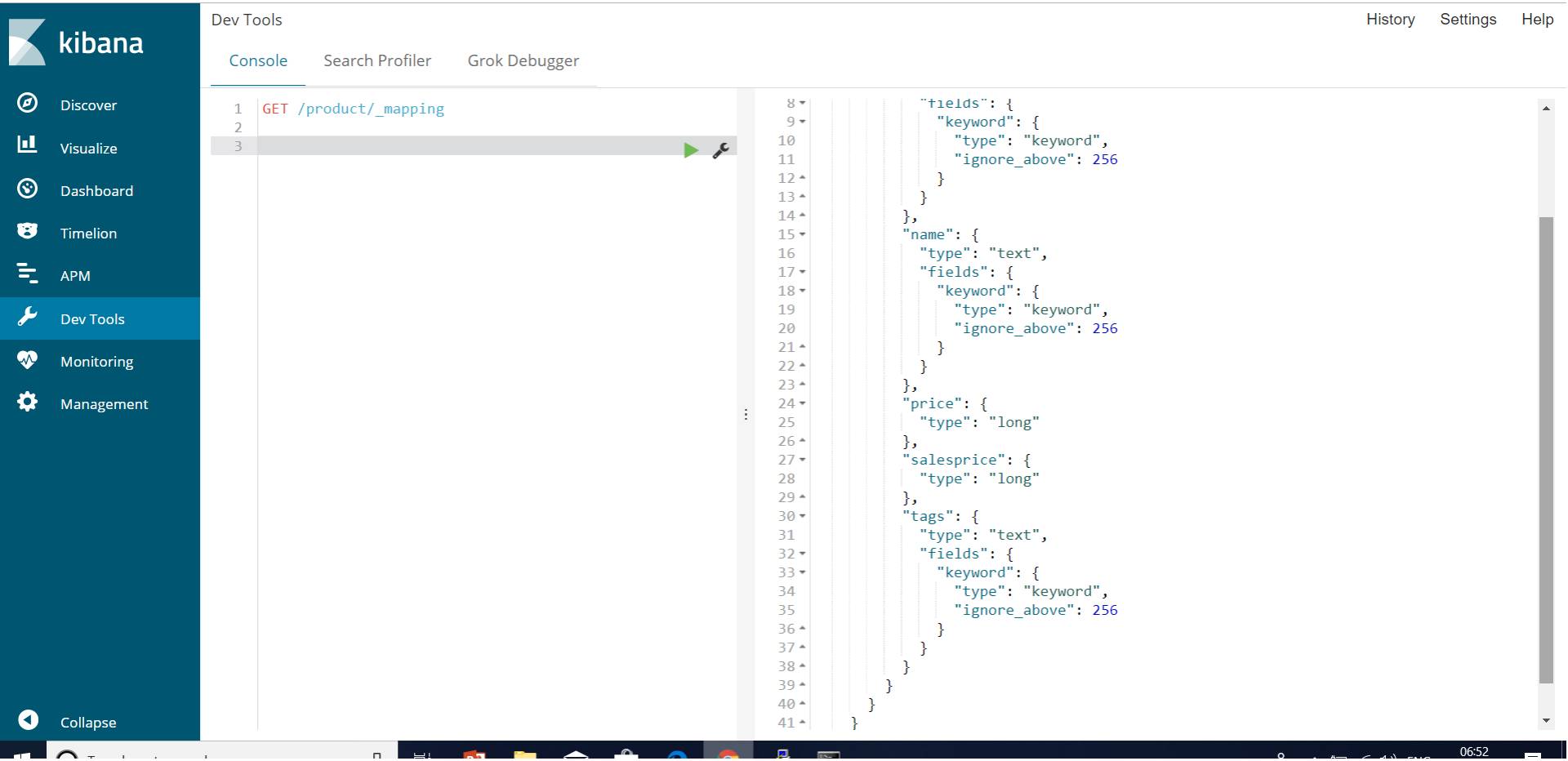
JSON	ELASTICSEARCH
string	One of the following: • `text` field with `keyword` mapping • `date` field • (`float` or `long` field)
integer	`long`
floating point number	`float`
boolean (`true` or `false`)	`boolean`
object	`object`
array	Depends on the first non-null value

```
[ "strict_date_optional_time","yyyy/MM/dd HH:mm:ss Z||yyyy/MM/dd Z"]
```

Managing Documents Dynamic Mapping



Managing Documents Dynamic Mapping



Use Explicit Mappings

- Dynamic mapping is convenient, but often not a good idea in production
- Save disk space with optimized mappings when storing many documents
- Set `dynamic` to `"strict"`, not `false`
  - Avoids surprises and unexpected results

Mapping of text fields

- Don’t always map strings as both `text` and `keyword`
  - Typically only one is needed
  - Each mapping requires disk space
- Do you need to perform full-text searches?
  - Add a `text` mapping
- Do you need to do aggregations, sorting, or filtering on exact values?
  - Add a `keyword` mapping

Disable Coercion

- Coercion forgives you for not doing the right thing
- Try to do the right thing instead
- Always use the correct data types whenever possible

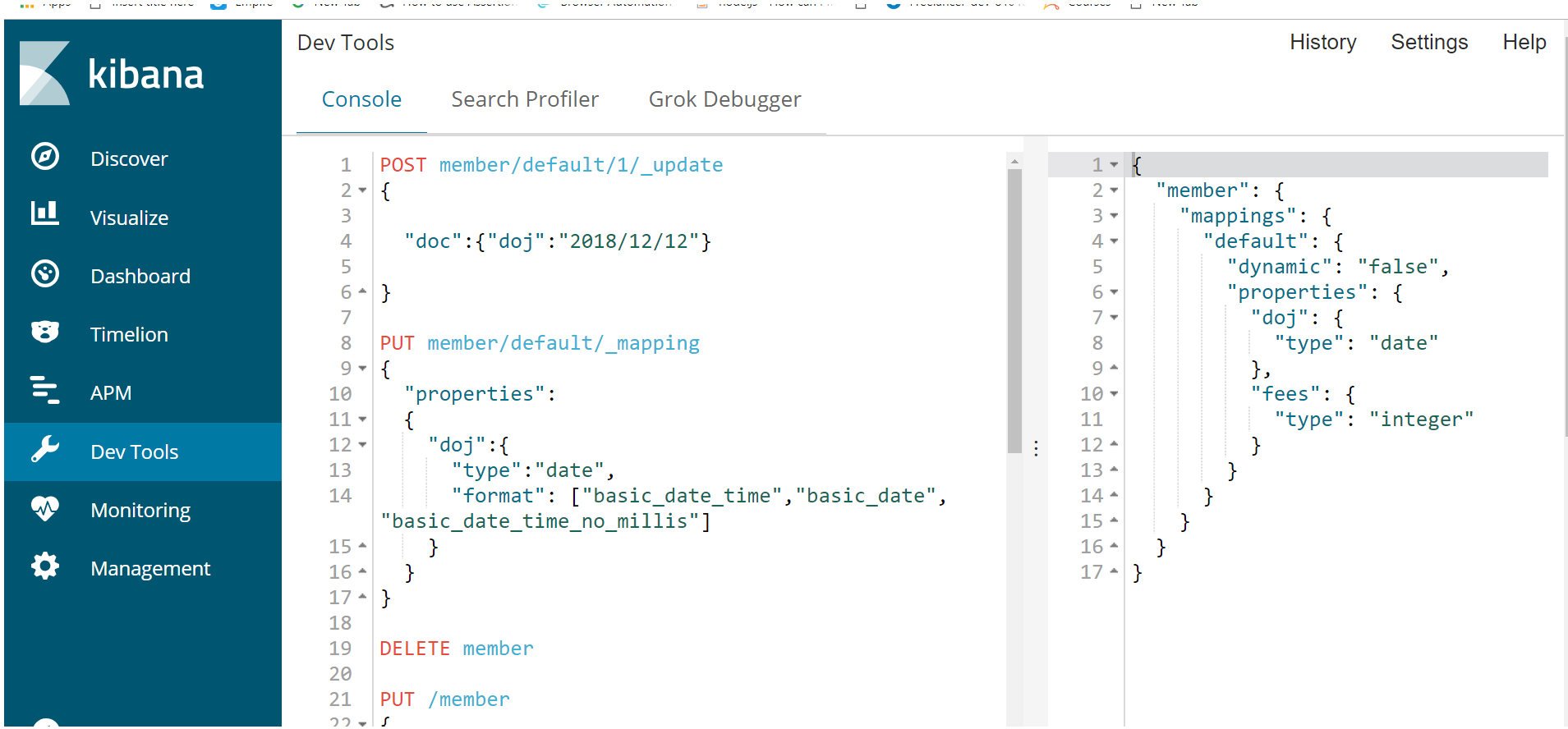
Use appropriate numeric data types

- For whole numbers, the `integer` data type might be enough
  - `long` can store larger numbers, but also uses more disk space
- For decimal numbers, the `float` data type might be precise enough
  - `double` stores numbers with a higher precision but uses 2x disk space
  - Usually, `float` provides enough precision

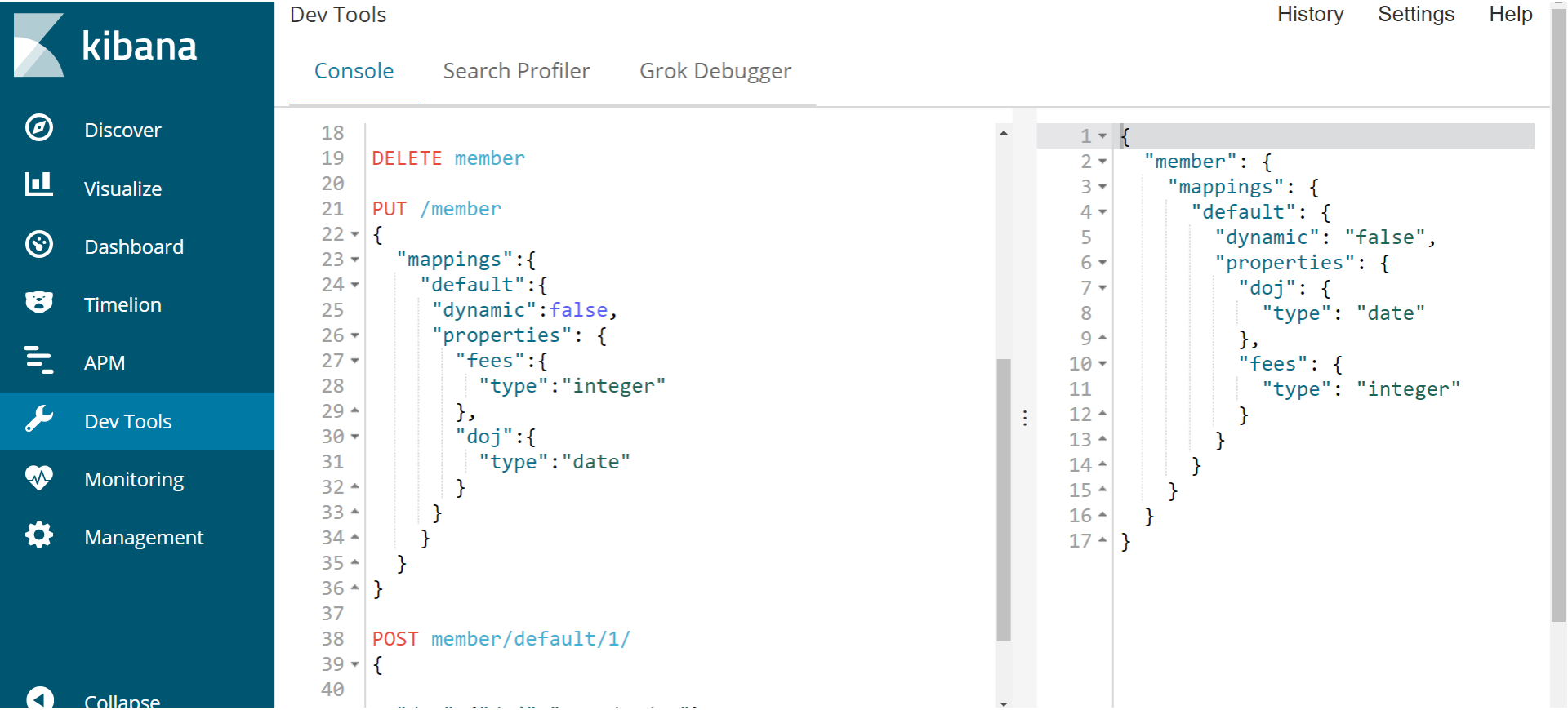
Mapping Parameters

- Set `doc_values` to `false` if you don't need sorting, aggregations, and scripting
- Set `norms` to `false` if you don't need relevance scoring
- Set `index` to `false` if you don't need to filter on values
  - You can still do aggregations, e.g. for time series data
- Probably only worth the effort when storing lots of documents
  - Otherwise it's probably an over complication
- Worst case scenario, you will need to reindex documents

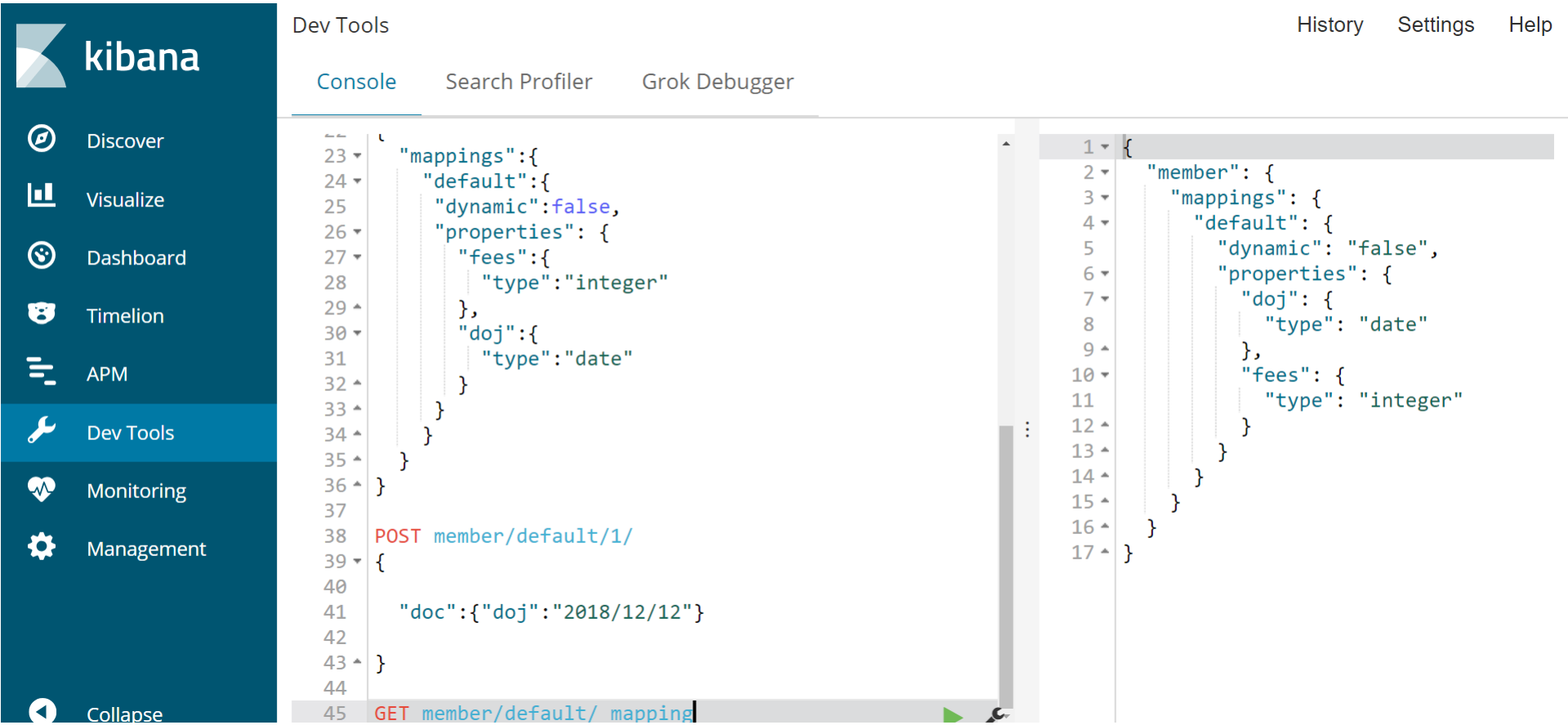
Managing Documents Custom Data Type



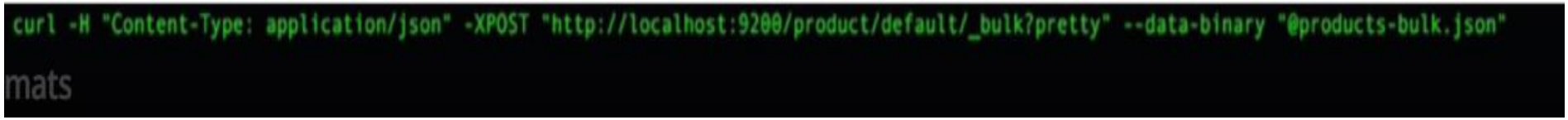
Managing Documents Custom Data Type



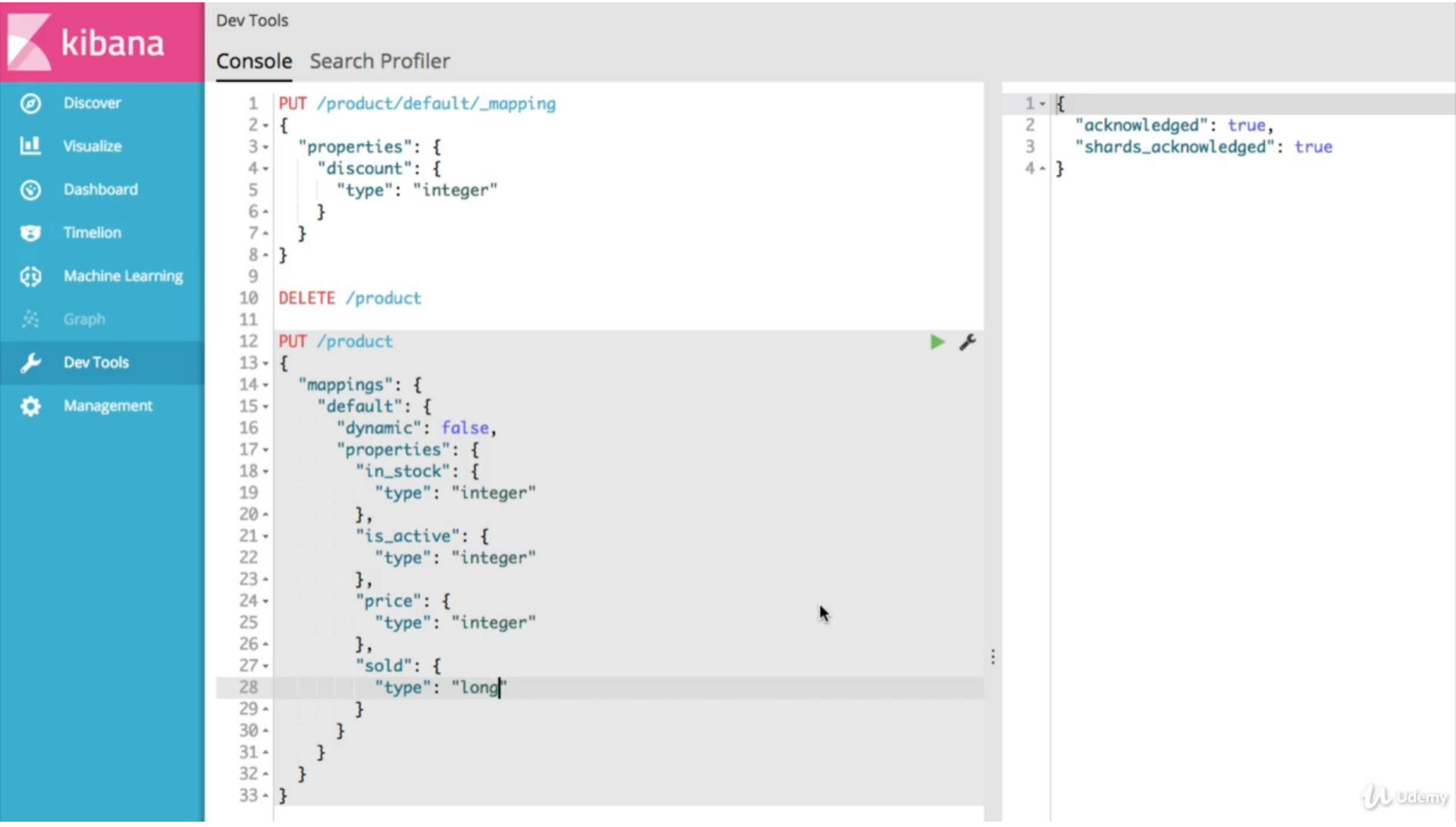
Managing Documents Custom Data Type

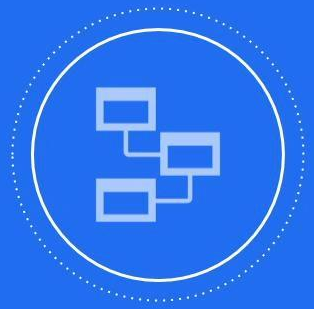


Managing Documents Custom Data Type



Managing Documents Changing Mapping

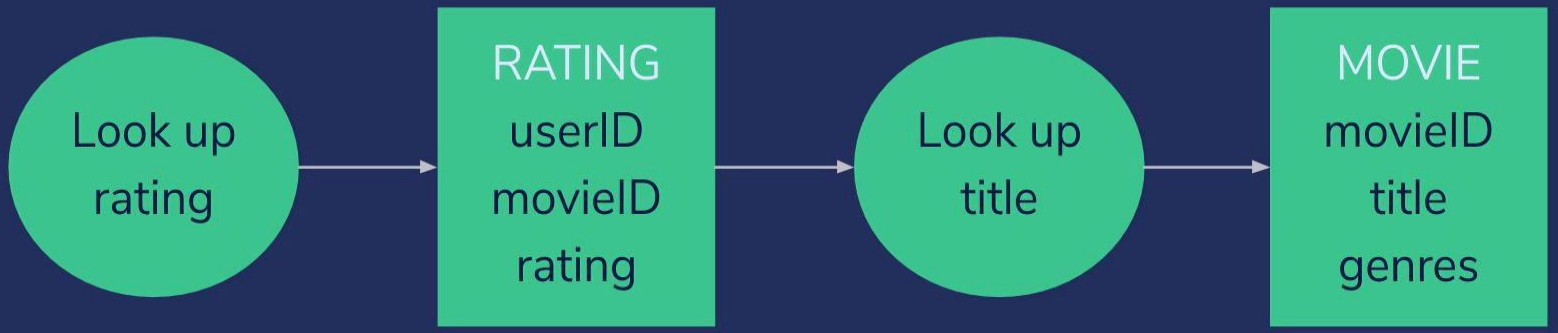




Data Modeling

Strategies For Relational Data

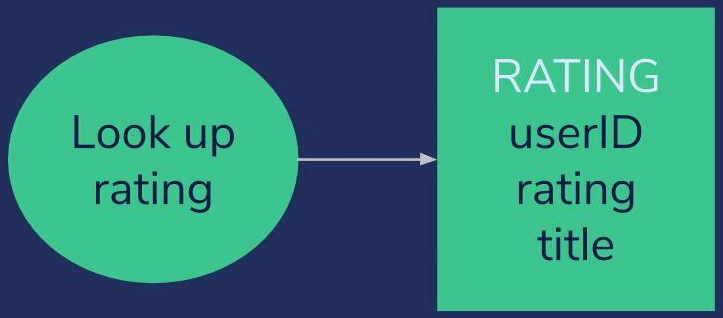
Normalized data



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

Strategies For Relational Data

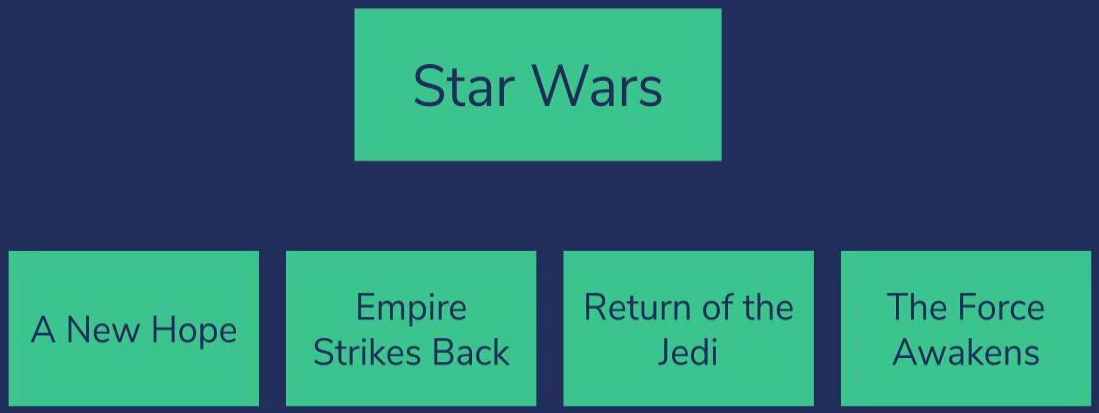
Denormalized Data



Titles are duplicated, but only one query

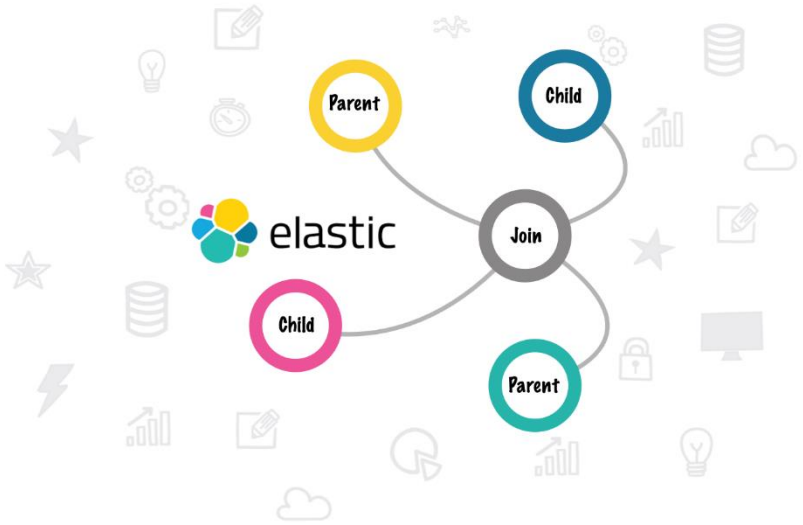
Strategies For Relational Data

Parent / Child Relationship



Parent child Relationship

```
PUT /music-5_6
{
  "settings": {
    "number_of_shards": 1, "number_of_replicas": 0,
    "mapping.single_type": true
  },
  "mappings": {
    "doc": {
      "properties": {
        "artist": { "type": "text" },
        "song": { "type": "text" },
        "user": { "type": "keyword" },
        "artist_relations": {
          "type": "join",
          "relations": {
            "artist": "song",
            "song": "user"
          }    }    }    }  } }
```



Refer github.com/rpsvelkjan2023

Parent child Relationship

- POST /music-5_6/doc/_bulk
- {"index":{"_id":1}}
- {"name":"John Legend","artist_relations":{"name":"artist"}}
- {"index":{"_id":2}}
- {"name":"Ariana Grande","artist_relations":{"name":"artist"}}

Parent child Relationship

- PUT music-5_6/doc/3?routing=1
- {"song":"All of Me","artist_relations":{"name":"song","parent":1}}

- PUT music-5_6/doc/4?routing=1
- {"song":"Beauty and the Beast","artist_relations":{"name":"song","parent":1}}

- PUT music-5_6/doc/5?routing=2
- {"song":"Beauty and the Beast","artist_relations":{"name":"song","parent":2}}

Parent child Relationship

- POST music-5_6/doc/_bulk?routing=3
- {"index":{"_id":"l-1"}}
- {"user":"Gabriel","artist_relations":{"name":"user","parent":3}}
- {"index":{"_id":"l-2"}}
- {"user":"Berte","artist_relations":{"name":"user","parent":3}}
- {"index":{"_id":"l-3"}}
- {"user":"Emma","artist_relations":{"name":"user","parent":3}}

Parent child Relationship

- GET music-5_6/_search
- {
- "query": {
- "has_parent": {
- "parent_type": "artist",
- "query": { "match": { "name": "John Legend" } }
- }
- }
- }

Parent child Relationship

```
GET music-5_6/_search
{
  "query": {
    "has_parent": {
      "parent_type": "song",
      "query": {
        "match": { "song": "all of Me" }
      }
    }
  }
}
```

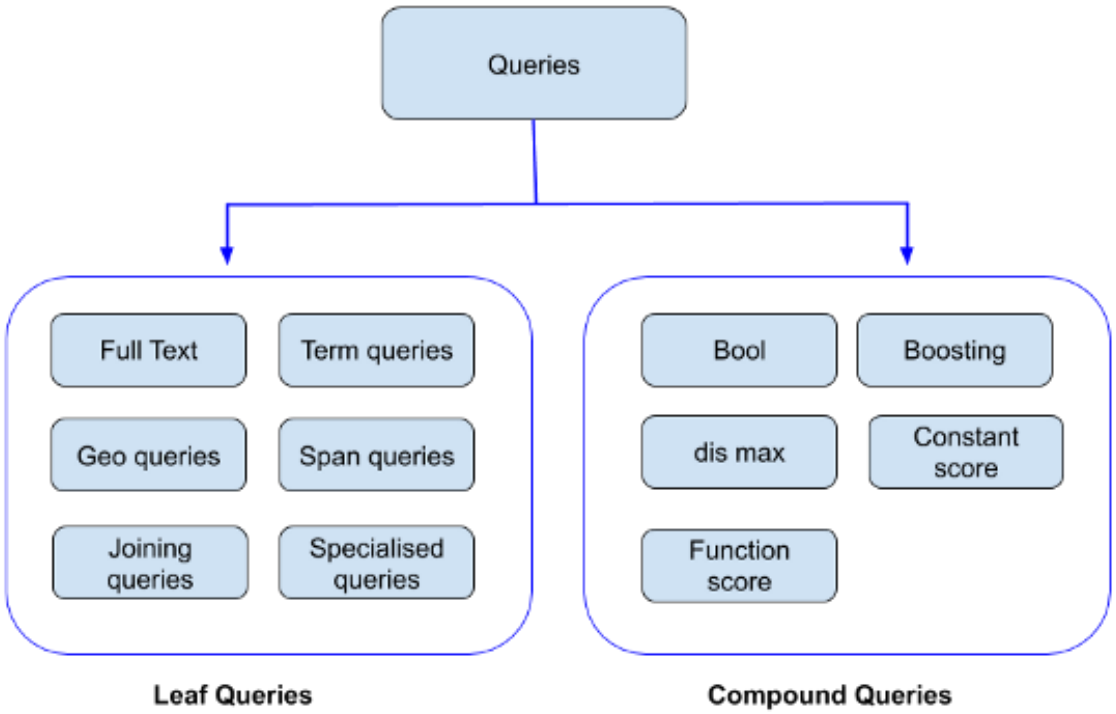
Parent child Relationship

```
GET music/_search
{
  "query": {
    "has_child": {
      "type": "song",
      "min_children": 1, "max_children": 10,
      "query": { "match_all": {} }
    }
  }
}
```

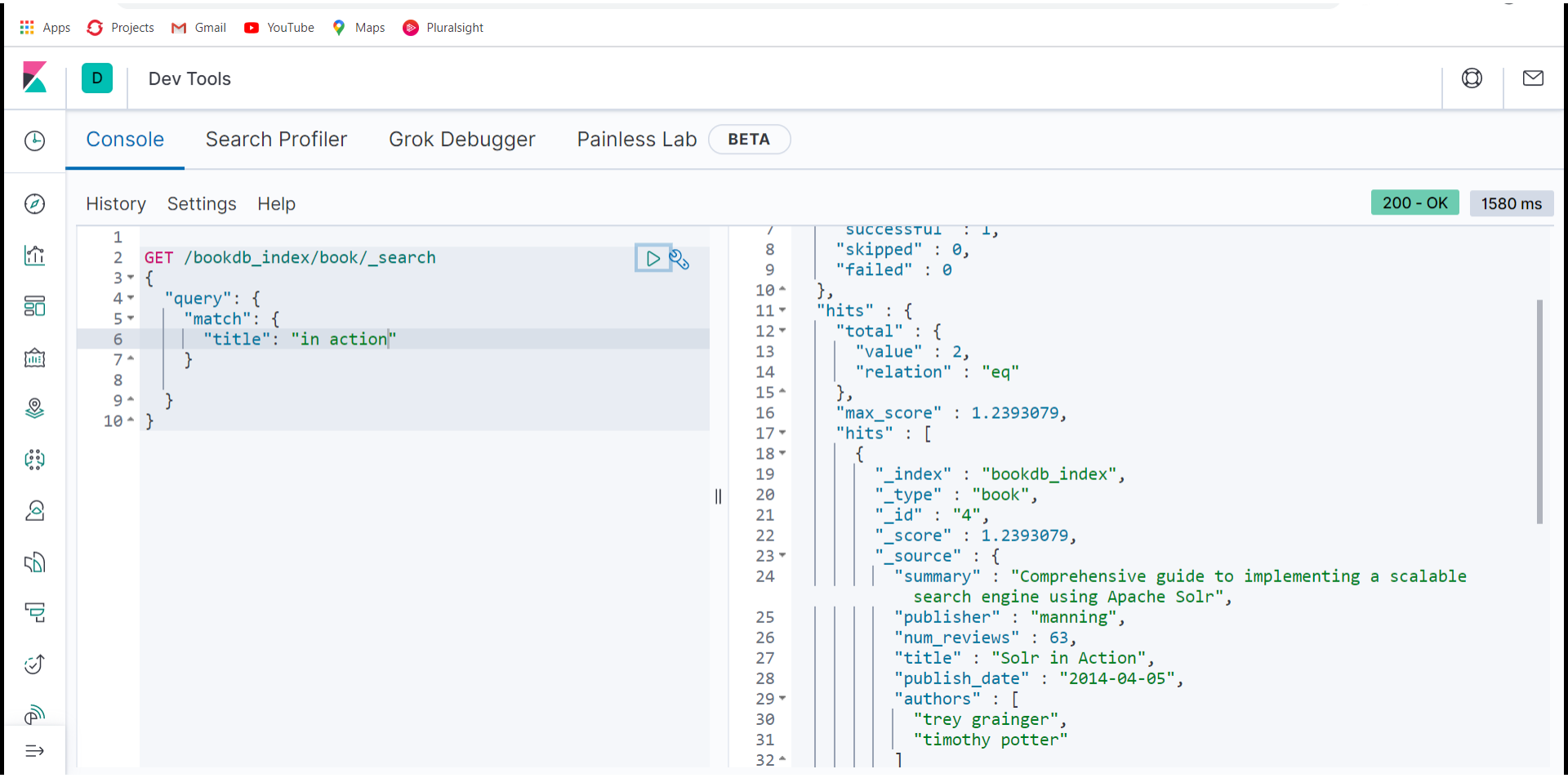
Parent child Relationship

- GET music-5_6/doc/3?routing=1
- {
- "_index": "music-5_6",
- "_type": "doc",
- "_id": "3",
- "_version": 3,
- "_routing": "1",
- "found": true,
- "_source": {
- "song": "All of Me",
- "artist_relations": {
- "name": "song",
- "parent": 1
- }
- }
- }

Queries

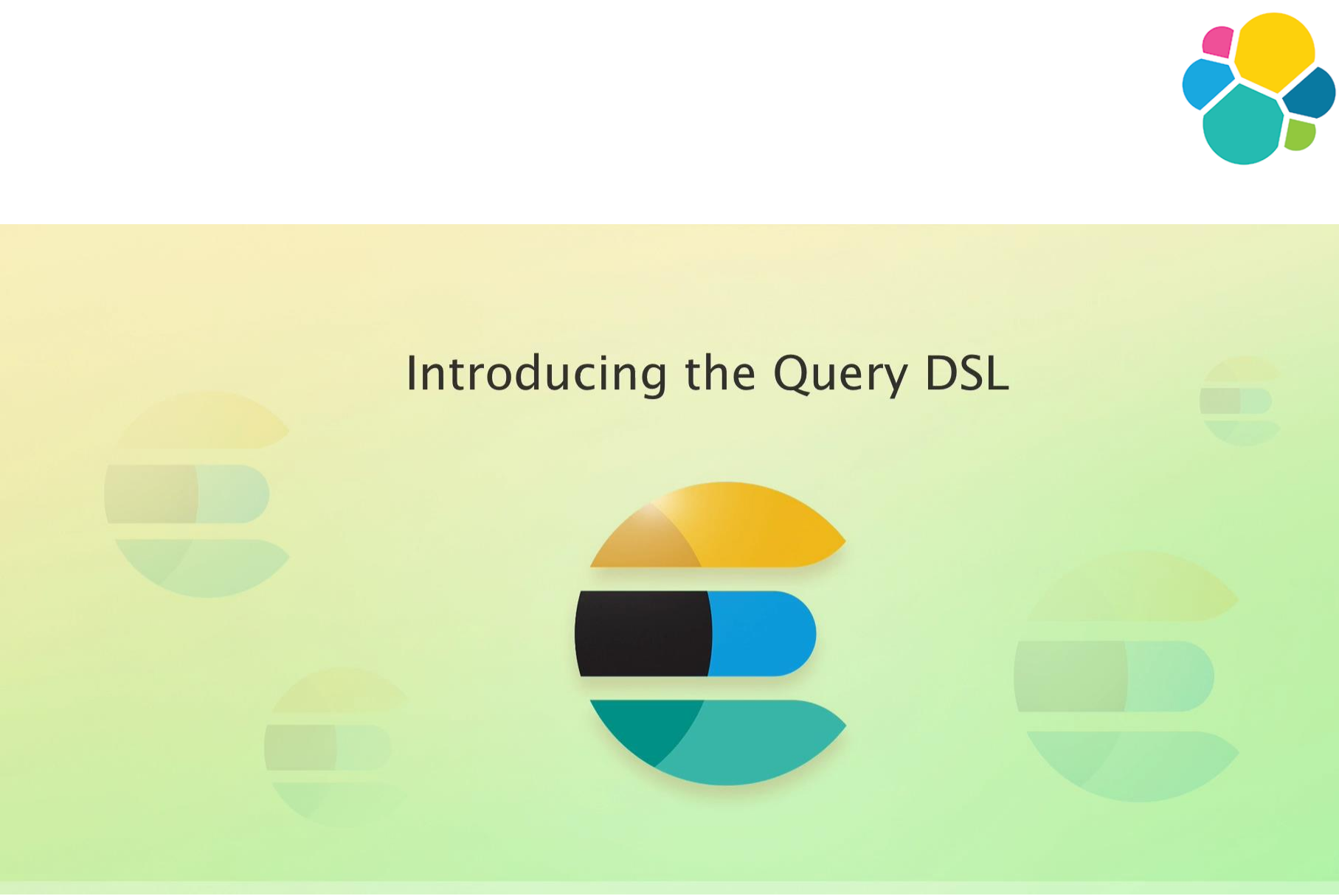

Queries
Full Text
Term queries
Geo queries
Span queries
Joining queries
Specialised queries
Bool
Boosting
dis max
Constant score
Function score
Leaf Queries
Compound Queries

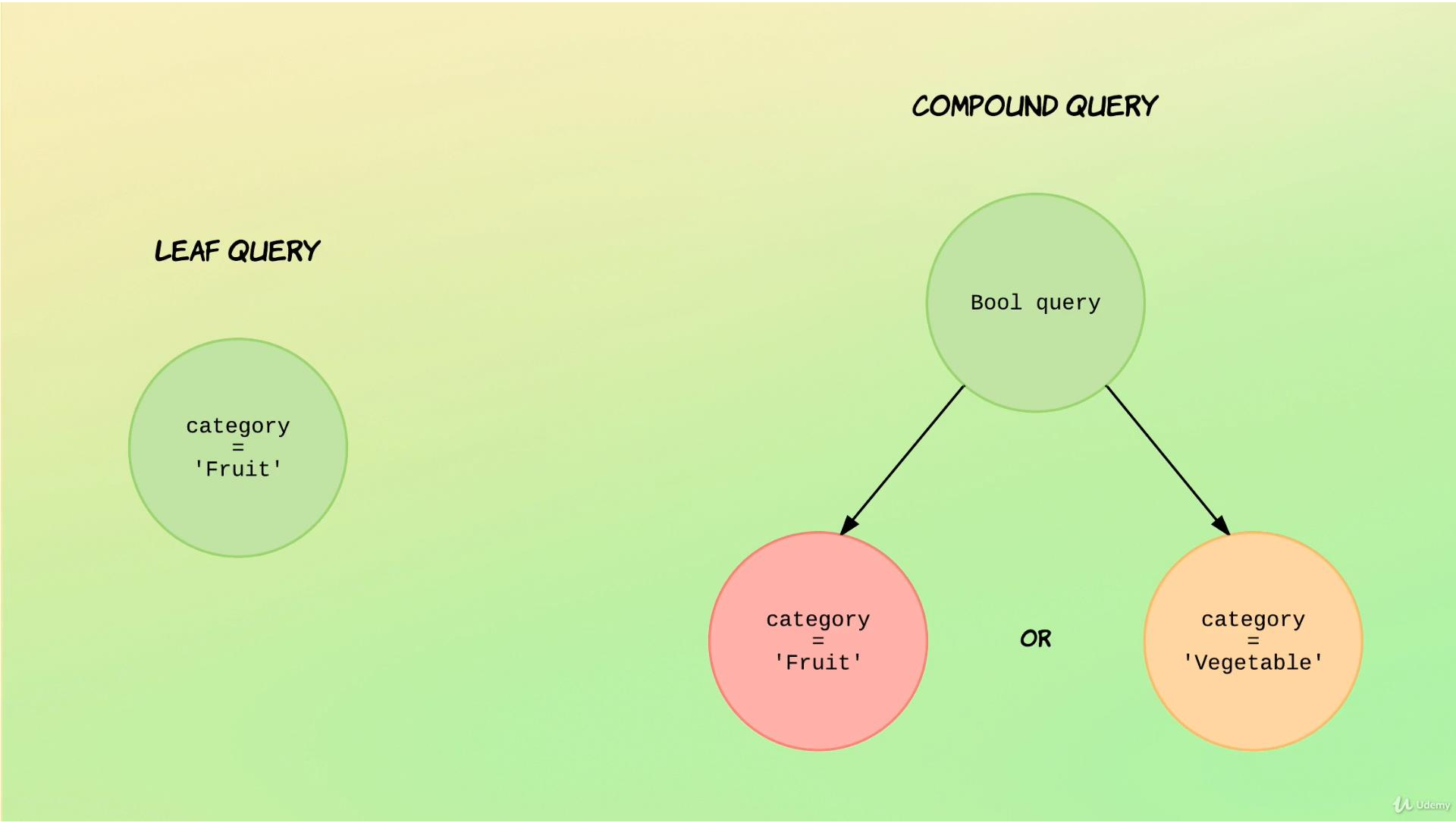
Search Methods Term Search



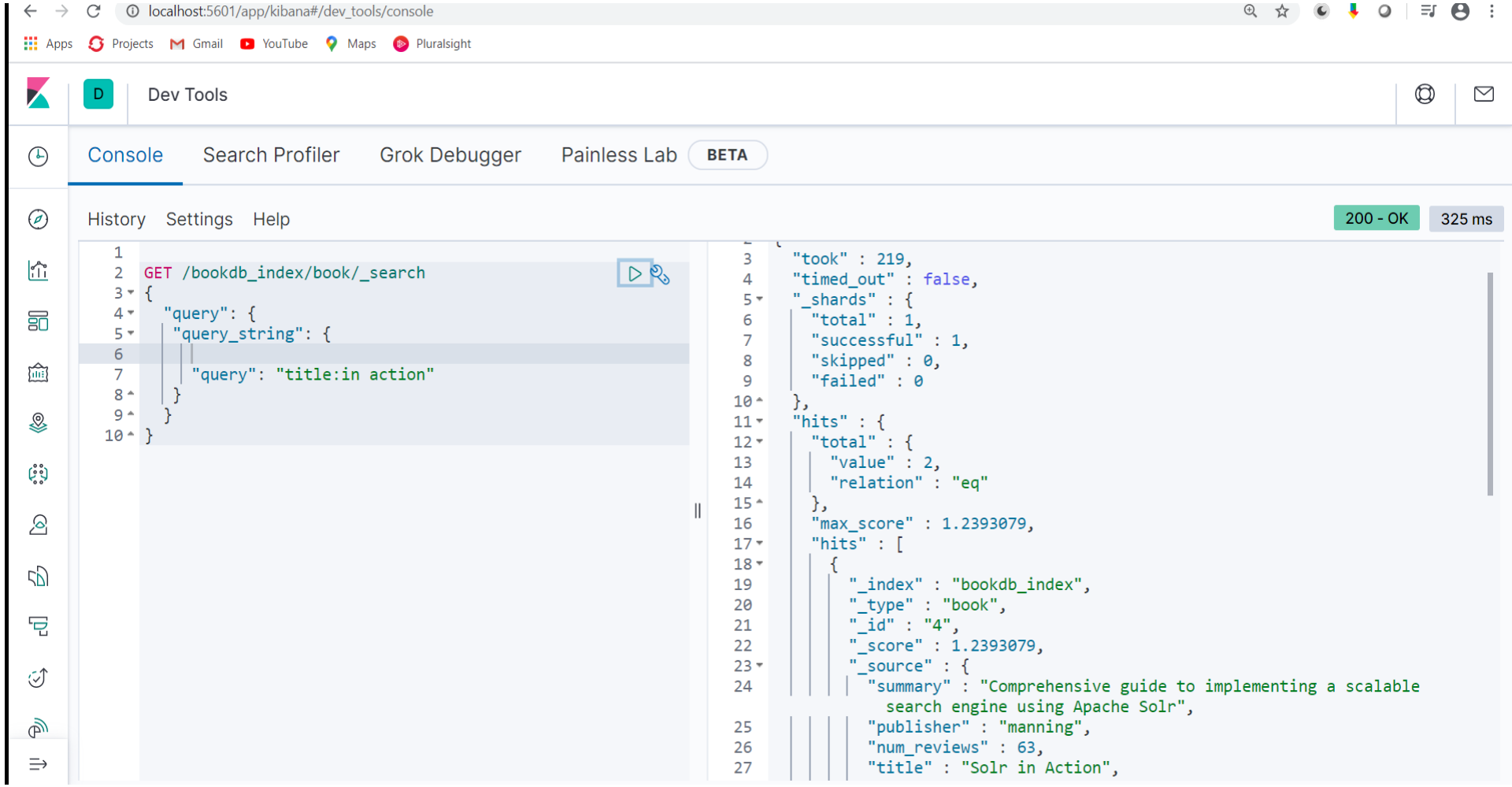
Request URI Search

- GET /bookdb_index/book/_search?q=title:in action

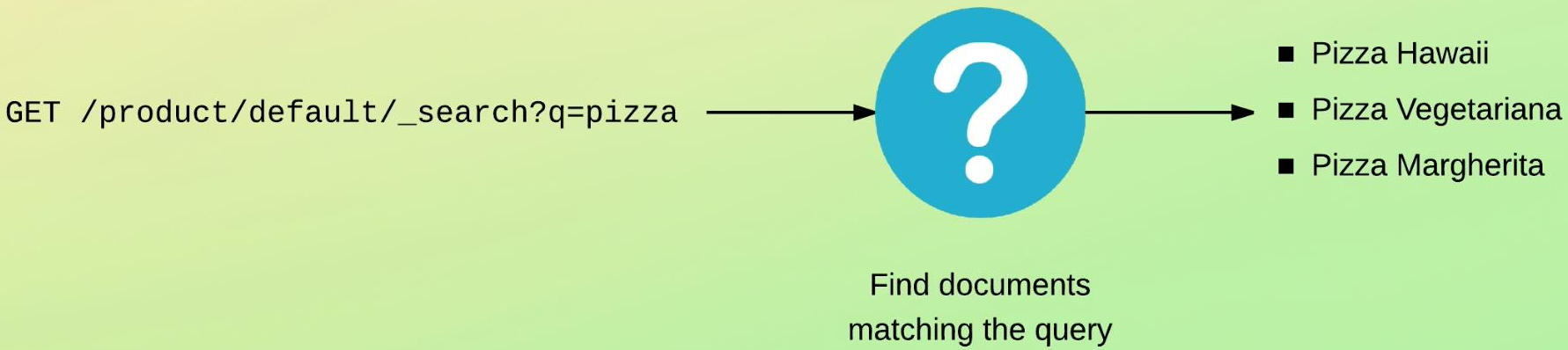

Introducing the Query DSL


COMPOUND QUERY
LEAF QUERY
category = 'Fruit'
Bool query
category = 'Fruit'
OR
category = 'Vegetable'

Query DSL

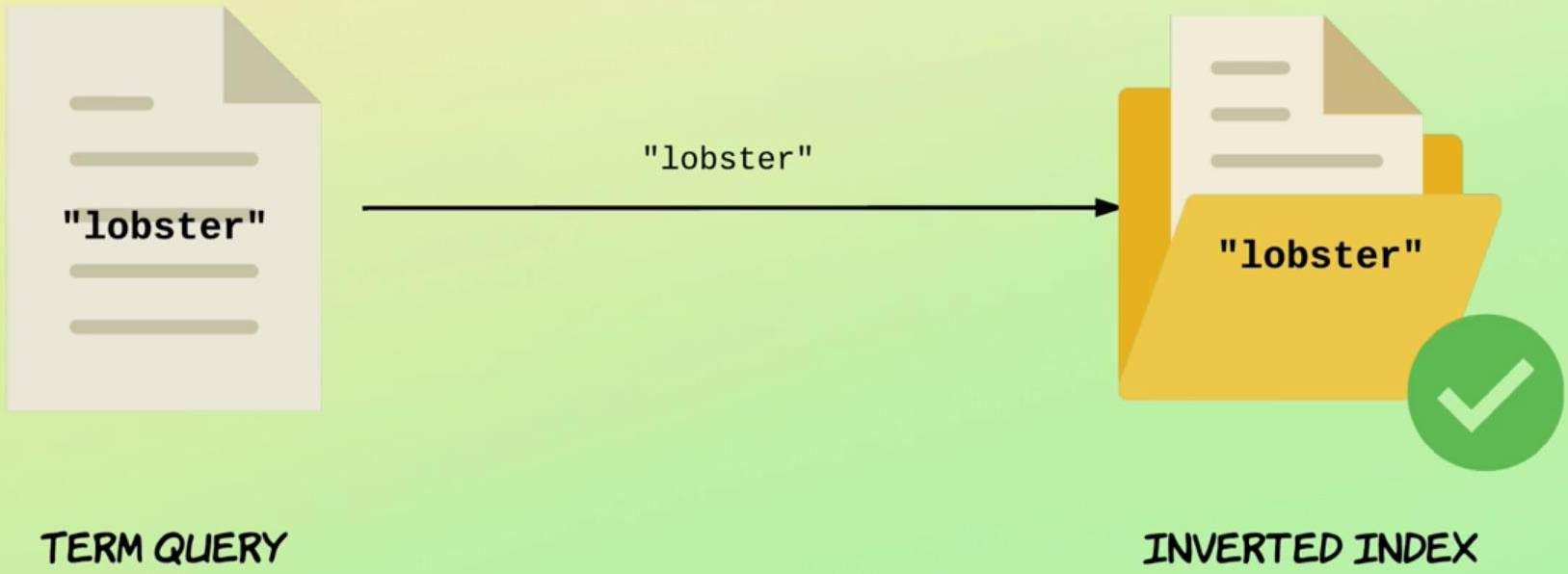

localhost:5601/app/kibana#/dev_tools/console
Apps
Projects
Gmail
YouTube
Maps
Pluralsight
Dev Tools
Console
Search Profiler
Grok Debugger
Painless Lab
BETA
History
Settings
Help
200 - OK
325 ms
GET /bookdb_index/book/_search
{
"query": {
"query_string": {
"query": "title:in action"
}
}
}
{
"took" : 219,
"timed_out" : false,
"_shards" : {
"total" : 1,
"successful" : 1,
"skipped" : 0,
"failed" : 0
},
"hits" : {
"total" : {
"value" : 2,
"relation" : "eq"
},
"max_score" : 1.2393079,
"hits" : [
{
"_index" : "bookdb_index",
"_type" : "book",
"_id" : "4",
"_score" : 1.2393079,
"_source" : {
"summary" : "Comprehensive guide to implementing a scalable search engine using Apache Solr",
"publisher" : "manning",
"num_reviews" : 63,
"title" : "Solr in Action",


Understanding relevance scores

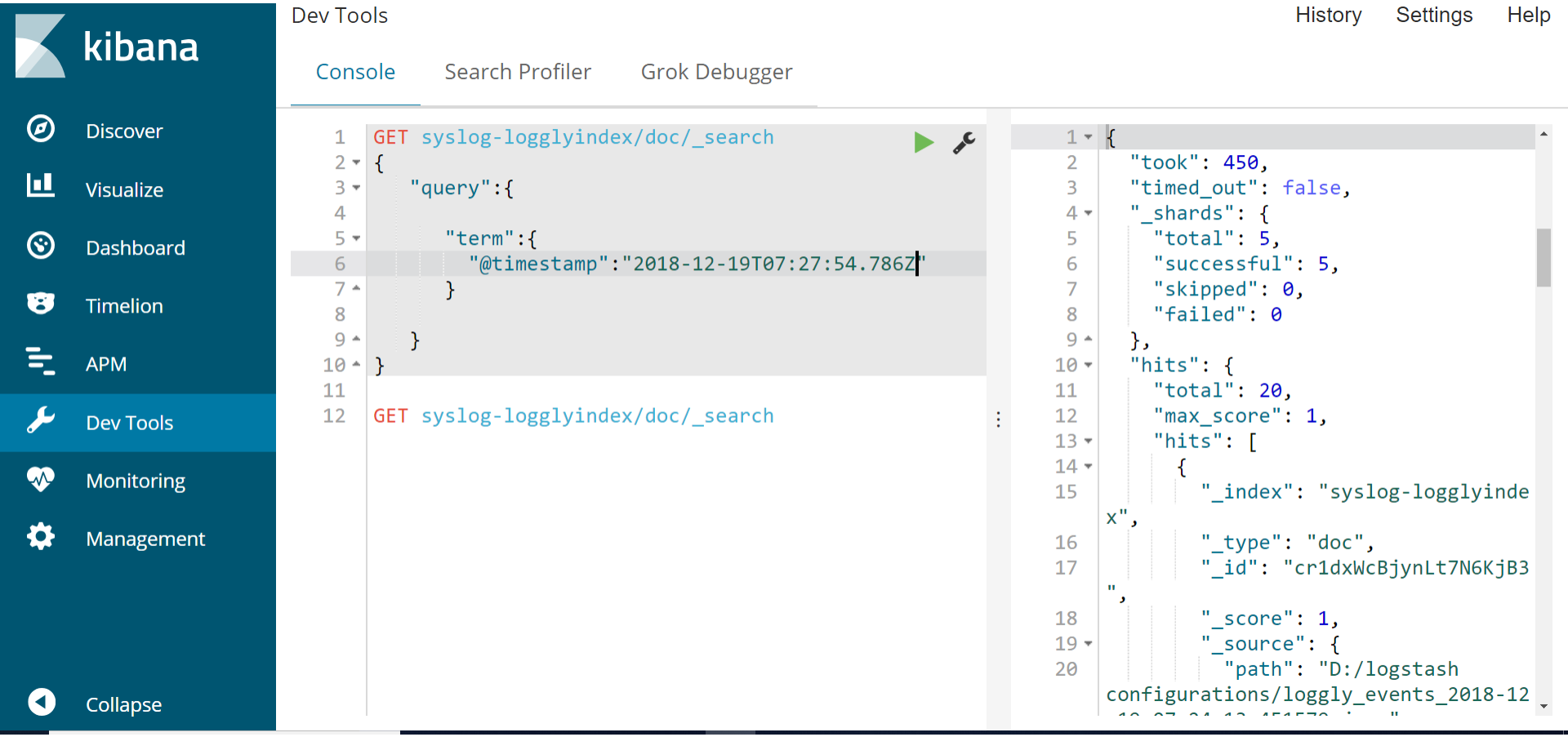

GET /product/default/_search?q=pizza
Pizza Hawaii
Pizza Vegetariana
Pizza Margherita
Find documents
matching the query


GET /product/default/_search?q=pizza
Find documents matching the query
How *well* do the documents match?
1. Pizza Hawaii
2. Pizza Vegetariana
3. Pizza Margherita

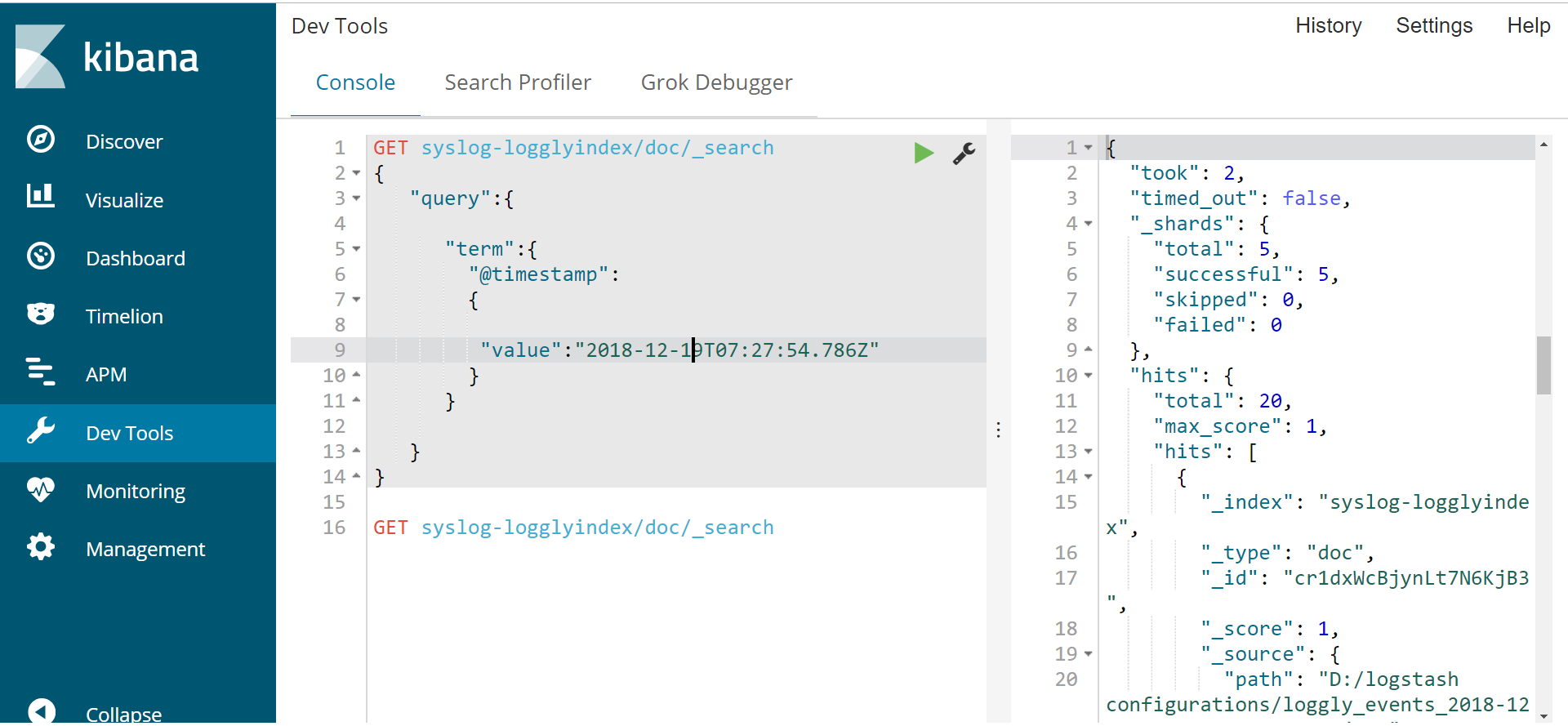
Term Level Queries



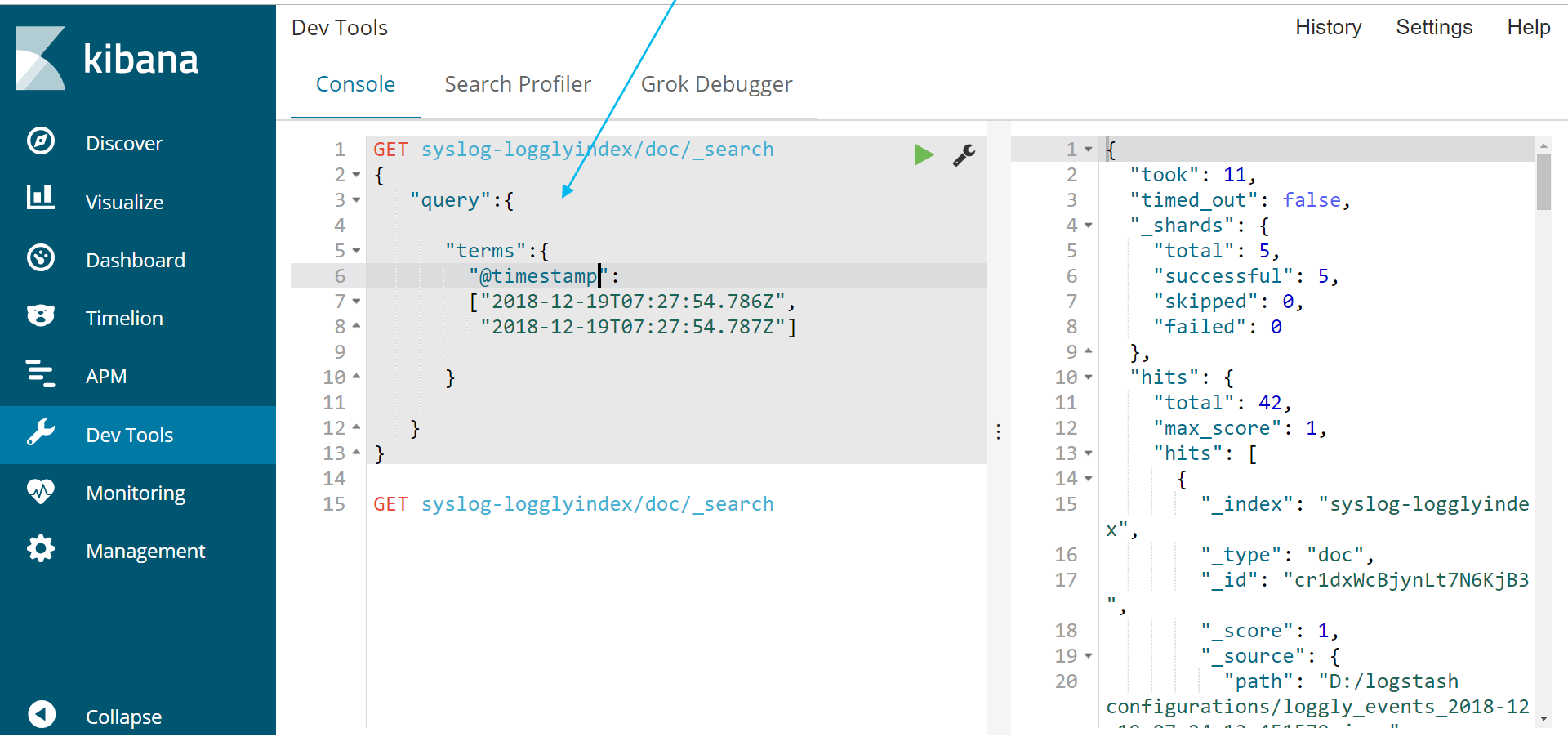
Term Level Queries



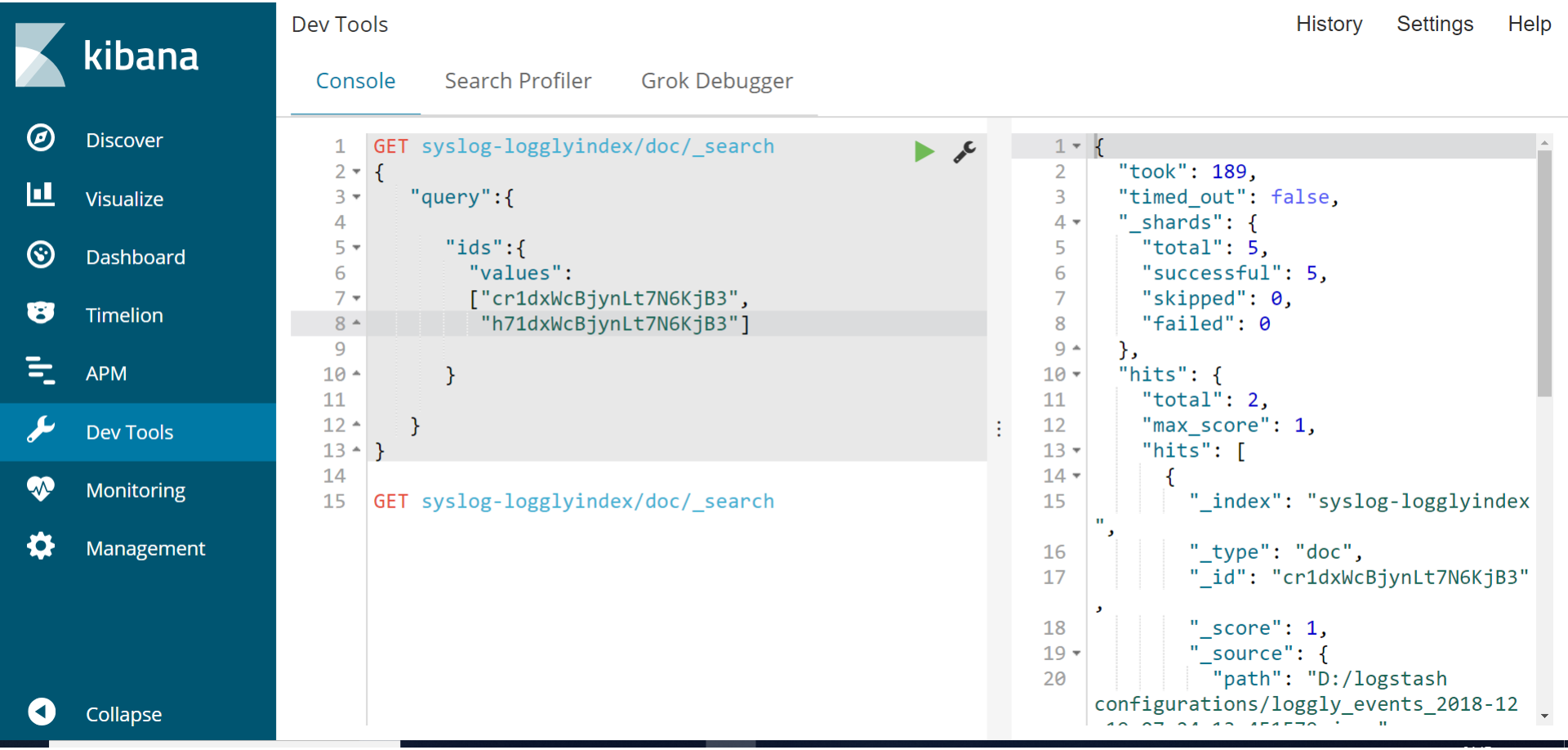
Term Level Queries



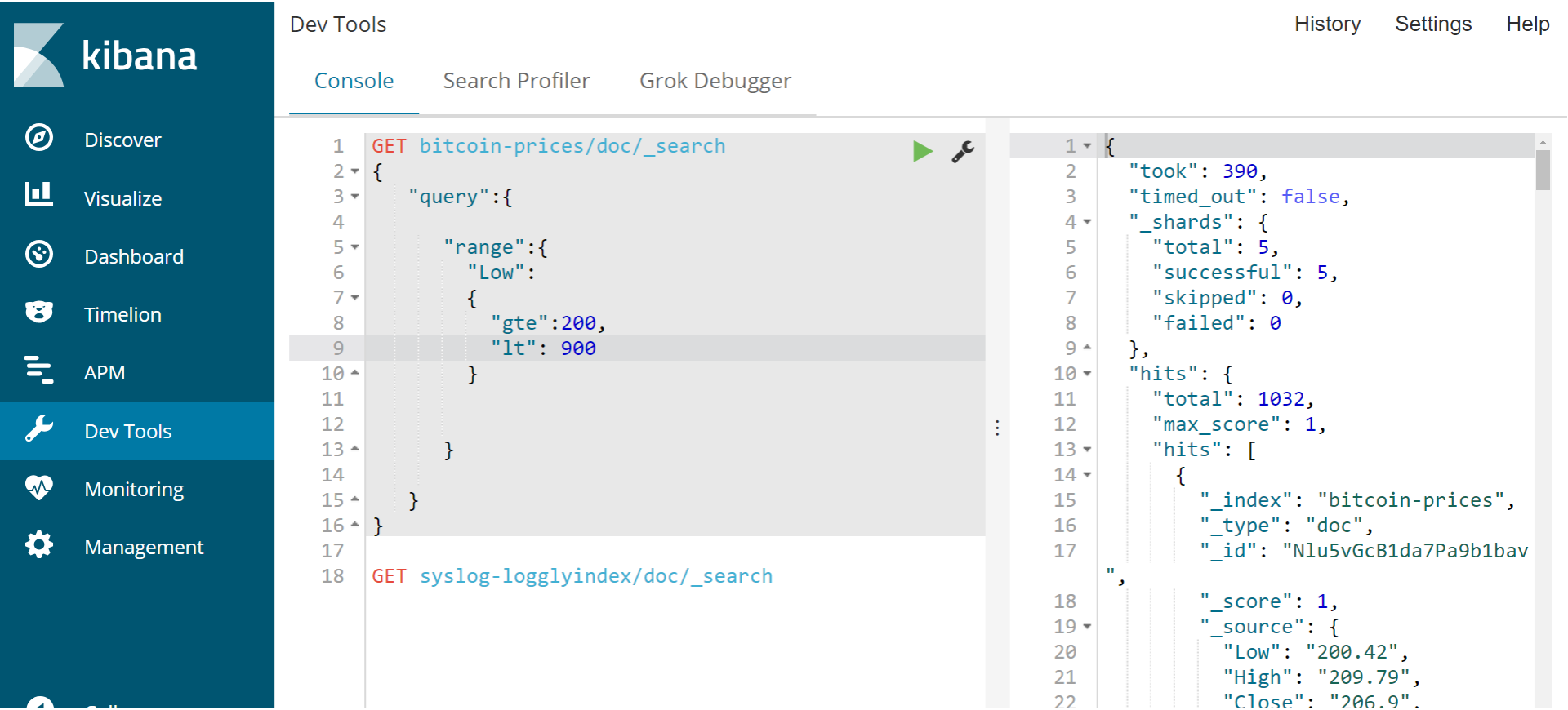
Term Level Queries



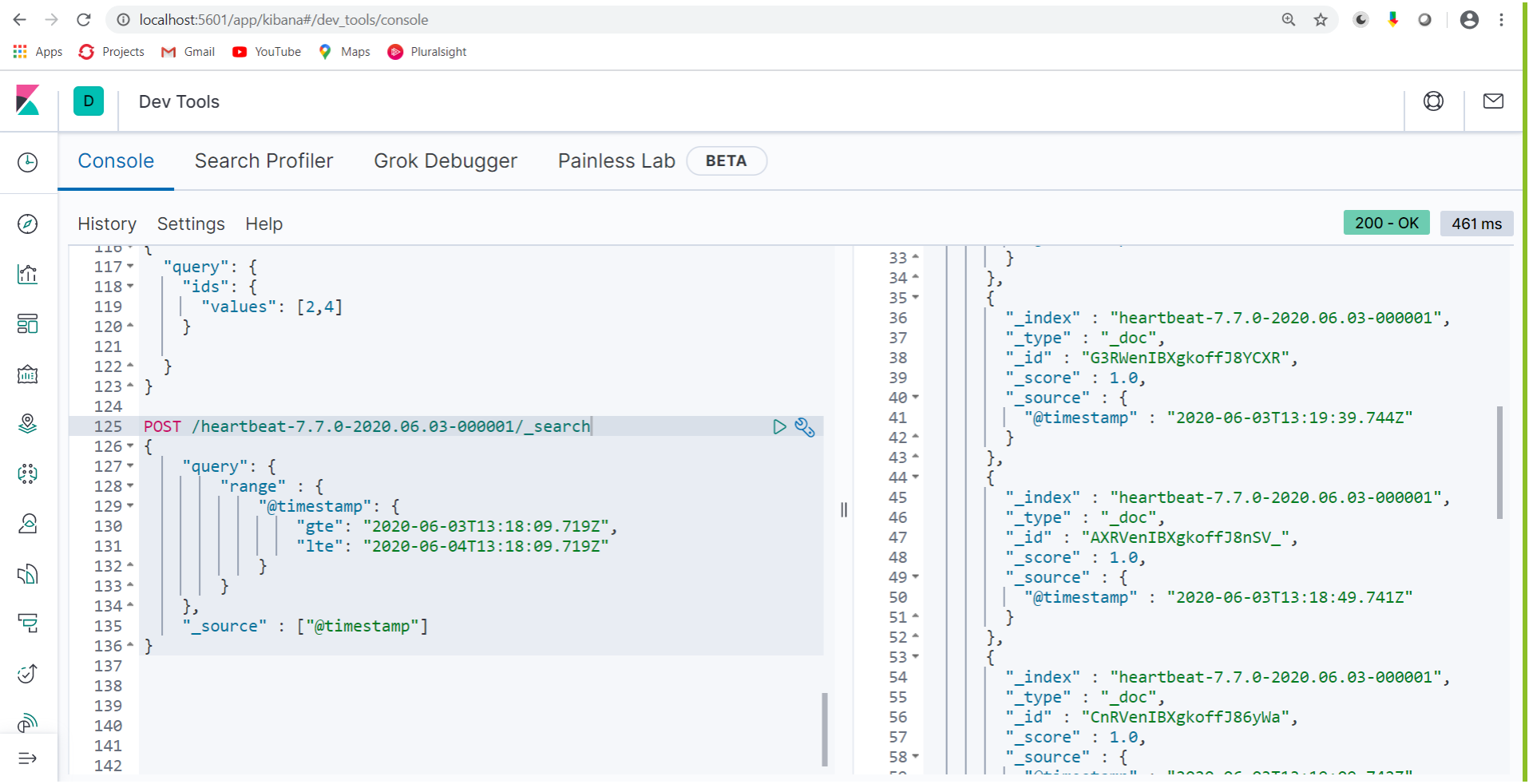
Retrieving Documents By Id



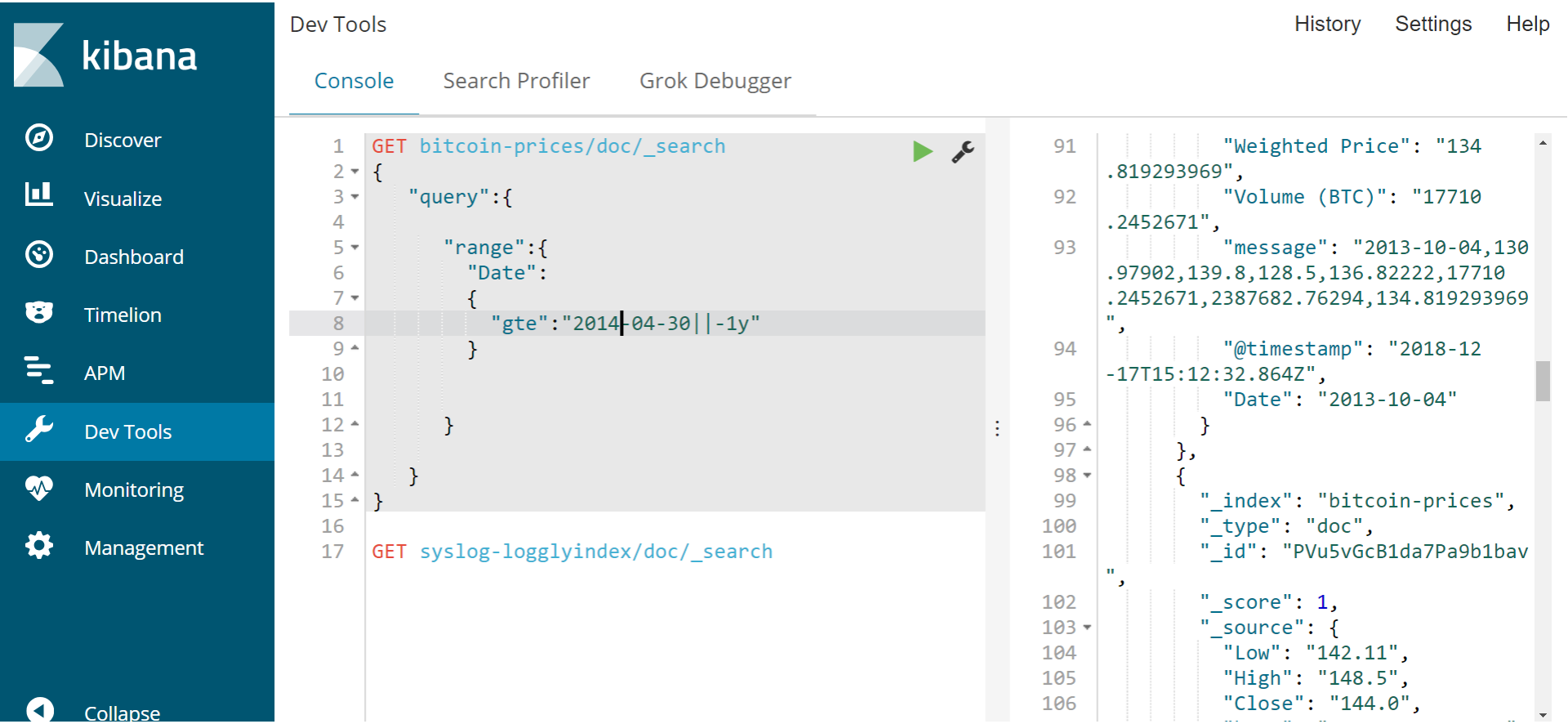
Matching Documents with Range Query



Matching Documents with Range Query



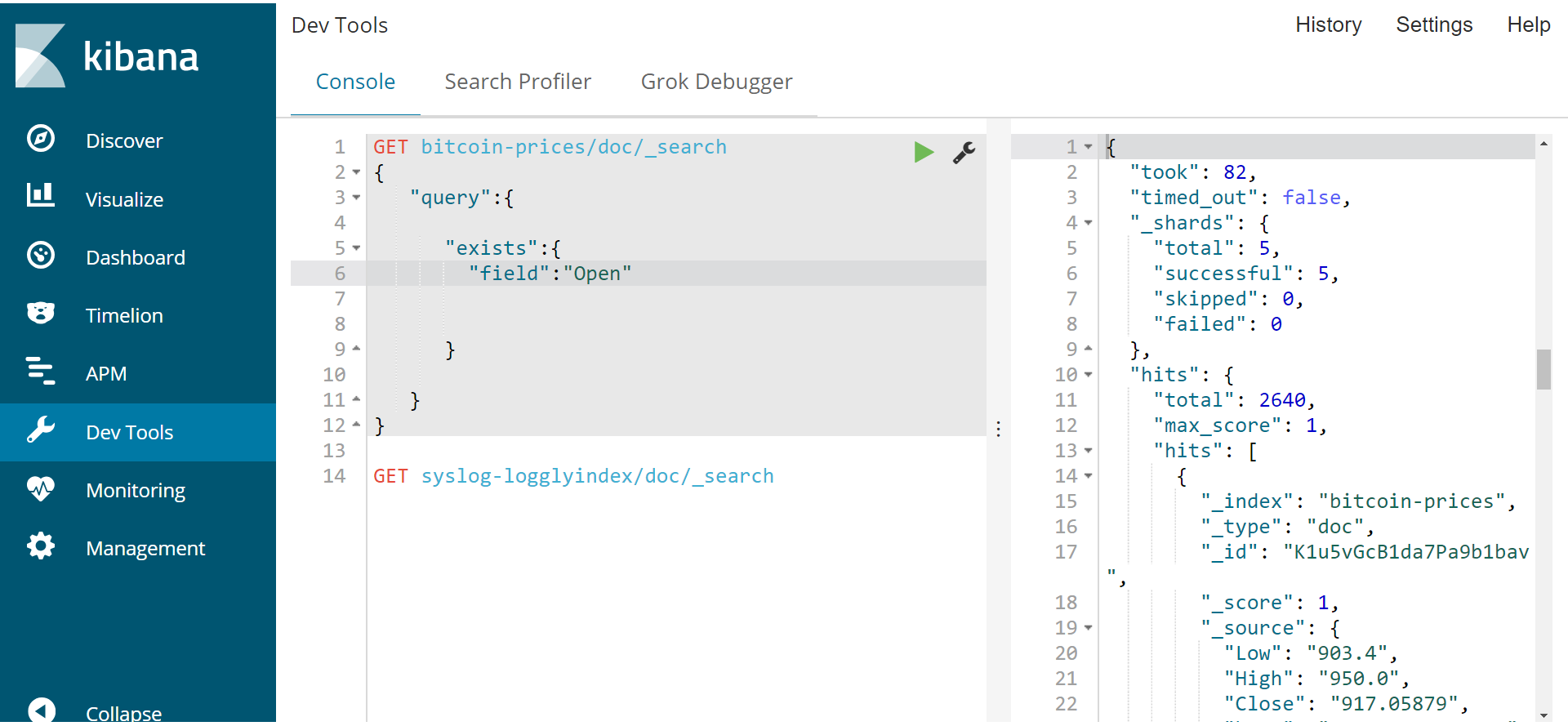
Working with Date Relevant Dates



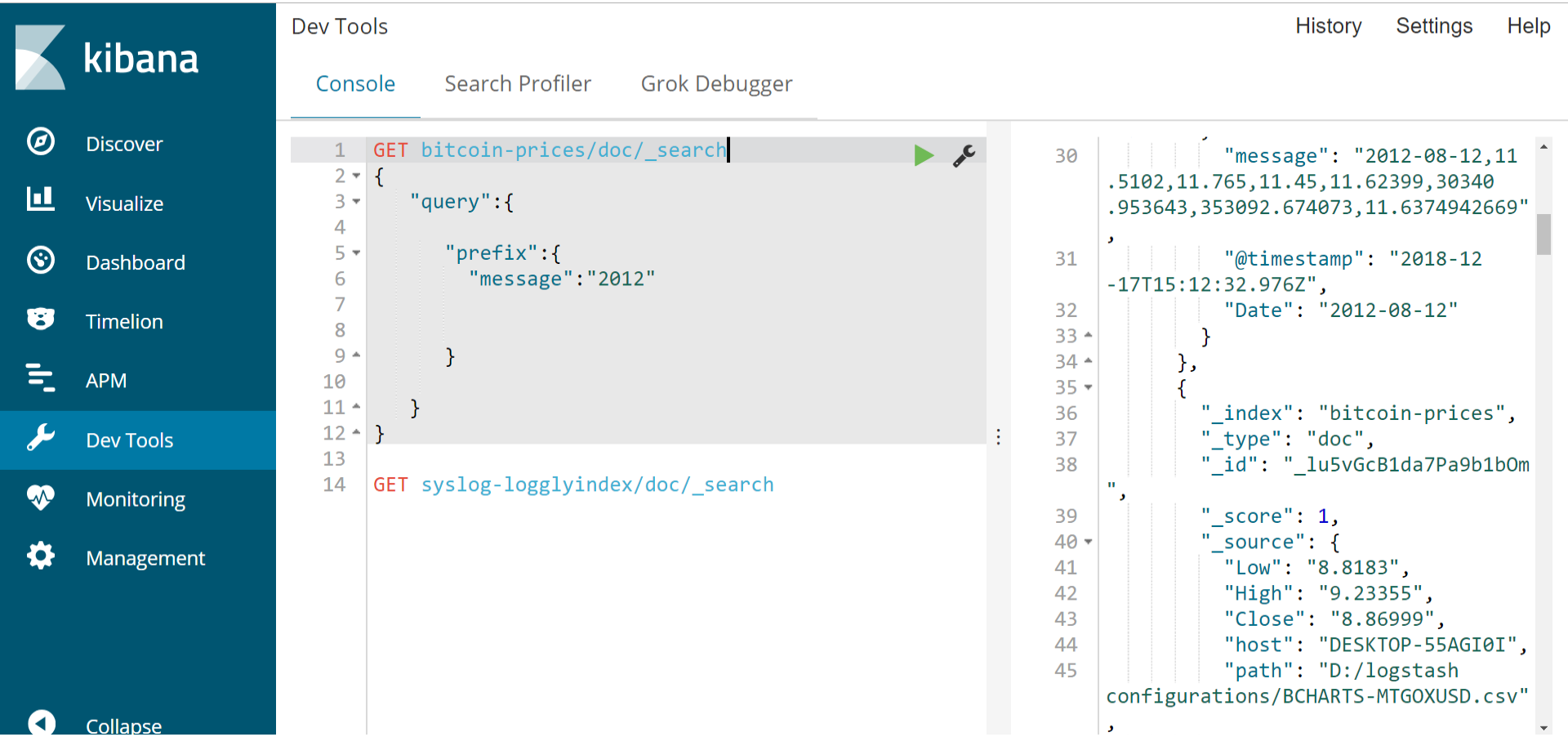
Working with Date Relevant Dates

Operator	Rounding direction	Before	After
gt	up	2010-01-20	2010-01-31
gte	down	2010-01-20	2010-01-01
lt	down	2010-01-20	2010-01-01
lte	up	2010-01-20	2010-01-31

Working with non null values



Matching on Prefixes



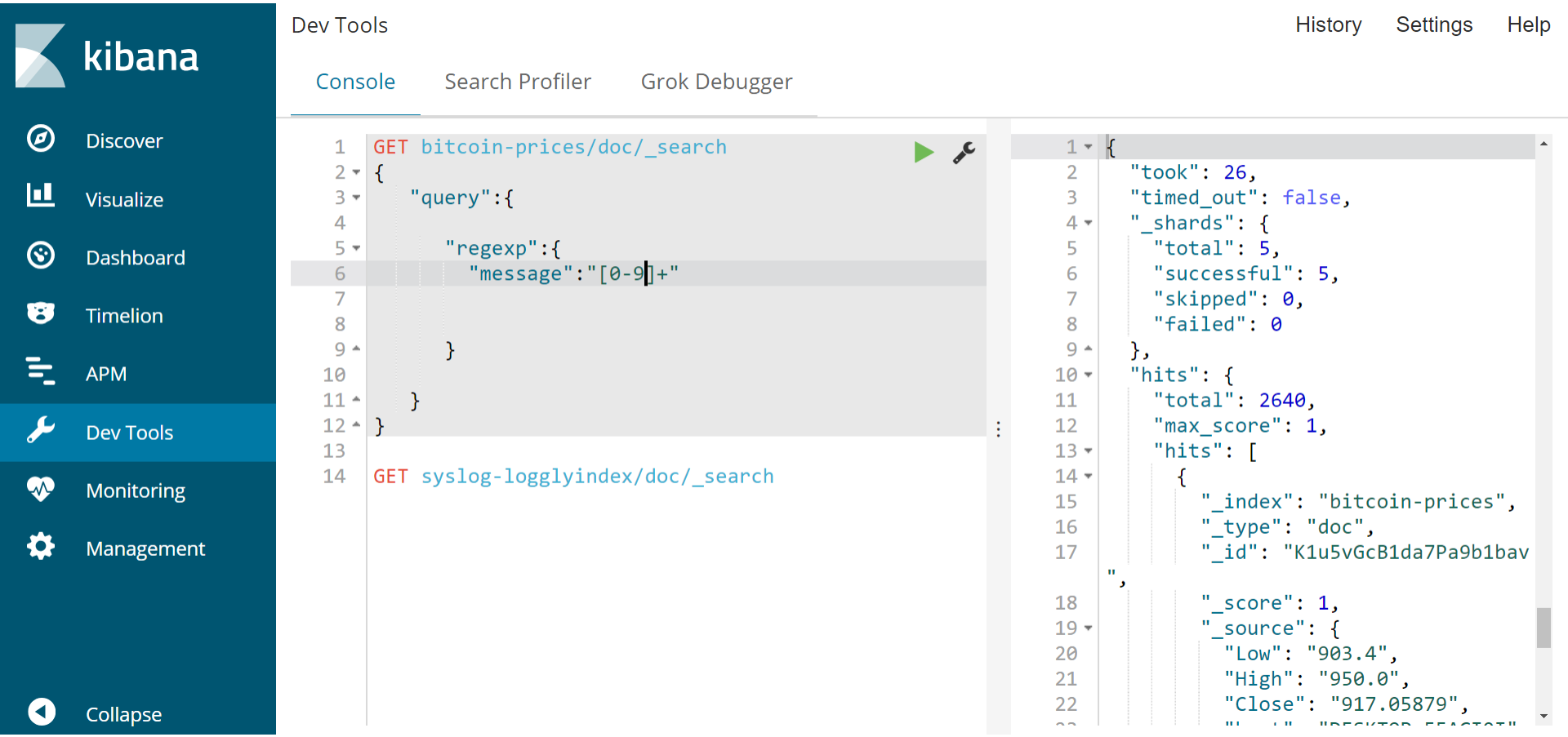
Searching with wild cards

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

GET /products/_search
{
"query": {
"wildcard": {
  "name.keyword": {
    "value": "N*"
  }
}
}
}
```

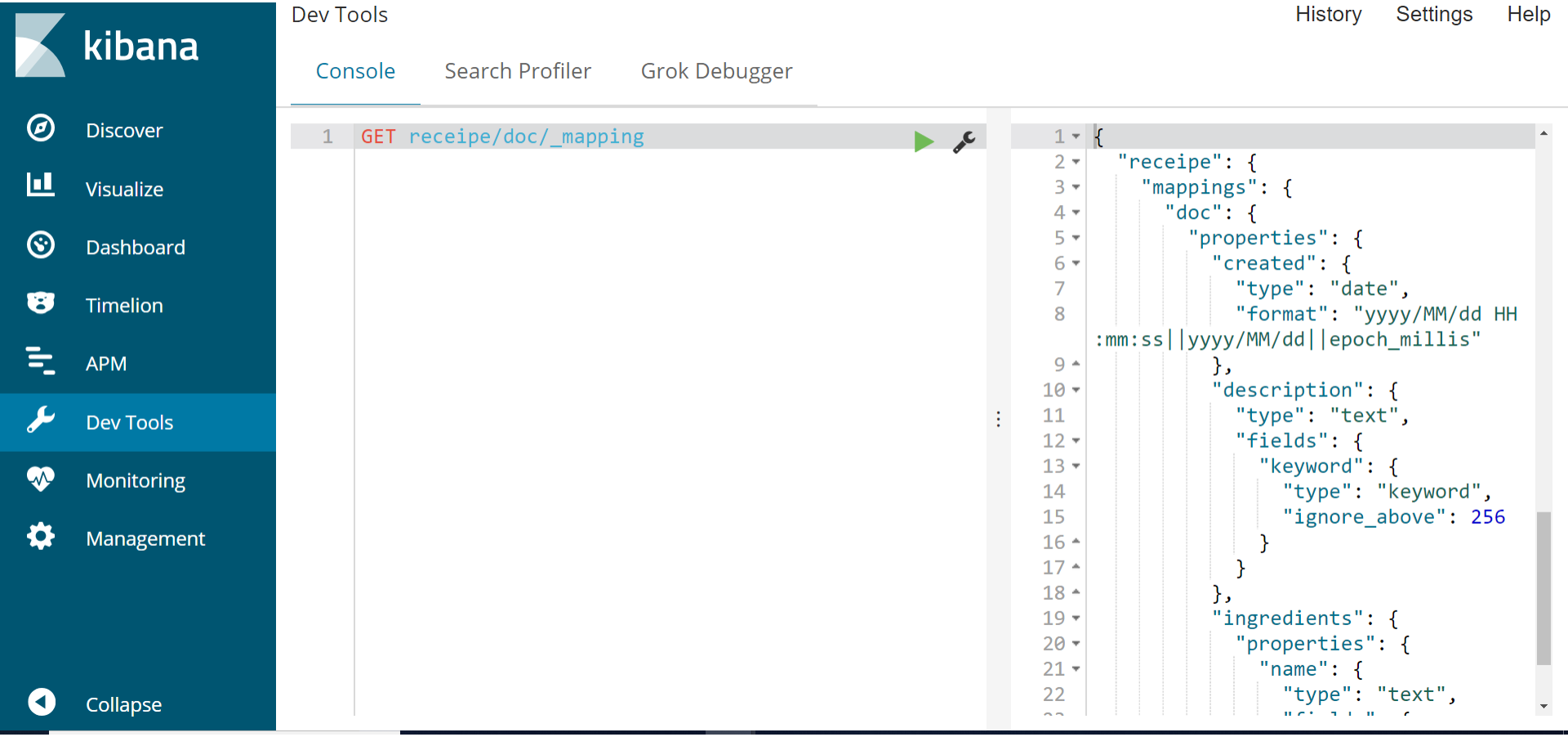
Click to send request

```
      "is_active" : true,
      "created" : "2011/11/06"
    }
  },
  {
    "_index" : "products",
    "_type" : "doc",
    "_id" : "14",
    "_score" : 1.0,
    "_ignored" : [
      "description.keyword"
    ],
    "_source" : {
      "name" : "Nori Sea Weed - Gold Label",
      "price" : 177,
      "in_stock" : 21,
      "sold" : 411,
      "tags" : [
        "Seafood"
      ],
```

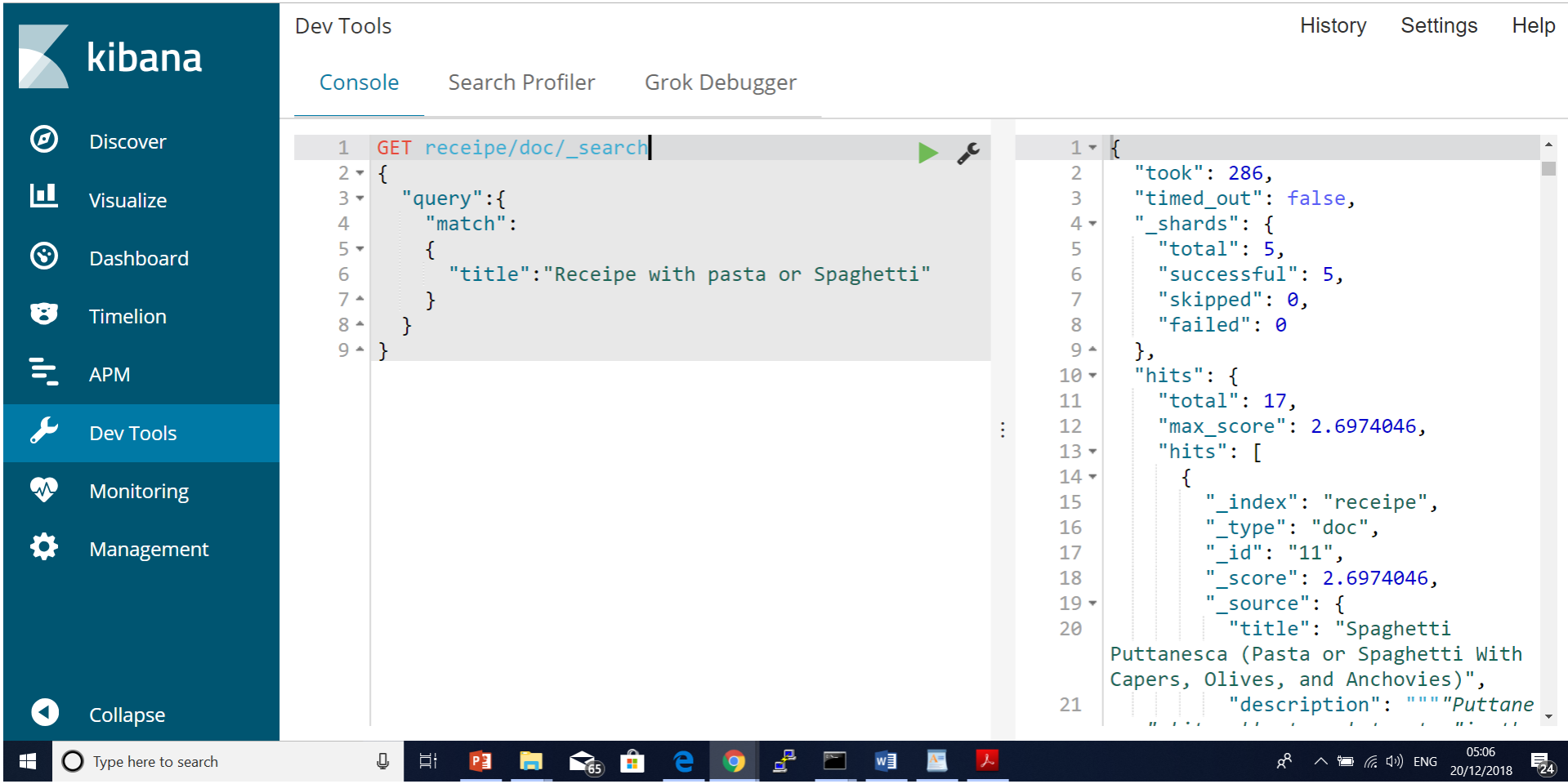
Searching with Regular Expressions



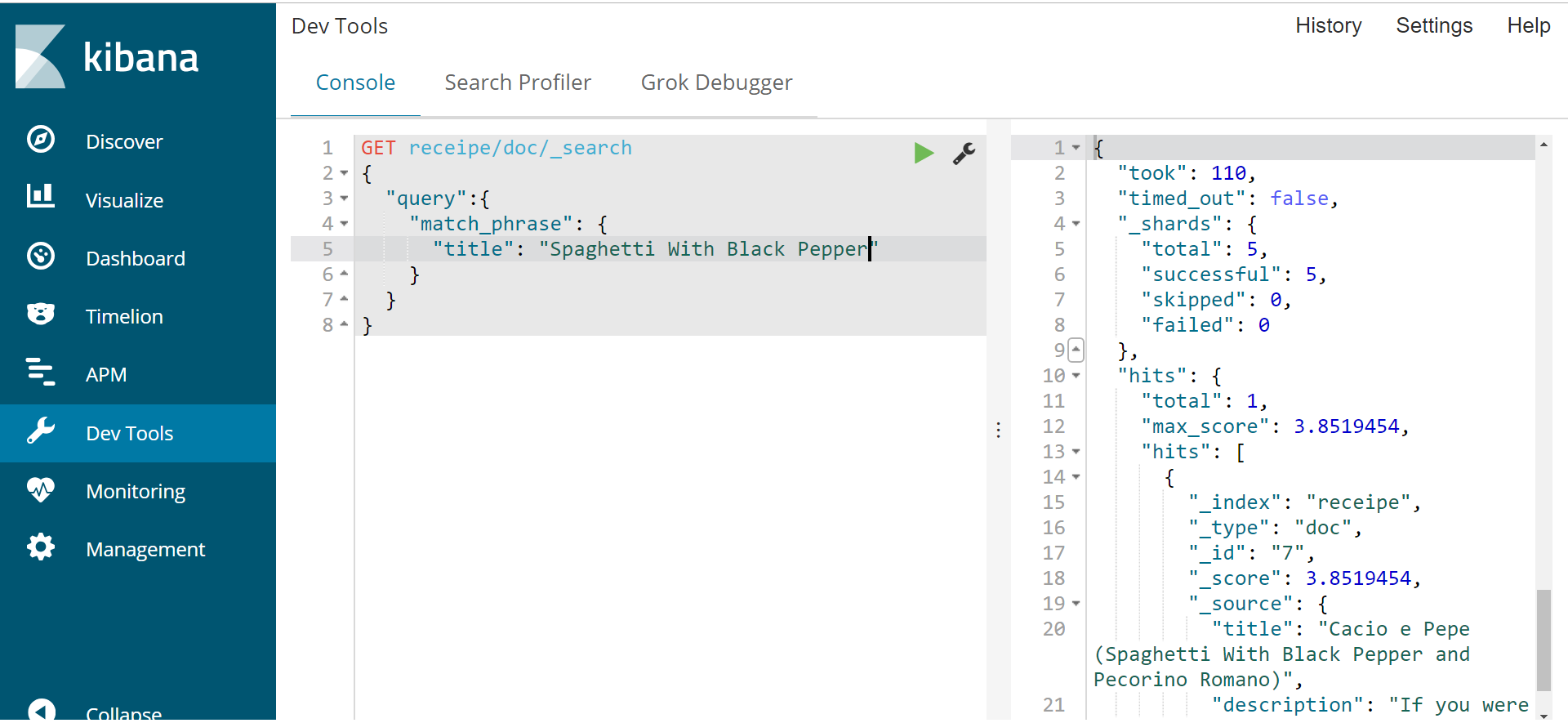
Full Text Queries



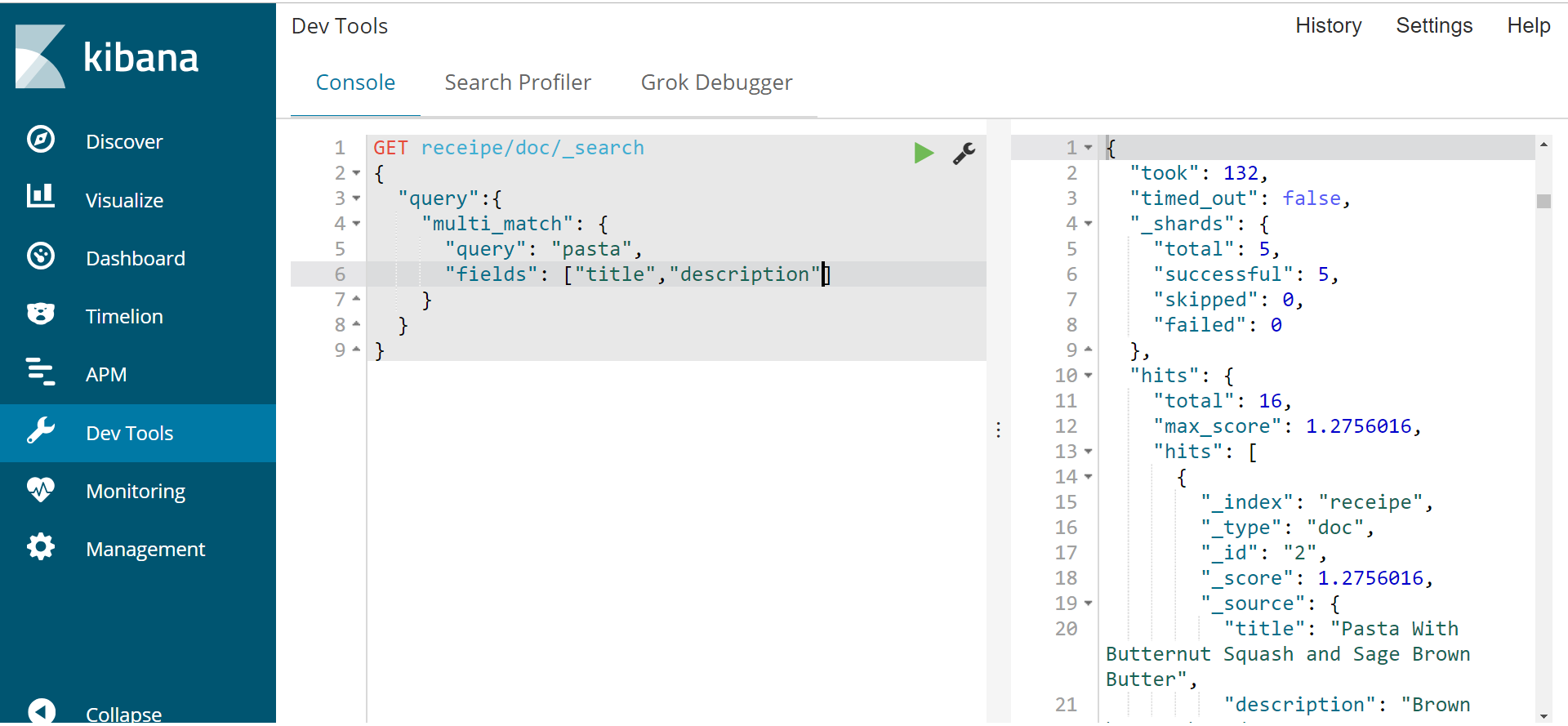
Flexi Queries



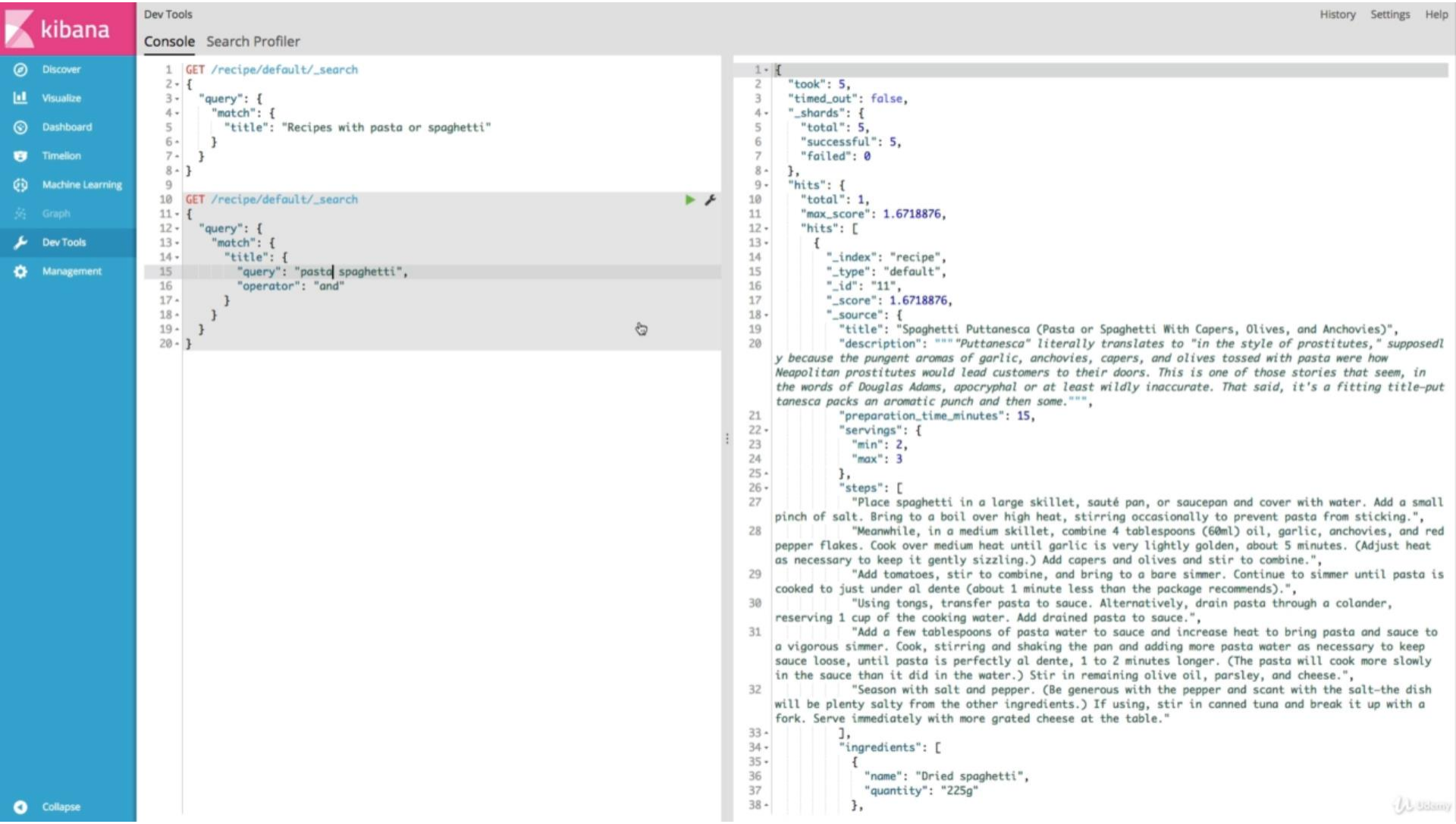
Phrase Query



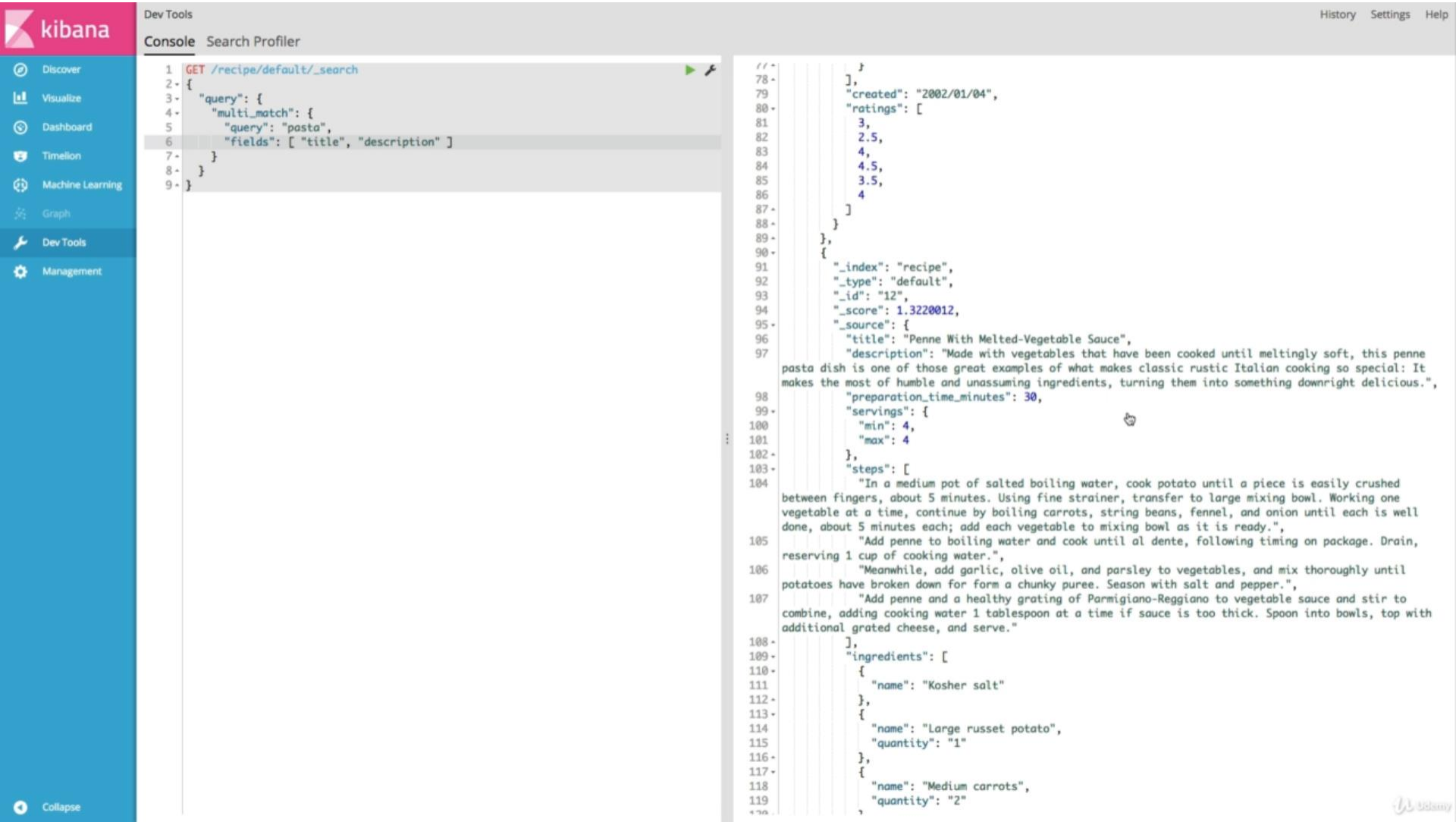
Multi Field Query



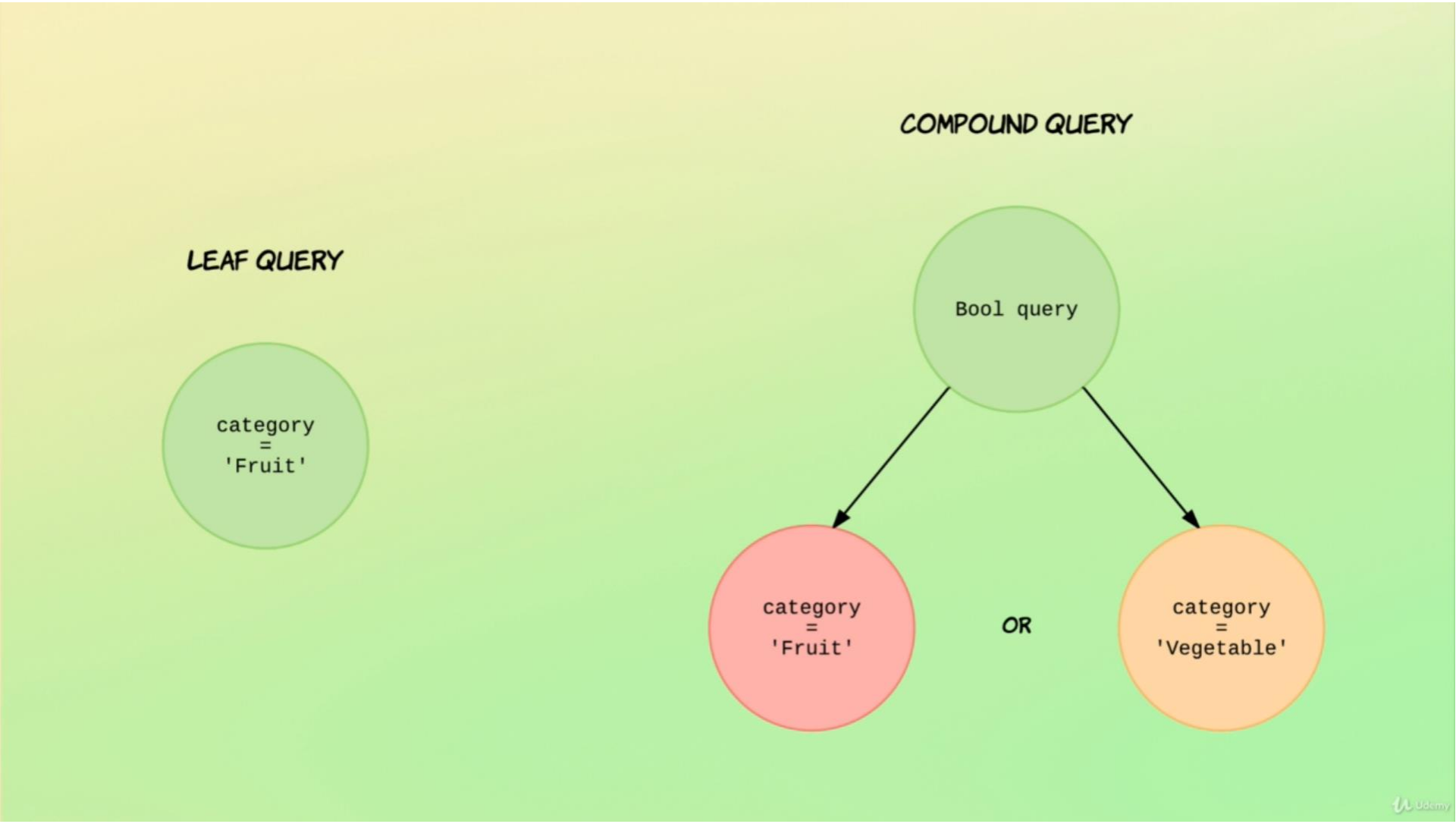
Managing Documents Full Text Query



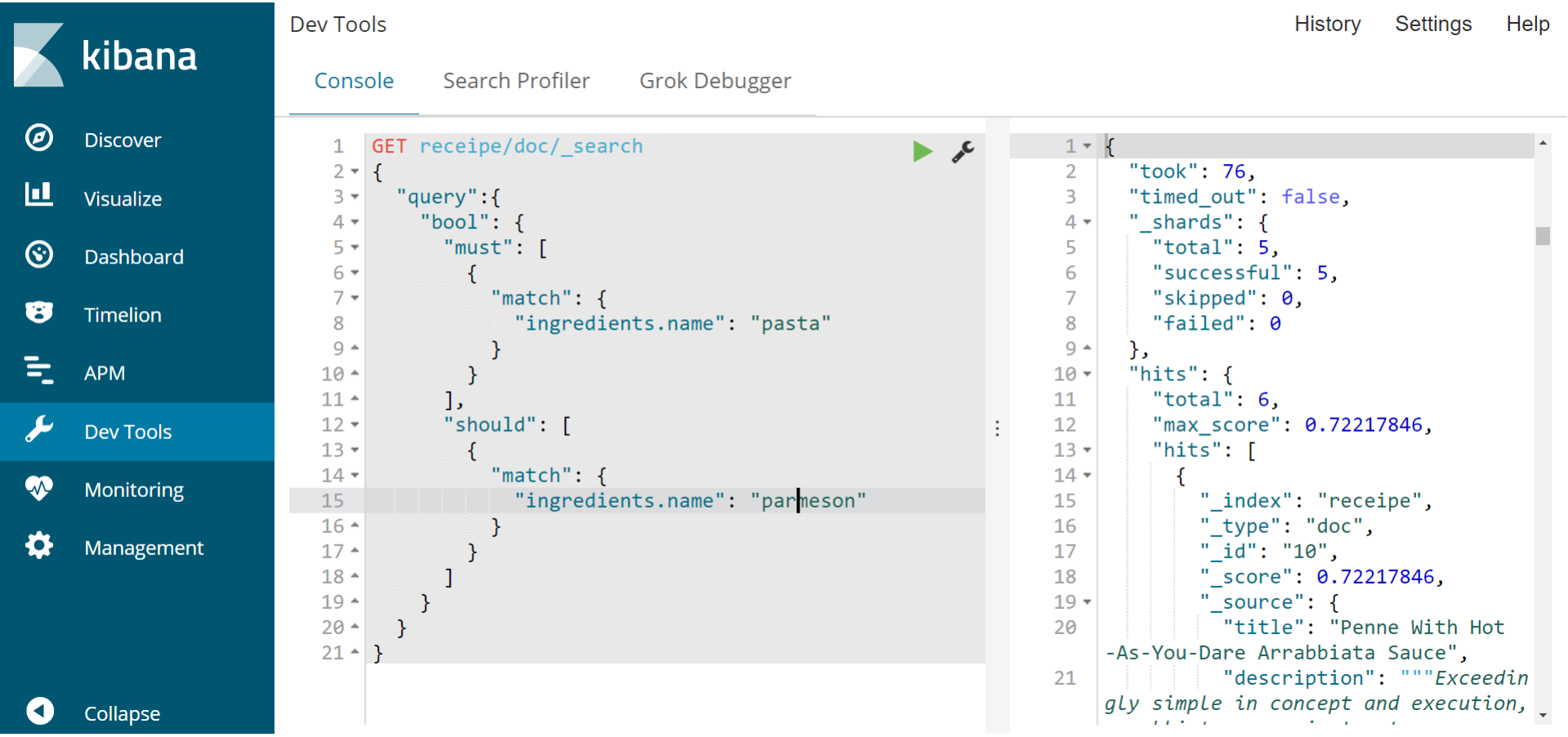
Managing Documents multiple fields



Compound Queries



Compound Queries



Document

Doc ID	Content
1	This is a brown fox
2	This is a brown dog
3	This dog is really brown
4	The dog is brown but this document is very very long
5	There is also a white cat
6	The quick brown fox jumps over the lazy dog

General Query

```
"query": {
    "match": {
        "content": {
            "query": "quick brown dog"
        }
    }
}
```

Document

Pos	Doc ID	Content	Score
1	6	The quick brown fox jumps over the lazy dog	0.81502354
2	2	This is a brown dog	0.26816052
3	3	This dog is really brown	0.26816052
4	4	The dog is brown but this document is very very long	0.15323459
5	1	This is a brown fox	0.055916067

Query

```
{
    "query": {
        "match": {
            "content": {
                "query": "quick brown dog",
                "minimum_should_match": 75%
            }
        }
    }
}
```

Phrase Query

```
"query": {
    "match_phrase": {
        "content": "brown dog"
    }
}
}
```

Proximity Query

```
{
    "query": {
        "match_phrase": {
            "content": {
                "query": "brown dog",
                "slop": 3
            }
        }
    }
}
```

Result

Pos	Doc ID	Content	Score
1	2	This is a brown dog	0.9547657
2	4	The dog is brown but this document is very very long	0.2727902

Query

```
{
    "query": {
        "match_phrase": {
            "content": {
                "query": "brown dog",
                "slop": 4
            }
        }
    }
}
```

Result

Pos	Doc ID	Content	Score
1	2	This is a brown dog	0.9547657
2	3	This dog is really brown	0.4269842
3	4	The dog is brown but this document is very very long	0.2727902

Proximity Search

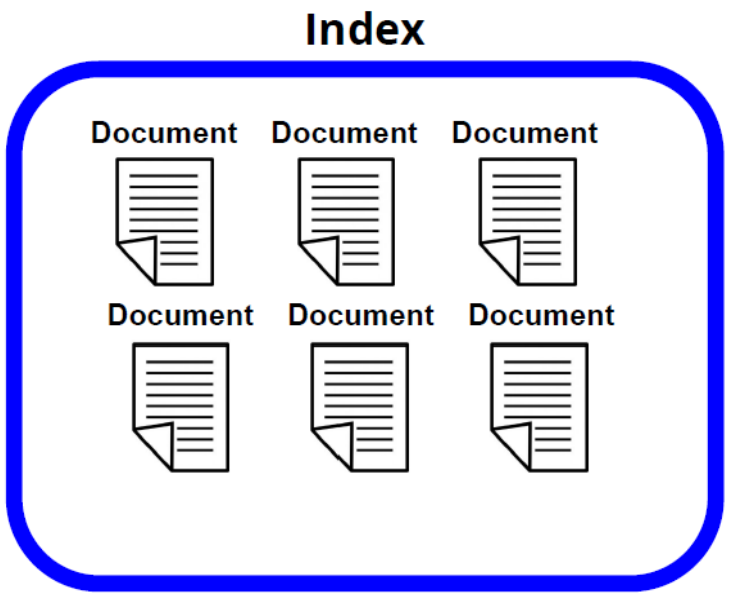
	Position 1	Position 2	Position 3
Document	tomato	sauce	spicy
Query	spicy	sauce	
Slop 1	→	spicy|sauce	
Slop 2		sauce	→ spicy

Proximity Search(Order Relevance – slop)

Relevance

- Precision
- Recall

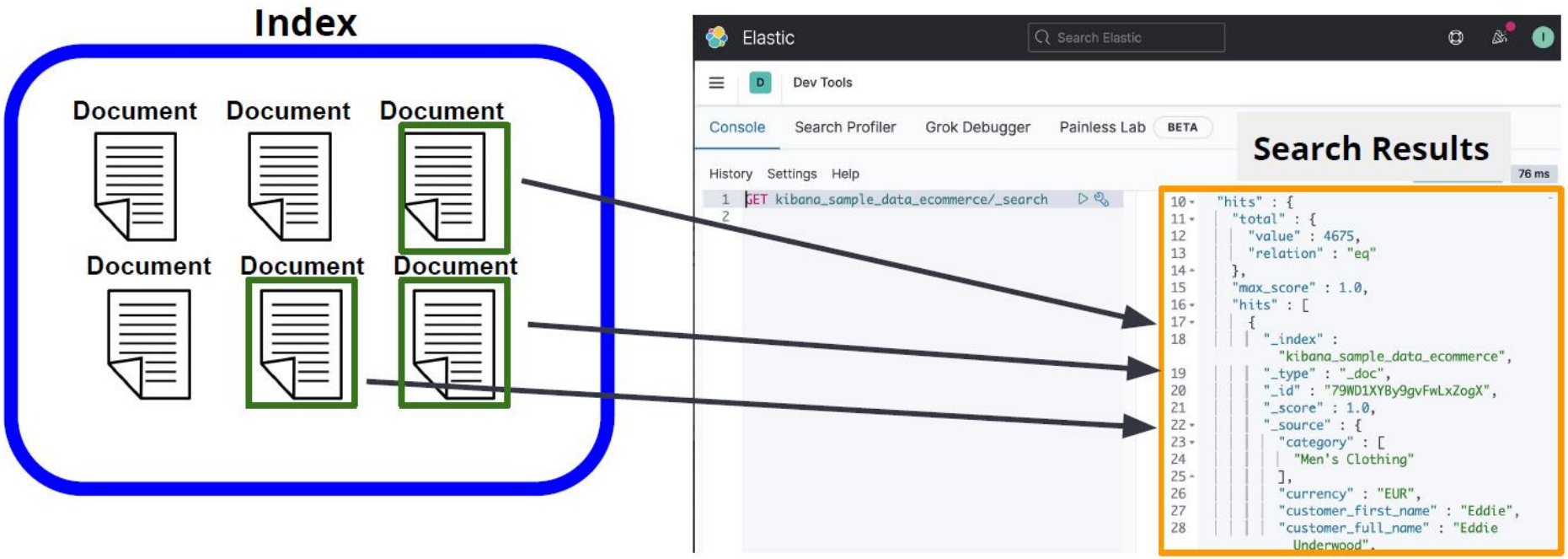
Relevance



Documents with similar traits are grouped into an index!

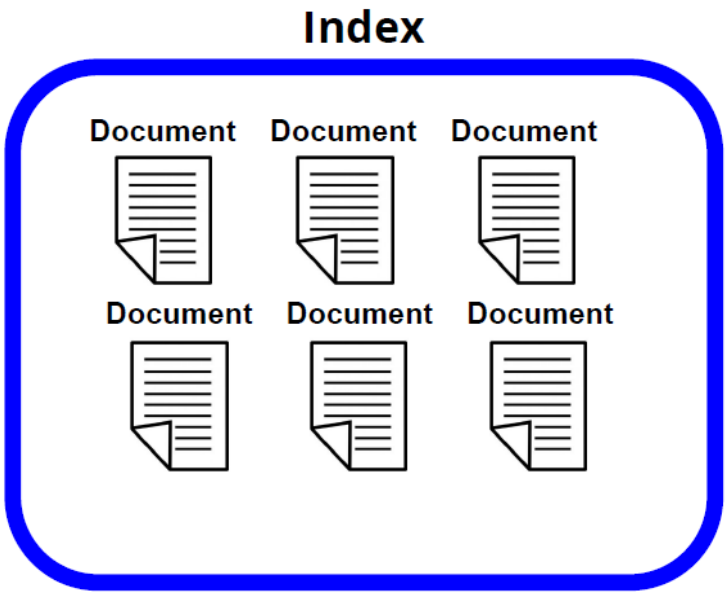
Relevance

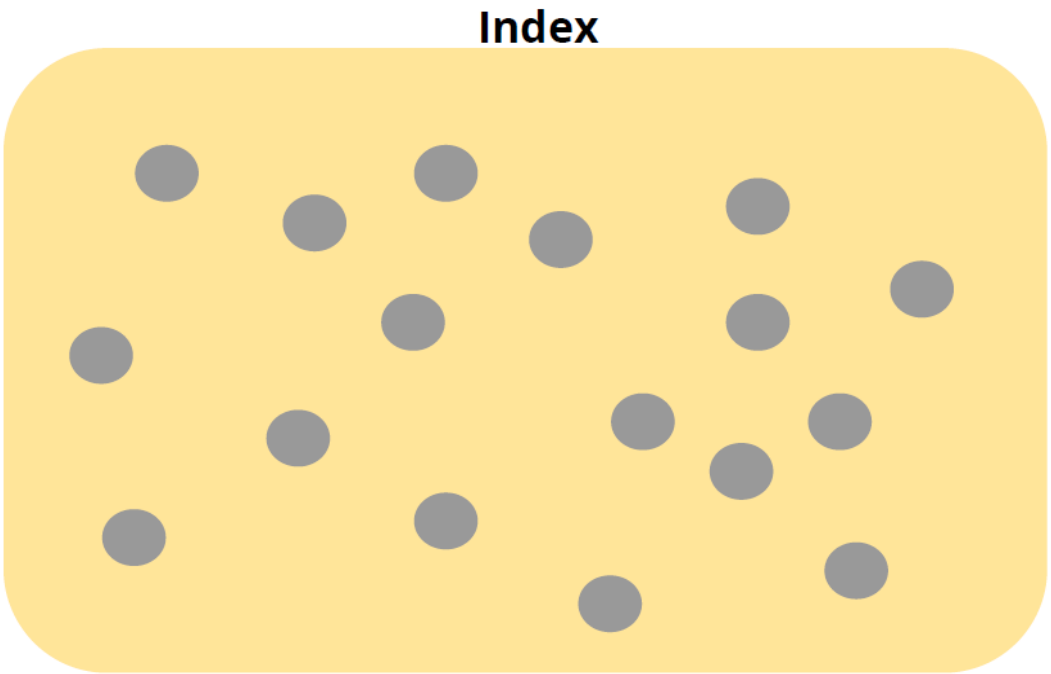
When search query is sent, Elasticsearch retrieves relevant documents and presents the documents as search results.



Relevance

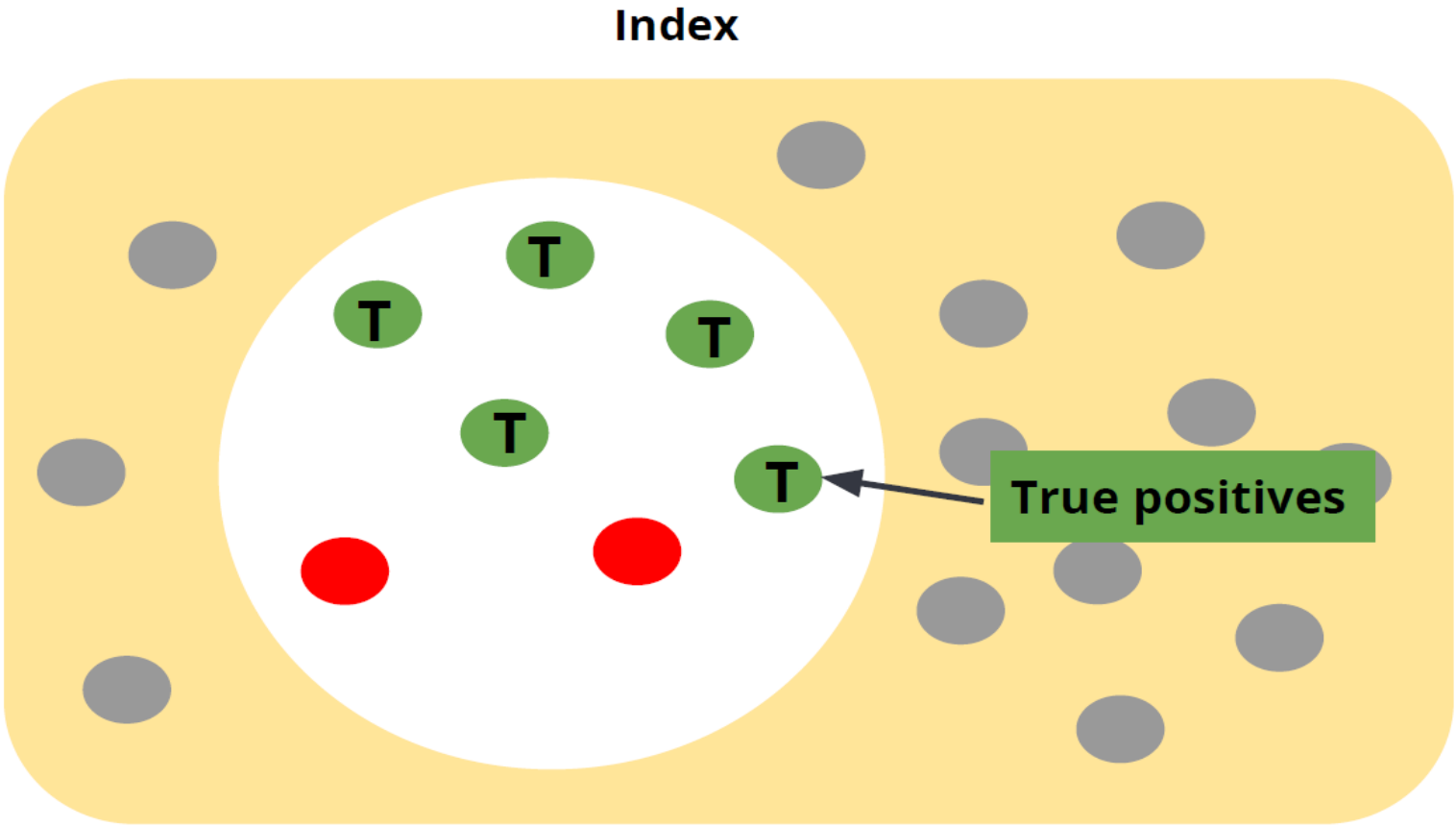
These two diagrams depict the same thing!





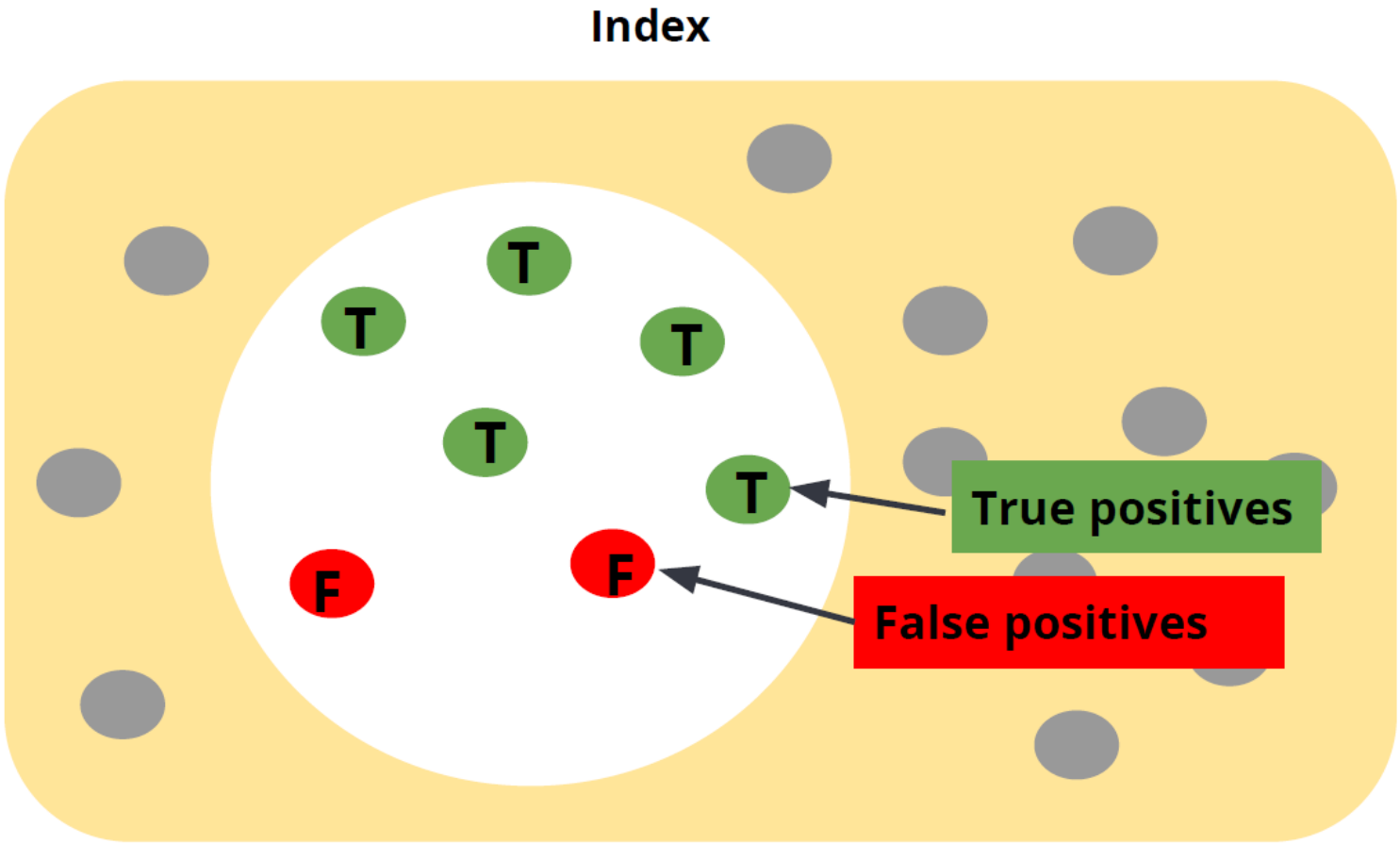
Relevance

True positives are relevant documents that are returned to the user.



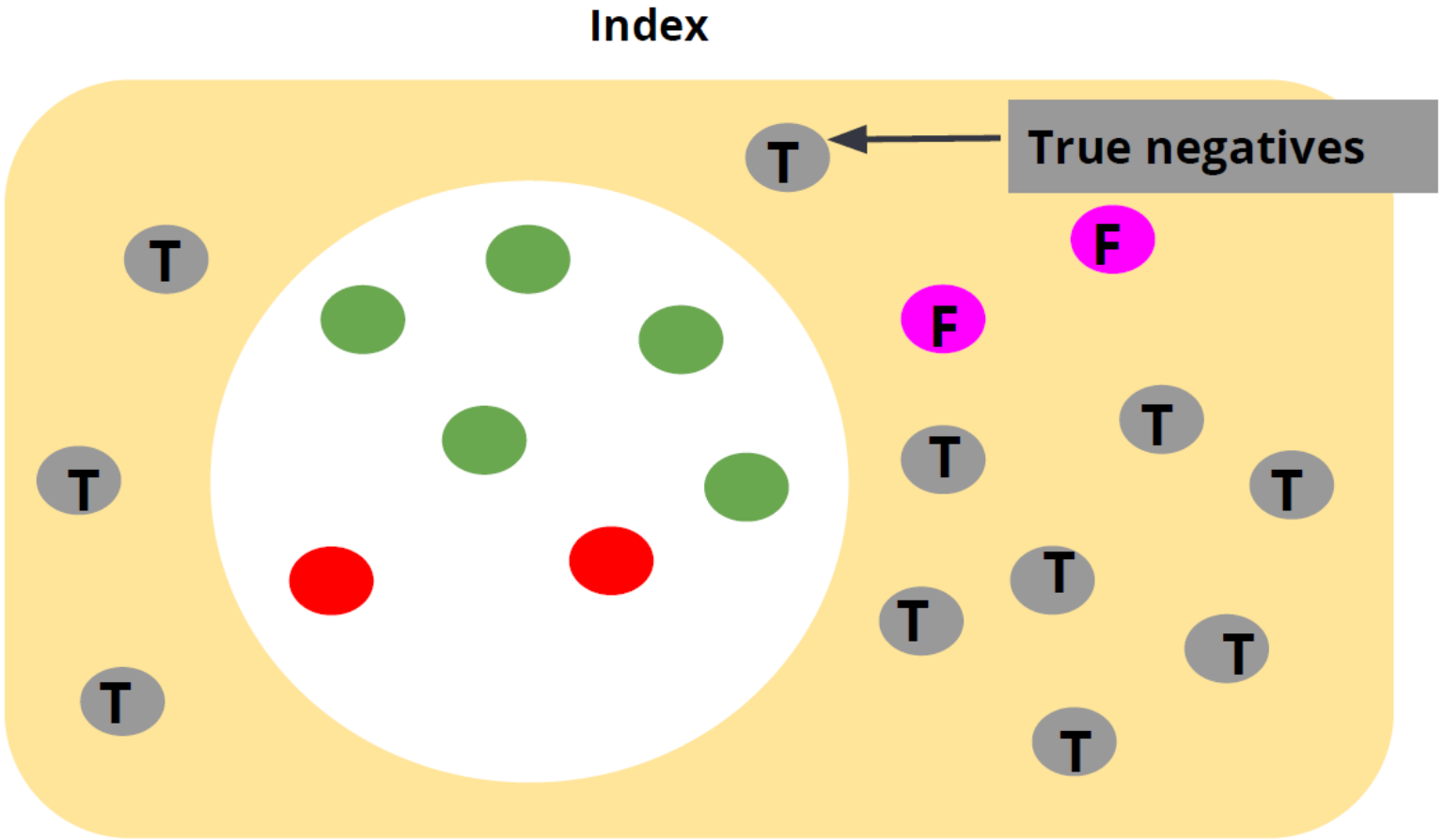
Relevance

False positives are irrelevant documents that are returned to the user.



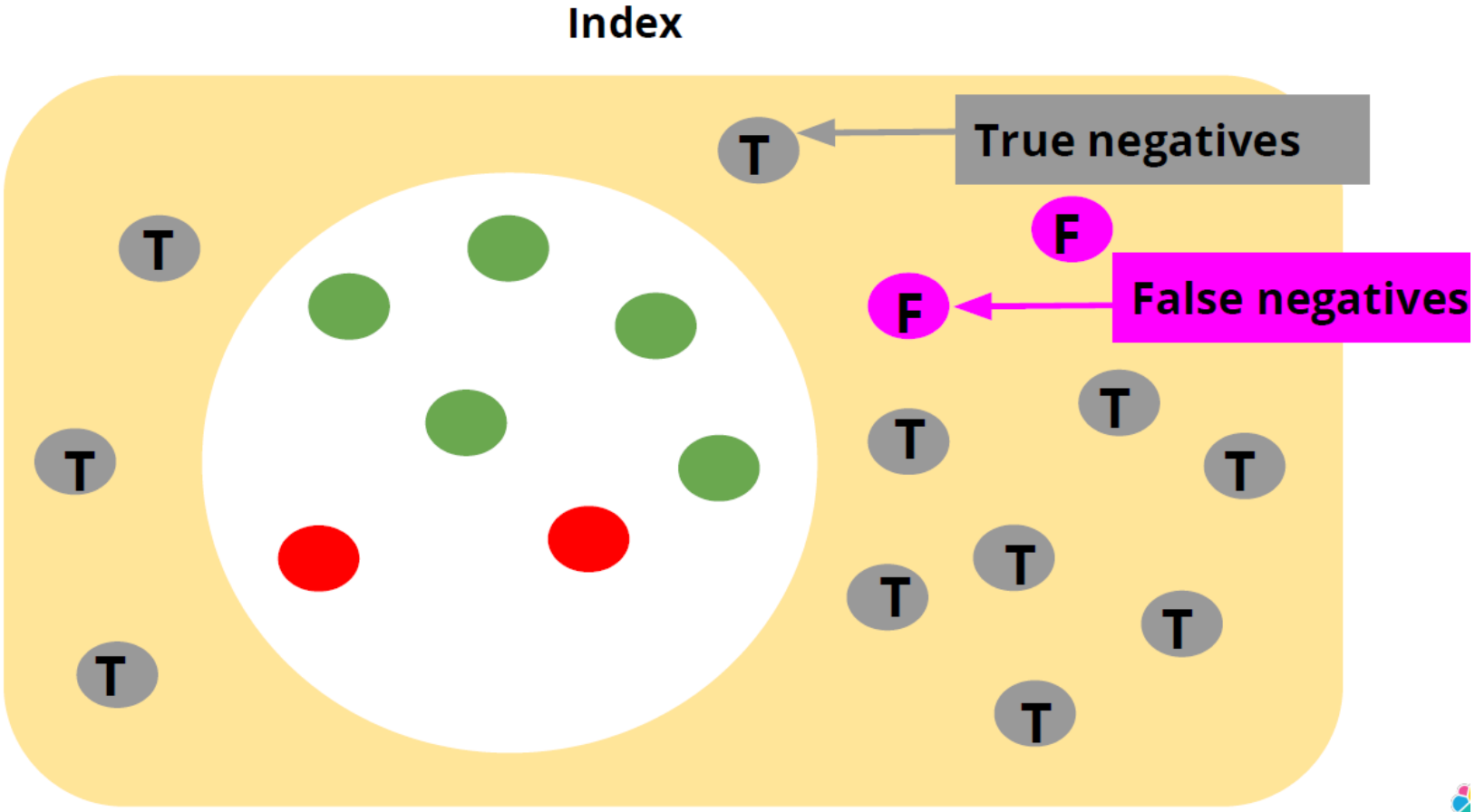
Relevance

True negatives are irrelevant documents that are not returned to the user.



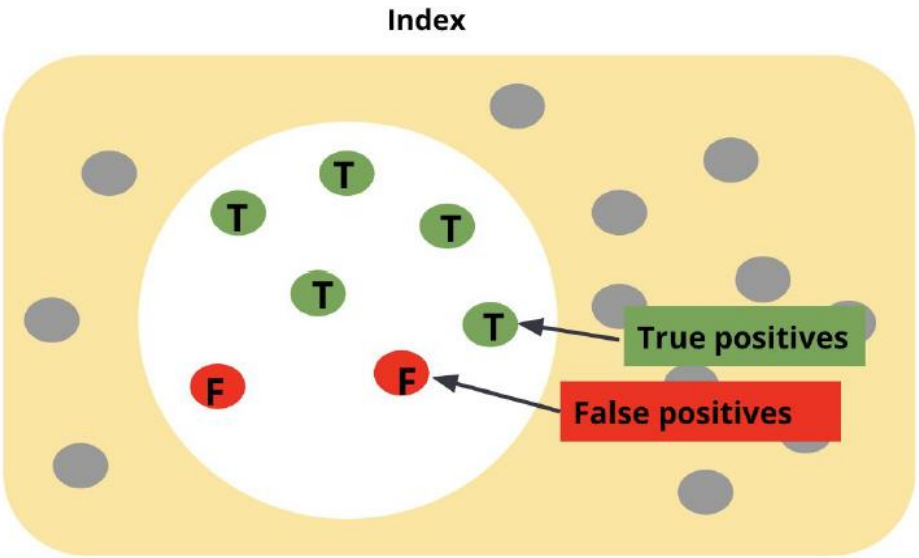
Relevance

False negatives are relevant documents that were not returned to the user.



Relevance

What is precision?

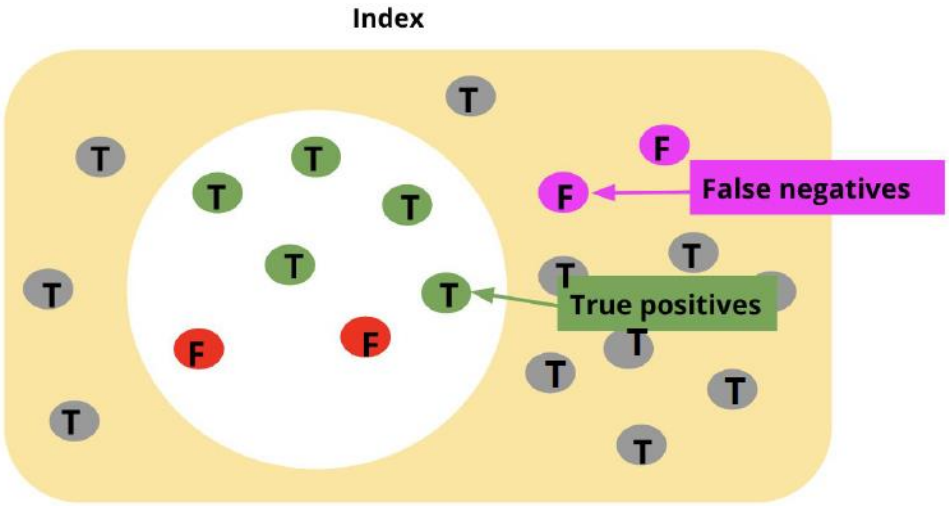


$$\text{Precision} = \frac{\text{True positives}}{\text{True positives} + \text{False positives}}$$

What portion of the retrieved data is actually relevant to the search query?

Relevance

What is recall?

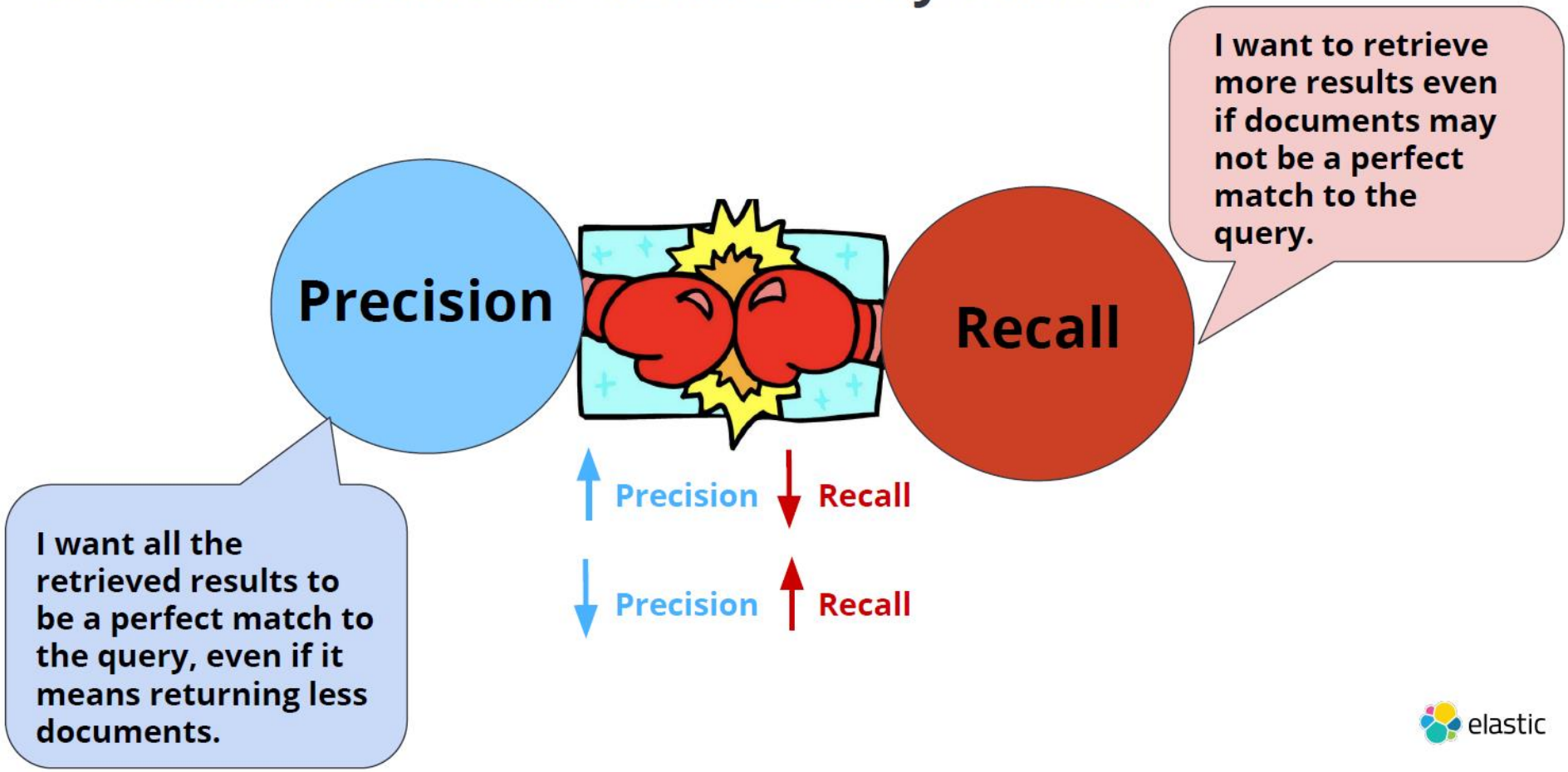


$$\text{Recall} = \frac{\text{True positives}}{\text{True positives} + \text{False negatives}}$$

What portion of relevant data is being returned as search results?

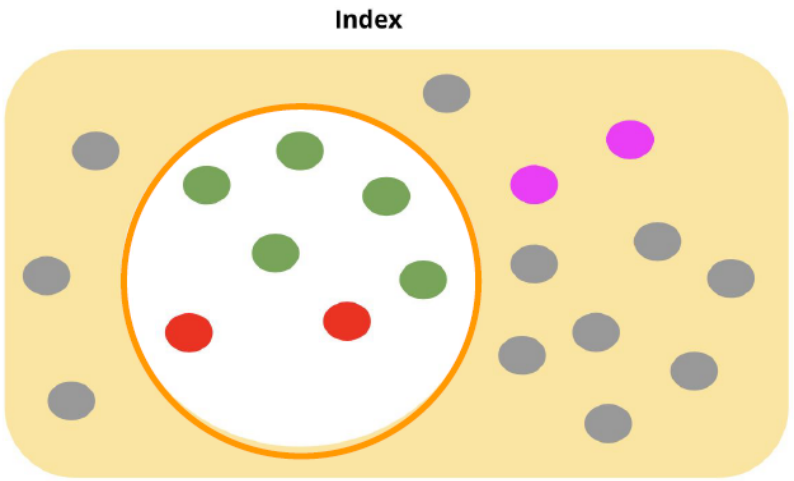
Relevance

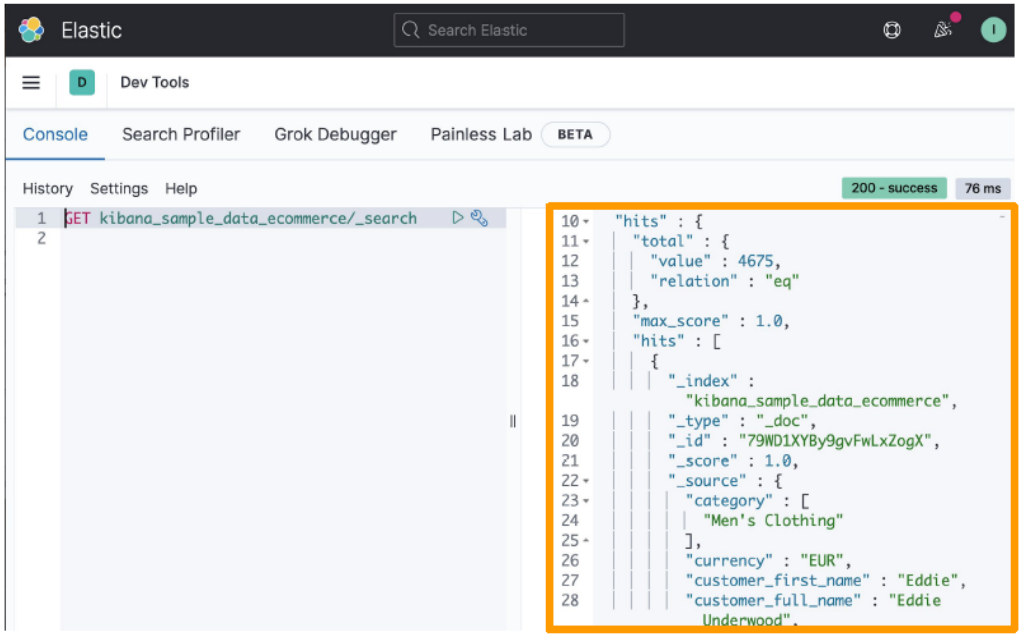
Precision and Recall are inversely related



Relevance

Precision and recall determine which documents are included in the search results.



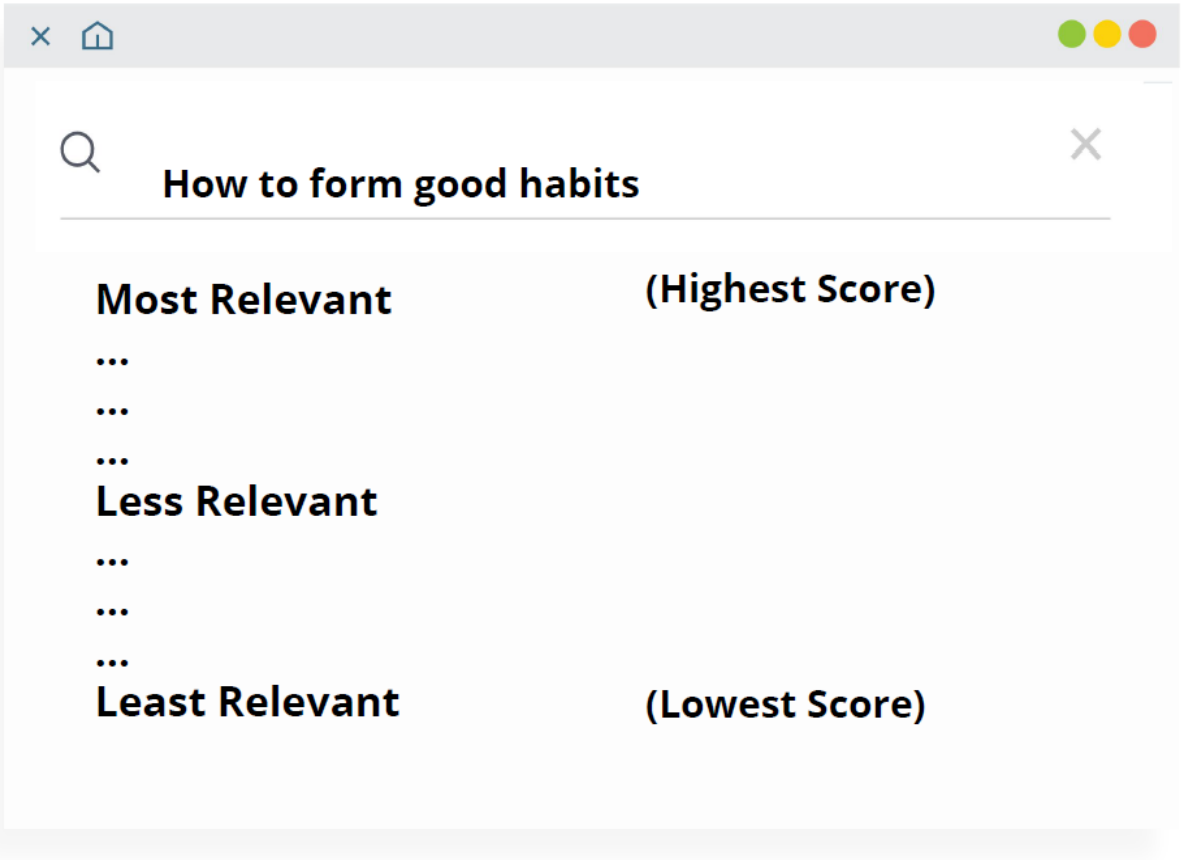


Precision and recall do not determine which of the returned documents are more relevant compared to the other!

elastic

Relevance

Ranking refers to ordering of the results (from most relevant results at the top, to least relevant at the bottom).



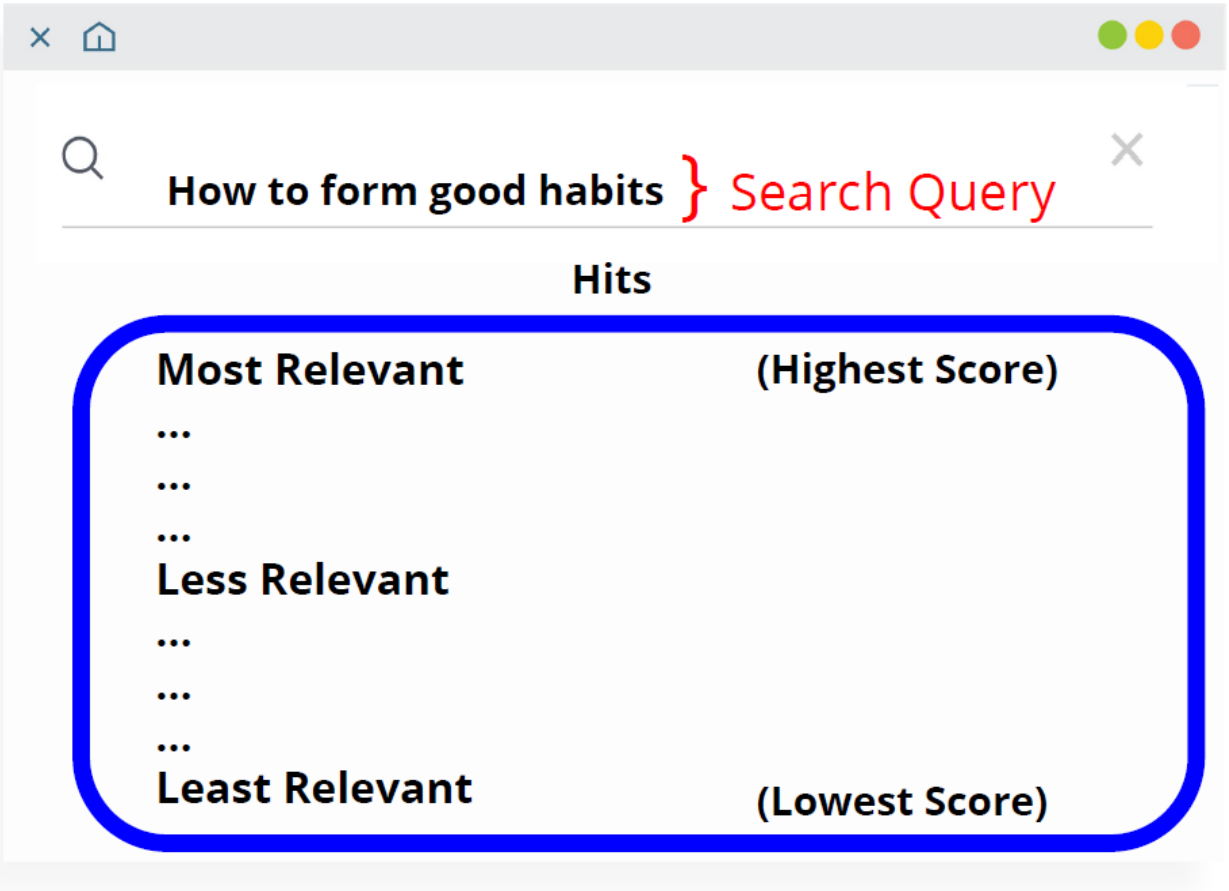
Score

- The score is a value that represents how relevant a document is to that specific query
- A score is computed for each document that is a hit

Score

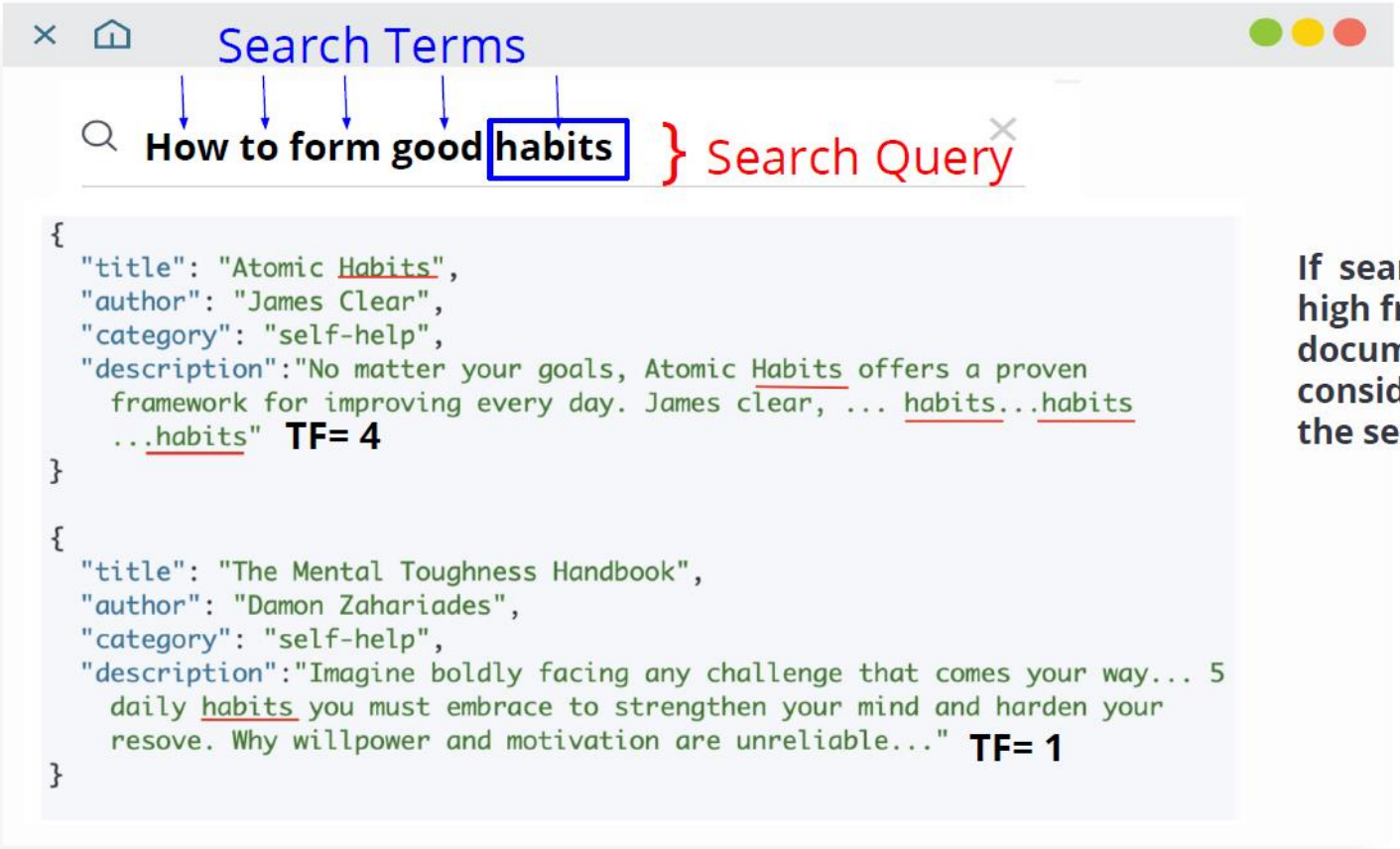
- Term Frequency(TF)
- Inverse Document Frequency(IDF)

Score


How to form good habits
Search Query
Hits
Most Relevant
(Highest Score)
...
...
...
Less Relevant
...
...
...
Least Relevant
(Lowest Score)

Score

Term Frequency(TF) determines how many times each search term appears in a document.

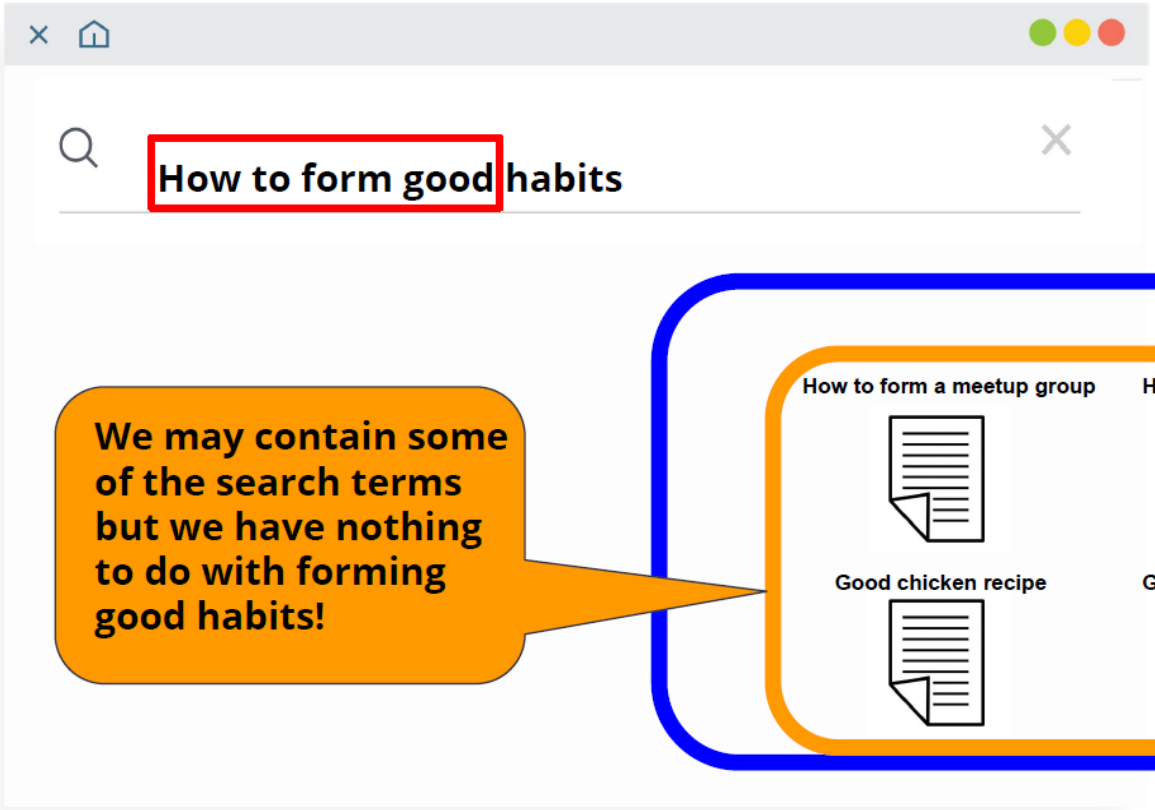


If search terms are found in high frequency in a document, the document is considered more relevant to the search query.

elastic

Score

What is Inverse Document Frequency(IDF)?



IDF diminishes the weight of terms that occur very frequently in the document set and increases the weight of terms that occur rarely!

elastic

Function Score Query, using Field Value Factor

- Predefined functions supported by Function Score:
  - weight: Apply a single enlargement to each document without the enlargement being normalized.
  - field_value_factor: Use the value of a field in the document to change the _score.
  - random_score: Use consistent random score to rank results differently for each user.
  - decay functions: Score a document with a function that decays depending on the distance of a numeric field value in the document from a user-determined source.
  - script_score: Use a custom script to take full control of the scoring logic.

Function Score Query, using Field Value Factor

- max_boost: limit the maximum effect the function can have.
- score_mode: Combines the result of the used functions. Score mode has the following parameters:
  - multiply: Function results are multiplied together (default).
  - sum: Function results are added up.
  - avg: The average of all the function results.
  - max: The highest function result is used.
  - min: The lowest function result is used.
  - first: Uses only the result from the first function that either doesn’t have a filter or that has a filter matching the document.

Function Score Query, using Field Value Factor

- boost_mode: Combines the _score result of the query with the _score of the functions. Boost mode accepts the following parameters:
  - multiply: Multiply the _score with the function result.
  - sum: Add the function result to the _score.
  - min: The lower of the _score and the function result
  - max: The higher of the _score and the function result.
- - replace: Replace the _score with the function result

Function Score

- Start Searching for Apartments
- In the first query, we are searching for apartments that match our first criteria:
  - We need at least three rooms, but we would favor apartments with four rooms.
  - Apartments with four rooms get a boost of 1.

Refer github.com/eswaribala/rpsvelkjan2023/functionscore.txt

Function Score

```
|  rank  |  rooms  |  city          |  score  |
|  ----  |  -----  |  ------------  |  ------ |
|  1     |  4.5    |  Biel          |  2.0    |
|  2     |  4.5    |  Brügg         |  2.0    |
|  3     |  4.5    |  Lyss          |  2.0    |
|  4     |  4.5    |  Bern          |  2.0    |
|  5     |  4.5    |  Grenchen      |  2.0    |
|  6     |  5.5    |  Schüpfen      |  2.0    |
|  7     |  4.5    |  Zollikofen    |  2.0    |
|  ====  |  =====  |  ============  |  =====  |
|  8     |  3.5    |  Moosseedorf   |  1.0    |
|  9     |  3.5    |  Ostermundigen |  1.0    |
```

Decay Functions

- This decay separates our results into three neat compartments with a very natural transition between all three parts.
- Apartments, that farther it gets from the origin price, receive a minor penalty or decay.
- Moderately more expensive apartments get a more substantial score penalty, increasing more and more within the scale.
- Apartments outside our preferred range get a severe punishment.
- Decay functions score a document with a function that decays depending on the distance of a numeric field value of the document from a user given origin.

Decay Functions

rank	price in CHF	rooms	city	score
1	1790	4.5	Brügg	2.8185682
2	1795	4.5	Zollikofen	2.7985601
3	1980	4.5	Lyss	1.6401769
4	1980	5.5	Grenchen	1.6401769
5	1990	4.5	Biel	1.5697317
6	2070	3.5	Ostermundigen	0.34841746
7	2100	3.5	Moosseedorf	0.29163226
8	2350	4.5	Schüpfen	0.115865104
9	2390	4.5	Bern	0.07667832

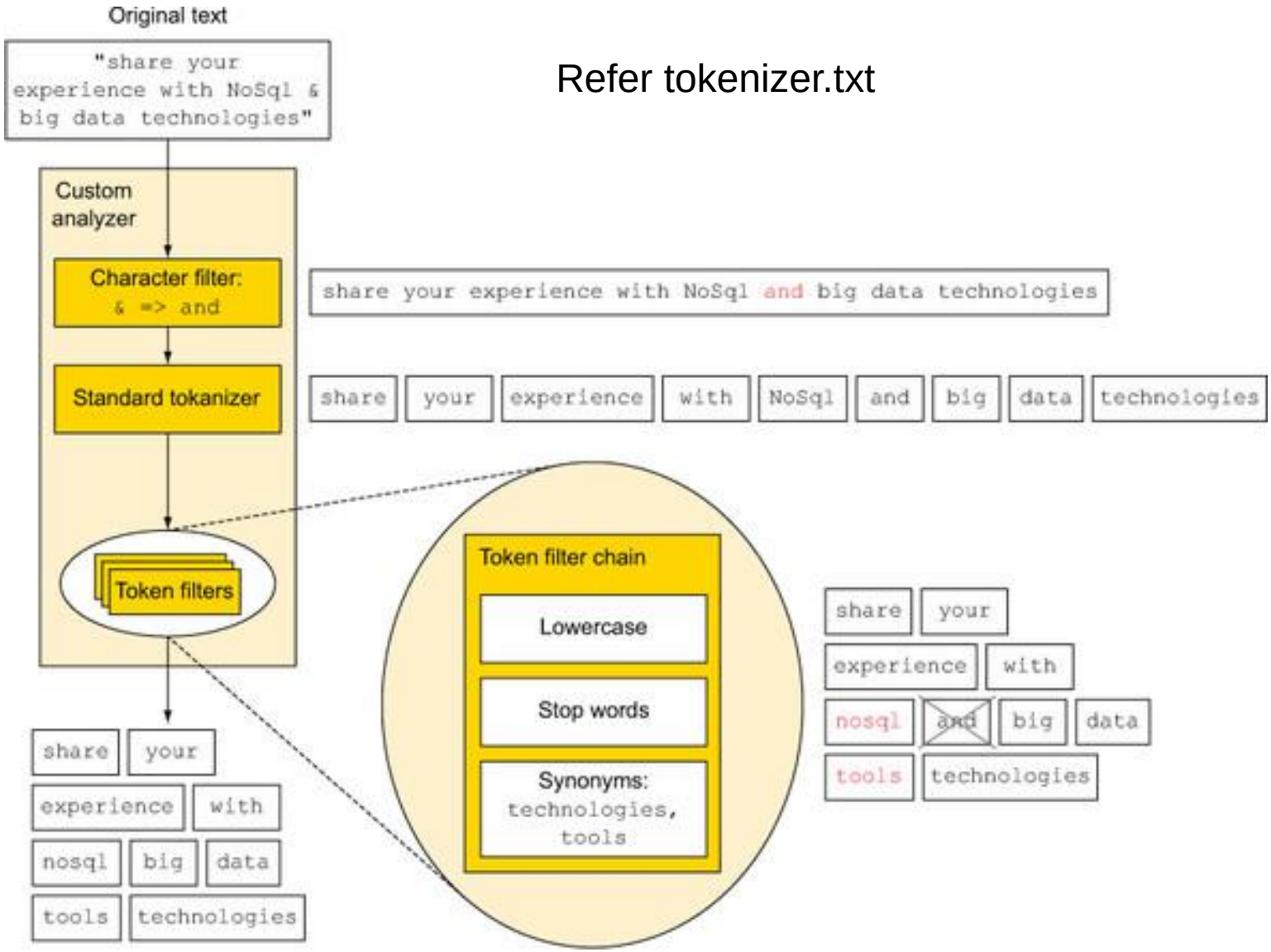
Combine Function Scores

- Let us assume I have switched my working location.
- Now I want to find a new apartment based on my new working site in a 20 km radius.
- Now we add another influence to our ranking: the location.
- My new origin is Lyss, a very family-friendly city, and my scale is 5 kilometres.
- We do a function score on a geo_point field and our scale is defined as 5 km.
- The new gauss function on the location is similar to the geo_distance query.
- We want to multiply the scores to get a ranking based on price and location.

Combine Function Scores

rank	price in CHF	rooms	city	distance	score
1	1980	4.5	Lyss	0 km	1.0934513
2	1790	4.5	Brügg	7 km	0.7212604
3	1990	4.5	Biel	10 km	0.117894106
4	2350	4.5	Schüpfen	8 km	0.022560354
5	1795	4.5	Zollikofen	16 km	0.009281355
6	2540	5.5	Grenchen	17 km	0.0023489068
7	2100	3.5	Moosseedorf	16 km	6.728557E-4
8	2070	3.5	Ostermundigen	21.5 km	2.0915837E-5
9	2390	4.5	Bern	20.5 km	9.355606E-6

Analyzers and search queries

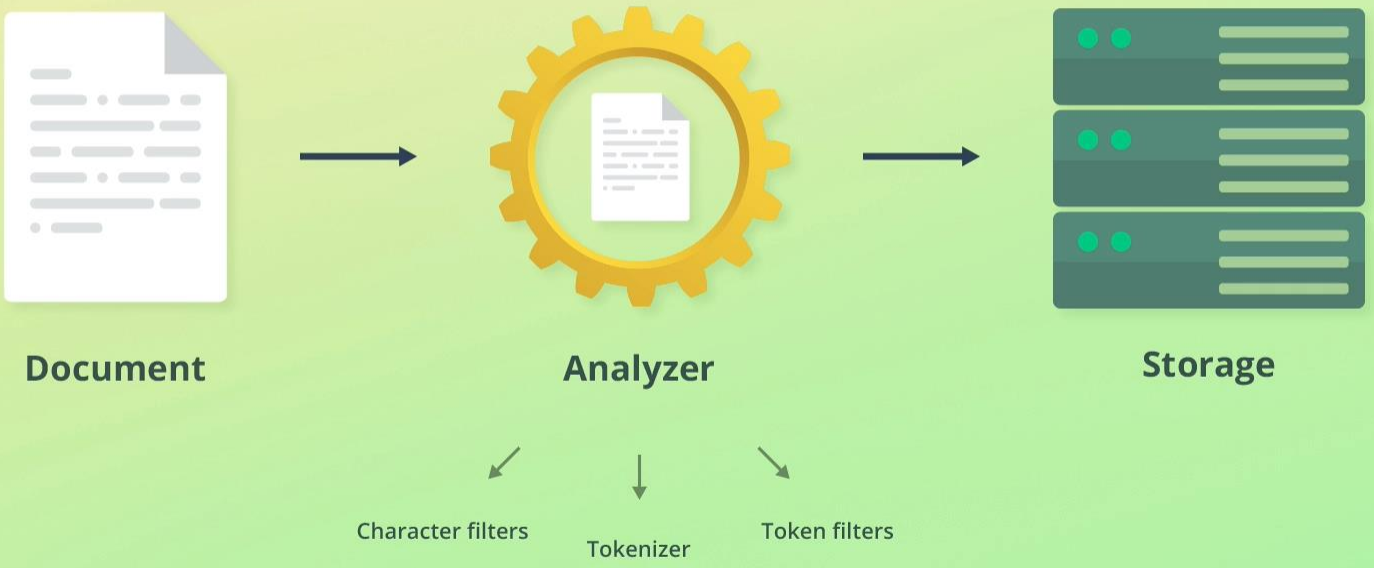


Refer tokenizer.txt

Analyzer

- Sometimes referred to as *text analysis*
- Applicable to text fields/values
- Text values are analyzed when indexing documents
- The result is stored in data structures that are efficient for searching etc.
- The `_source` object is **not** used when searching for documents
  - It contains the exact values specified when indexing a document

Analyzer


Document
Analyzer
Storage
Character filters
Tokenizer
Token filters

Character Filters

- Adds, removes, or changes characters
- Analyzers contain zero or more character filters
- Character filters are applied in the order in which they are specified
- Example (`html_strip` filter)
  - **Input:** `"I&apos;m in a <em>good</em> mood - and I <strong>love</strong> açaí!"`
  - **Output:** `"I'm in a good mood - and I love açaí!"`

Tokenizers

- An analyzer contains **one** tokenizer
- Tokenizes a string, i.e. splits it into tokens
- Characters may be stripped as part of the tokenization
- Example
  - **Input:** `"I REALLY like beer!"`
  - **Output:** `["I", "REALLY", "like", "beer"]`

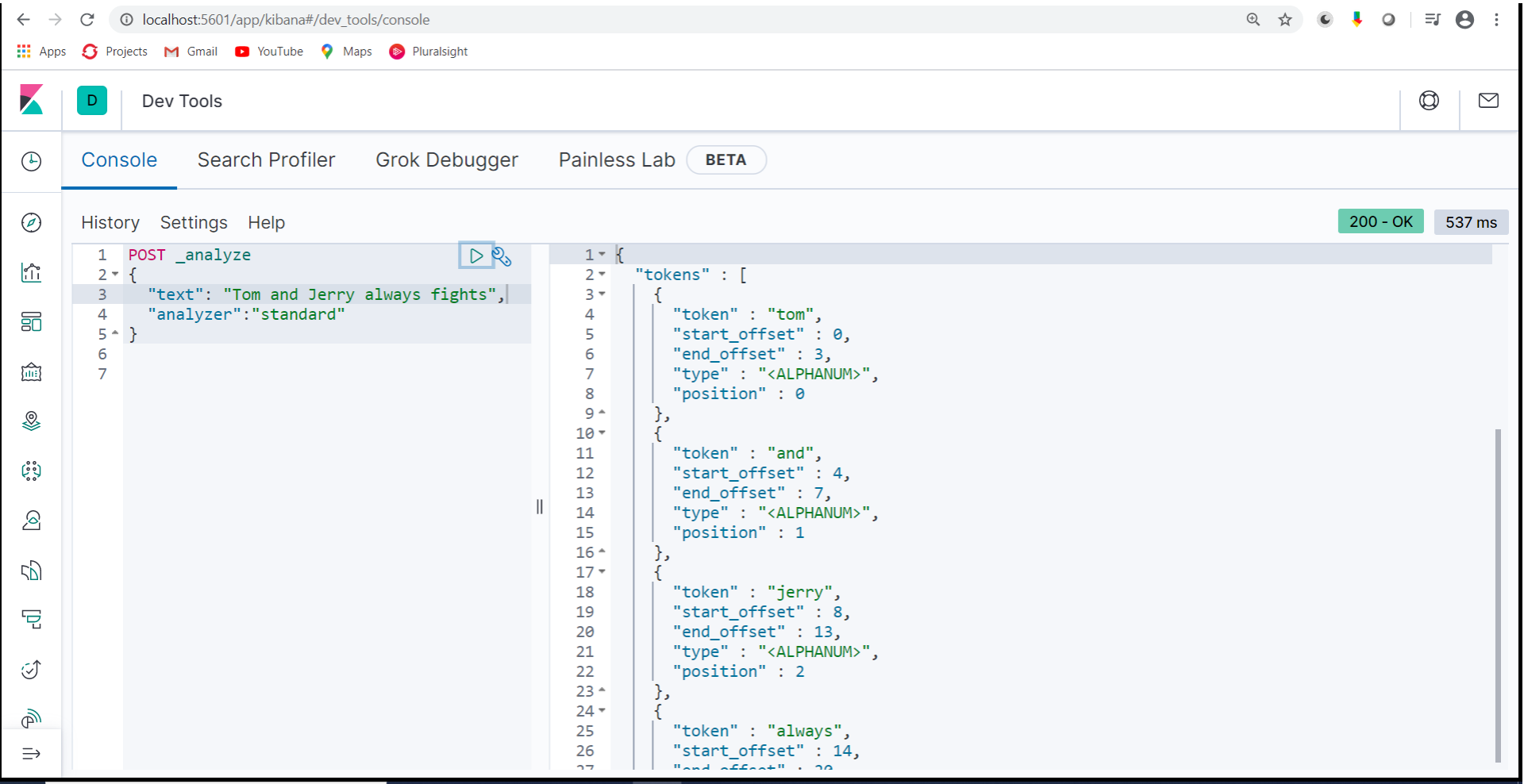
Token Filters

- Receive the output of the tokenizer as input (i.e. the tokens)
- A token filter can add, remove, or modify tokens
- An analyzer contains zero or more token filters
- Token filters are applied in the order in which they are specified
- Example (`lowercase` filter)
  - **Input:** `["I", "REALLY", "like", "beer"]`
  - **Output:** `["i", "really", "like", "beer"]`

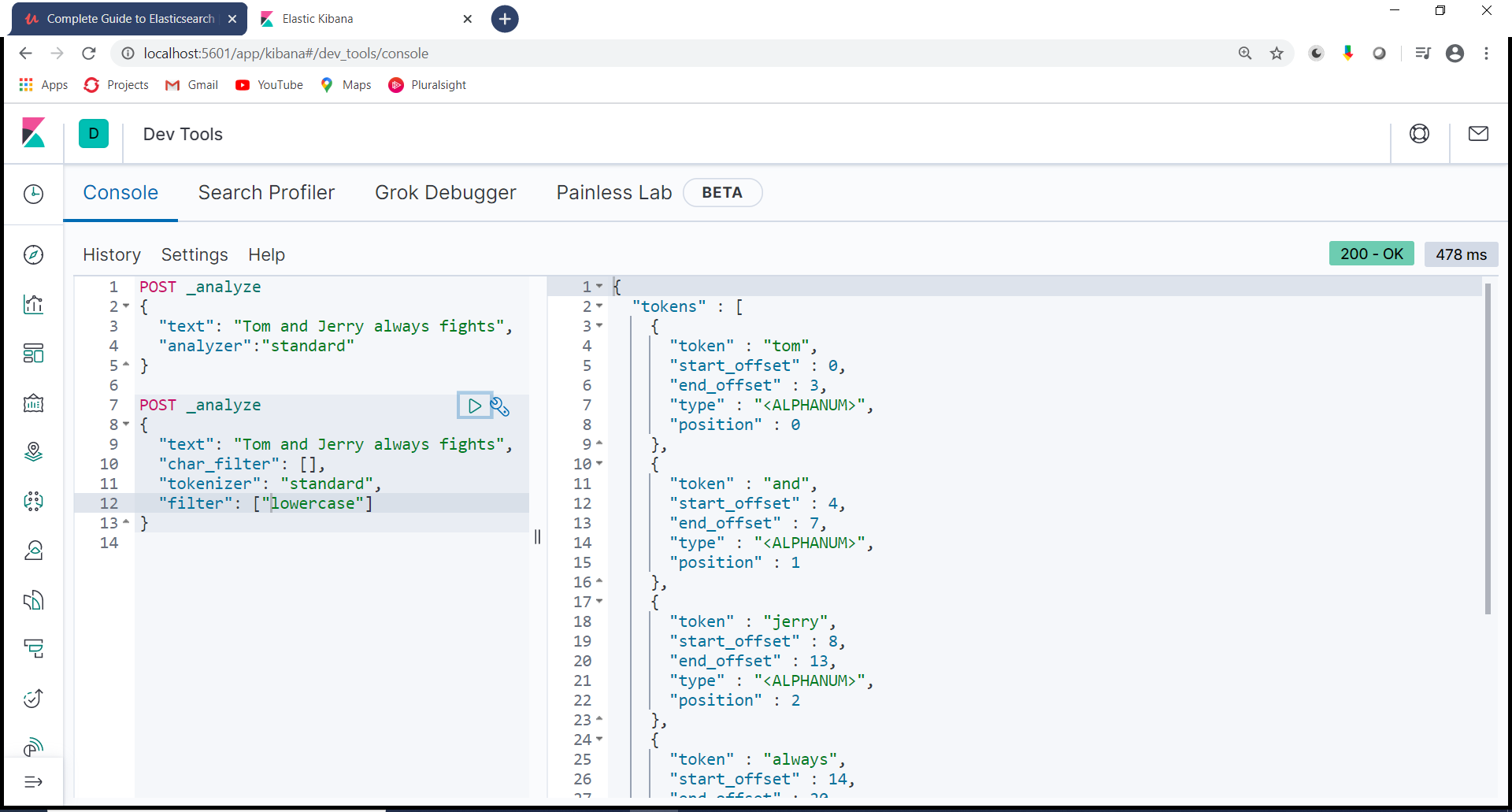
Filters


Character filters
(none)
Input: "I REALLY like beer!"
Output: "I REALLY like beer!"
Tokenizer
standard
Input: "I REALLY like beer!"
Output: ["I", "REALLY", "like", "beer"]
Token filters
["lowercase"]
Input: "I", "REALLY", "like", "beer"
Output: "i", "really", "like", "beer"
STANDARD ANALYZER

Filters


localhost:5601/app/kibana#/dev_tools/console
Apps
Projects
Gmail
YouTube
Maps
Pluralsight
Dev Tools
Console
Search Profiler
Grok Debugger
Painless Lab
BETA
History
Settings
Help
200 - OK
537 ms
POST _analyze
{
"text": "Tom and Jerry always fights",
"analyzer":"standard"
}
{
"tokens" : [
{
"token" : "tom",
"start_offset" : 0,
"end_offset" : 3,
"type" : "<ALPHANUM>",
"position" : 0
},
{
"token" : "and",
"start_offset" : 4,
"end_offset" : 7,
"type" : "<ALPHANUM>",
"position" : 1
},
{
"token" : "jerry",
"start_offset" : 8,
"end_offset" : 13,
"type" : "<ALPHANUM>",
"position" : 2
},
{
"token" : "always",
"start_offset" : 14,
Type here to search
ENG
IN
02:39
02/06/2020

Filters


Complete Guide to Elasticsearch
Elastic Kibana
localhost:5601/app/kibana#/dev_tools/console
Apps
Projects
Gmail
YouTube
Maps
Pluralsight
Dev Tools
Console
Search Profiler
Grok Debugger
Painless Lab
BETA
History
Settings
Help
200 - OK
478 ms
POST _analyze
{
"text": "Tom and Jerry always fights",
"analyzer":"standard"
}
POST _analyze
{
"text": "Tom and Jerry always fights",
"char_filter": [],
"tokenizer": "standard",
"filter": ["lowercase"]
}
{
"tokens" : [
{
"token" : "tom",
"start_offset" : 0,
"end_offset" : 3,
"type" : "<ALPHANUM>",
"position" : 0
},
{
"token" : "and",
"start_offset" : 4,
"end_offset" : 7,
"type" : "<ALPHANUM>",
"position" : 1
},
{
"token" : "jerry",
"start_offset" : 8,
"end_offset" : 13,
"type" : "<ALPHANUM>",
"position" : 2
},
{
"token" : "always",
"start_offset" : 14,
Type here to search
ENG
IN
02:42
02/06/2020

Tokens

Sentence → Tokens

`"2 guys walk into  a bar, but the third... DUCKS! :-)"` → `["2", "guys", "walk", "into", "a", "bar", "but", "the", "third", "ducks"]`

TERM	DOCUMENT #1
2	X
a	X
bar	X
but	X
ducks	X
guys	X
into	X
the	X
third	X
walk	X

Sentence and Tokens

Sentence → Tokens

`"2 guys walk into  a bar, but the third... DUCKS! :-)"` →	`["2", "guys", "walk", "into", "a", "bar", "but", "the", "third", "ducks"]`
`"2 guys went into a bar"` →	`["2", "guys", "went", "into", "a", "bar"]`
`"2 ducks walk around the lake"` →	`["2", "ducks", "walk", "around", "the", "lake"]`

TERM	DOCUMENT #1	DOCUMENT #2	DOCUMENT #3
2	X	X	X
a	X	X	
around			X
bar	X	X	
but	X		
ducks	X		X
guys	X	X	
into	X	X	
lake			X
the	X		X
third	X		
walk	X		X
went		X	

Inverted Indices

- Mapping between terms and which documents contain them
- Outside the context of analyzers, we use the terminology “terms”
- Terms are sorted alphabetically
- Inverted indices contain more than just terms and document IDs
    - E.g. information for relevance scoring

Inverted Indices

```
{
  "name": "Coffee Maker",
  "description": "Makes coffee super fast!",
  "price": 64,
  "in_stock": 10,
  "created_at": "2009-11-08T14:21:51Z"
}
```

```
{
  "name": "Toaster",
  "description": "Makes delicious toasts...",
  "price": 49,
  "in_stock": 4,
  "created_at": "2007-01-29T09:44:15Z"
}
```

name field

TERM	DOCUMENT #1	DOCUMENT #2
coffee	X	
maker	X	
toaster		X

description field

TERM	DOCUMENT #1	DOCUMENT #2
coffee		X
delicious	X	
fast		X
makes	X	X
super		X
toasts	X	

Inverted Indices

```
POST /stemming_test/_doc
{
  "description": "I loved drinking
   bottles of wine on last year's
   vacation."
}
```

TERM	DOCUMENT #1	...
i	X	
loved	X	
drinking	X	
bottles	X	
of	X	
wine	X	
on	X	
last	X	
year's	X	
vacation	X	

```
GET /stemming_test/_search
{
  "query": {
    "match": {
      "description": "loves"
    }
  }
}
```

no matches

Introduction to Stop Words

- Words that are filtered out during text analysis
  - Common words such as `"a", "the", "at", "of", "on"`, etc.
- They provide little to no value for relevance scoring
- Fairly common to remove such words
  - Less common in Elasticsearch today than in the past
    - The relevance algorithm has been improved significantly
- Not removed by default, and I generally don't recommend doing so

Inverted Indices

- Mapping between terms and which documents contain them
- Outside the context of analyzers, we use the terminology “terms”
- Terms are sorted alphabetically
- Inverted indices contain more than just terms and document IDs
    - E.g. information for relevance scoring
- One inverted index per text field
- Other data types use BKD trees, for instance

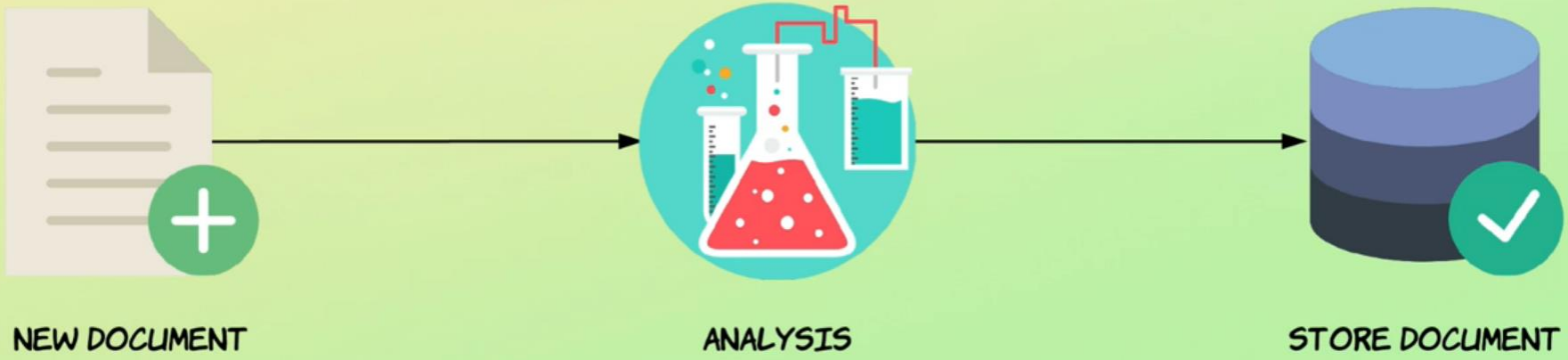
Inverted Indices

- Values for a text field are analyzed and the results are stored within an inverted index
- Each field has a dedicated inverted index
- An inverted index is a mapping between terms and which documents contain them
- Terms are sorted alphabetically for performance reasons
- Created and maintained by Apache Lucene, *not* Elasticsearch

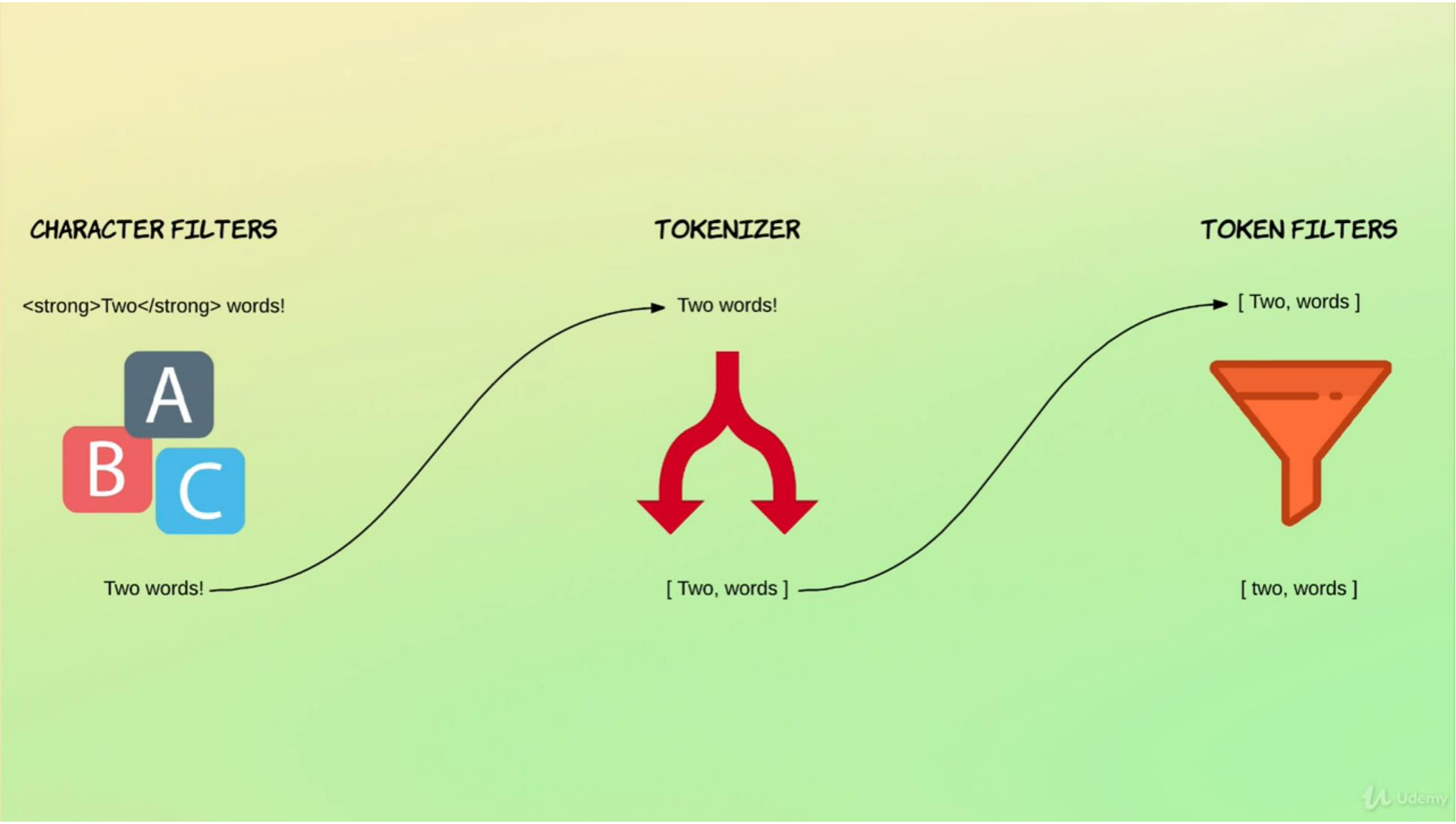
Inverted Indices

- Inverted indices enable **fast** searches
- Inverted indices contain other data as well
  - E.g. things used for relevance scoring (covered later)
- Elasticsearch (technically, Apache Lucene) uses other data structures as well
  - E.g. BKD trees for numeric values, dates, and geospatial data

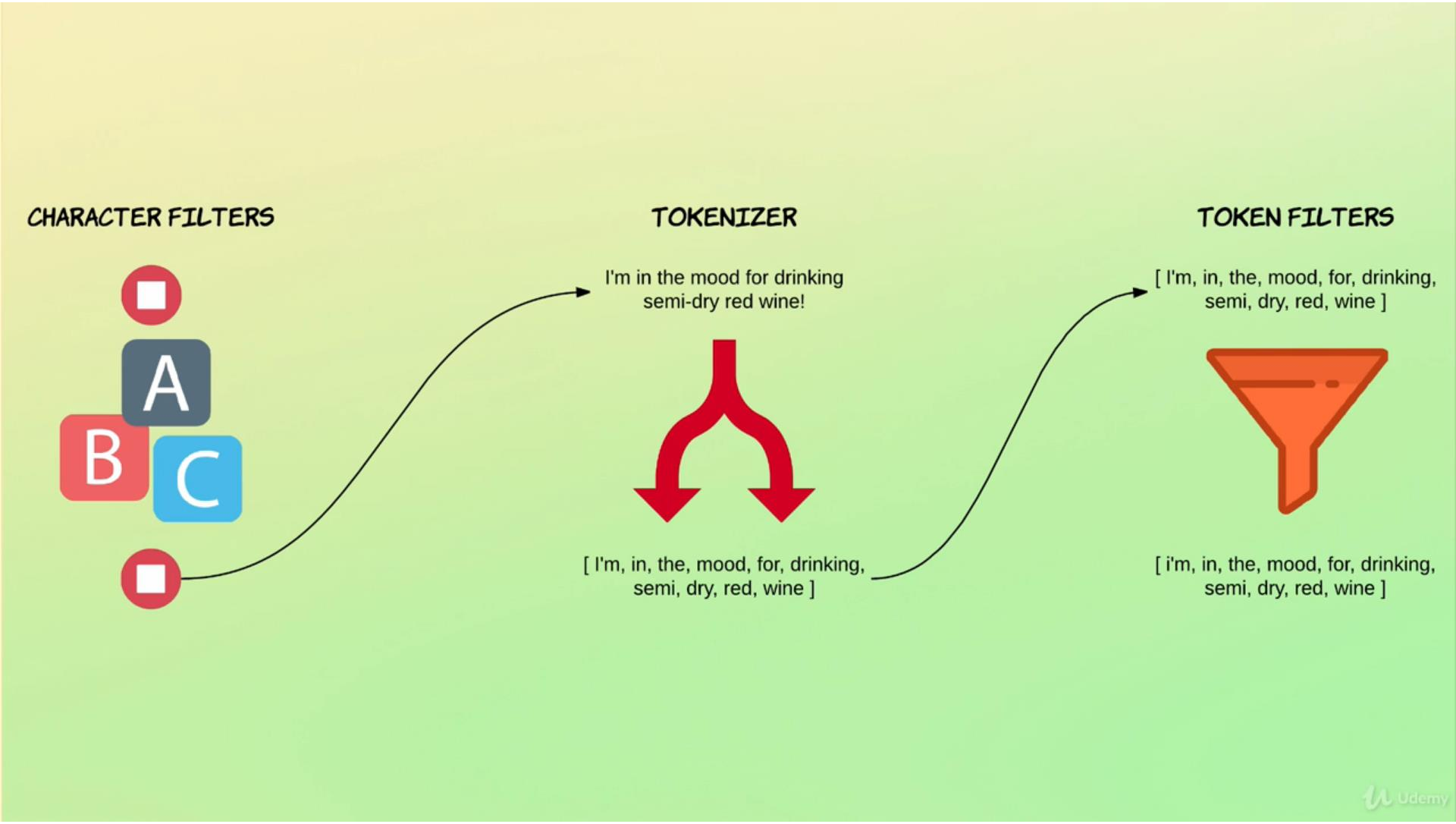
Analysis Process



Analysis Process



Analysis Process



Standard Analyzer


```
GET /stemming_test/_search
{
  "query": {
    "match": {
      "description": "drinking"
    }
  }
}
```

Stemming Analyzer