Elk Stack Cheat Sheet : With Examples + screenshots of results.

Why Kibana ?

Easiest way of running queries is using Kibana tool. It also sends requests to Elastic search API.

It formats the response for us and makes it easier to read.

It also sets the correct-content type header.

Auto typing.

Postman / Curl can also be used.

1. GET /\_cluster/health

\_cluster is the api

health is the command

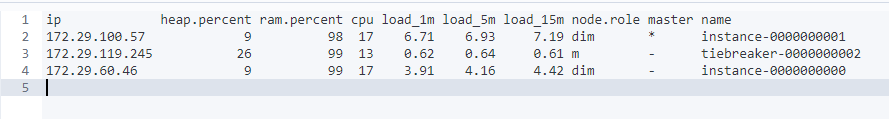
2. \_cat api which outputs data in a human readable format.

\_cat/nodes : all nodes in the cluster

command :

GET /\_cat/nodes?v

v : query parameter : instructs elastic search to include a descriptive header in the output- to identify each piece of info

o/p:  


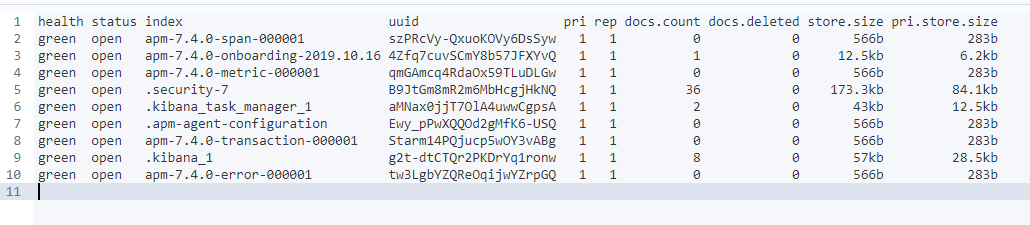
3. GET /\_nodes/stats

API : nodes

To inspect nodes in a lot of detail.  
  


4. GET /\_cat/indices?v

curl -XGET "http://24e374bd44bb4bf884cfd220f7f55cf0.containerhost:9244/\_cat/indices?v"

List all indices  


If its elastic cloud : we can send elastic search requests with the CURL http client

curl –XGET –u elastic: wp2ss2xAxTY5UnRUdgo2tUgW ‘<https://24e374bd44bb4bf884cfd220f7f55cf0.ap-southeast-1.aws.found.io:9243/.kibana/_serch> –H “Content-Type: application/json” –d { “query” : { “match\_all”:{} }}’

5. Sharding and scalability.

Elastic search has a cluster with nodes.

Each node has a capacity and you can store 200gb of data in 2 nodes each having 100gb of data.

Sharding : Makes it possible to scale the amount of documents we can store.

It’s a way to divide indices into small pieces called shard[any number of shards] – done at index level.  
 Indices could contain a few hundred records to a million records.

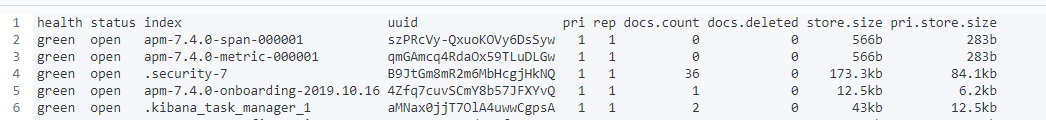
To horizontally scale the data volume, indices are divided into shards.

[ If an index has 5 shards : There is no need that all shards be in different nodes – they could be in the same node as well. ]

**Advantage of having shards**: Search query on an index can be run independently on different nodes – increasing the speed of output [as shards could be stored in different nodes.].

Sharding : it is a process to sub divide an index in smaller pieces.

: it increases the number of documents an index can store.

6. Get /\_cat/indices?v  
  


Pri – primary shard : the number of shards a given index has.

spilt api : to increase the number of shards in an index.

shrik api : to reduce the number of shards for an index.

7. Replication.

- it is configured at the index level

- Copies of shards are created – replica shards.

- When an index is created we can choose how many replicas of each shard we want. 1 being the default.

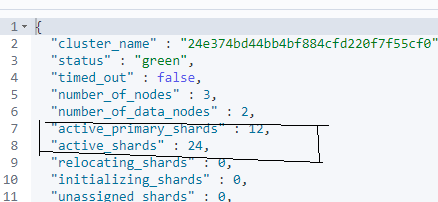
- Replica shard is never stored in the same node.

- Throughput can be increased as the query is distributed.

8. snapshot: to store an index completely.

Used for daily backup, manual backups are taken just before applying changes – just to be sure the changes are good.

9. Create and Index: PUT /indexname

10. Check the cluster Health: GET /\_cluster/health  


The cluster health is **green** as there is replication of shards.

11. check the shards : GET /\_cat/shards?v  


p- primary shard.

r – replica shard.

state = STARTED: both primary and replica shards are available for requests.

12. different roles of a node.

Master node :

Data node :

Ingest Node : This enables a node to run Ingest pipelines.

Node.ml : identifies a node as machine learning

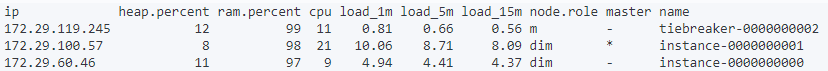
Xpack.ml.enabled : Determines if the node should respond to machine learning requests.

Co-ordination node : how ES process a request and delegates it internally to data nodes[by removing other roles – no role exists., kind of a load balances].

13. Roles of our nodes ?

Command : GET /\_cat/nodes?v

o/p :



dim – data ingest and master

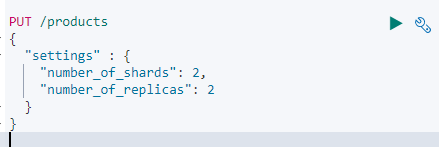
m - master

14. ES exposes a rest API.

- HTTP verb is important to perform a particular action.

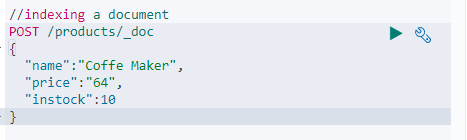
DELETE AN INDEX :   
  
CREATE AN INDEX by specifying the number of shards and replicas. [ For indices that we use to create for production purposes we should stick to the default values ]  
first line : http verb + end point

Other lines : *json request body* AND *index settings* should be passed as a json object



15. INDEXING A DOCUMENT BY sending a request to an end point.

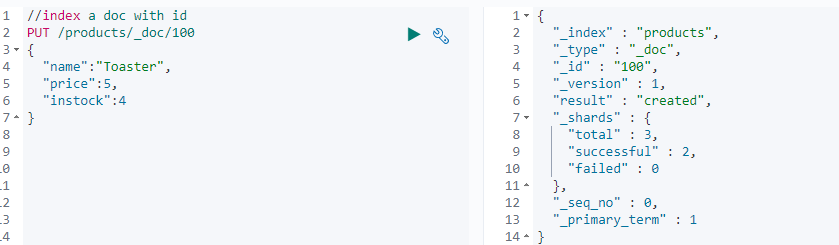
*- we need to define the document within the request body as a json object*

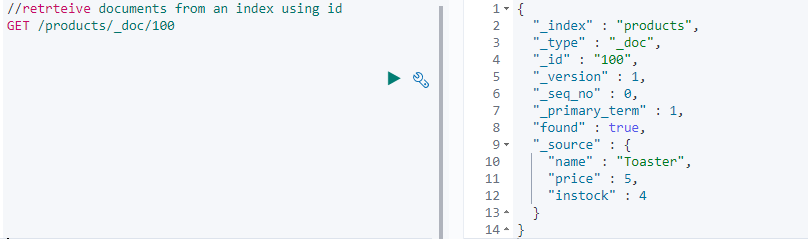


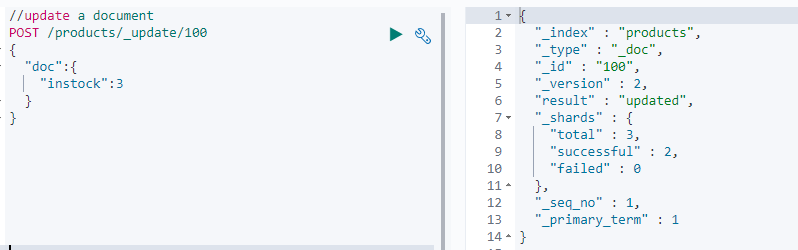
o/p :   
  
  
\_shards : The number of shards in which you successfully stored a document.  
In step 14 : replics are 2, so document was added to primary and also to the 2 replicas so total is 3.

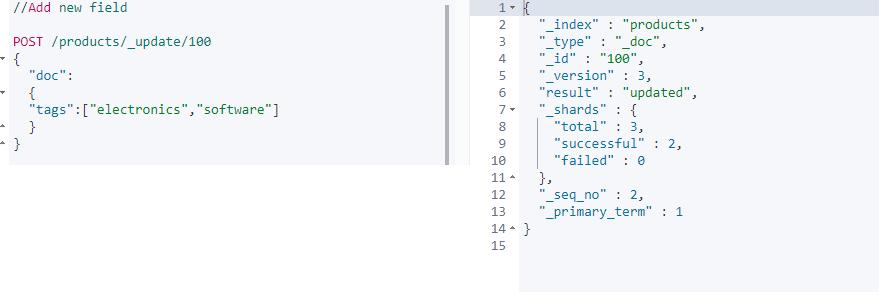
\_id : was created automatically but we can add that too.

16. Add \_id while indexing a document.

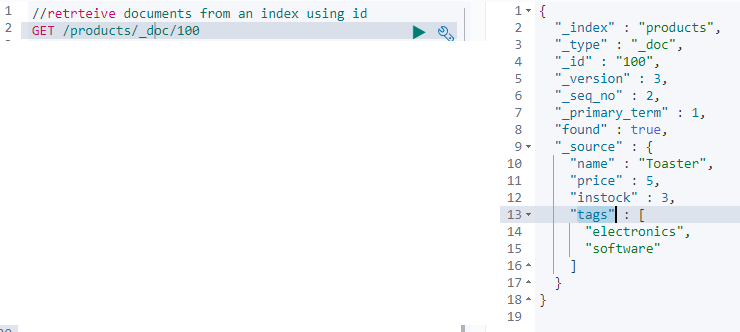
To specify an id : we need to change the http verb to **PUT** as this is a convention for REST api.  
  


17. Retrieving documents from an index.  


18. Updating documents: by sending a update request   


19. Adding new fields to existing documents.  


Now we see the new field with data.

****

How this works internally.

* ES documents are immutable (!).
* The \_update api : retrieved the document/changed its fields/re-indexed the document with the same id.

20. Scripted updates

- There is a use case where you retrieve a document first + update a field value + update the document.

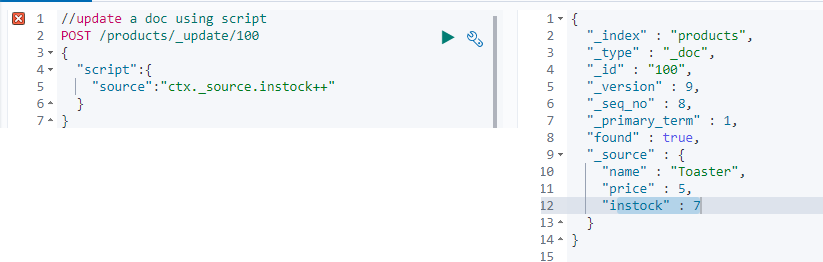
- all of this can be done in one go.

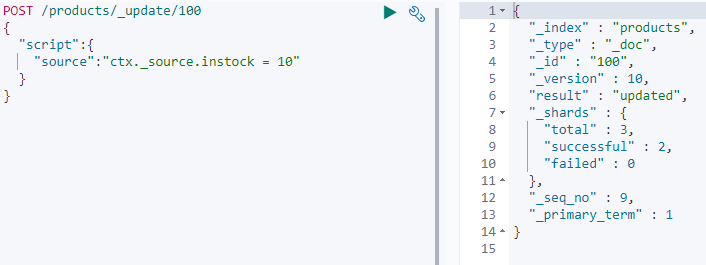
- with scripting : we can use if statements

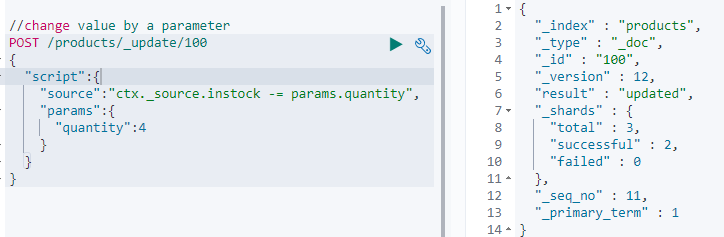
- use the \_update api + script update with the request body

Example :

ctx: it is a ES variable and ‘short for context’.

\_source: access the source document using the \_source property, which gives us an object containing the objects fields.  
  


* We can do assignments also apart from updating a field.  
    
  
* Now if some customer purchases 4 products, the application must send a request to reduce the instock count by 4.
* We use this by writing a params object and using its value to reduce the count

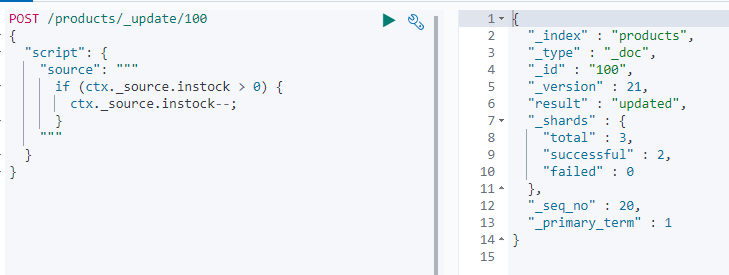
  


* If you try to update a field value with its existing value : result will be “no-op” else updated
* If the field value is set as part of a script the ”result” will always be as “updated”
* There are 2 exceptions to this – both being if we explicitly set the operation within the scripts.

Case 1 : A script to ignore a document based on a condition – by setting the **op** property on the **ctx** variable to “noop”.



What the above script does is reduce the instock field value in the doc by 1 and set the resultkeyto ‘updated’; if it is zero the **result** key will be set to a value of ‘noop’

Case 2 :  


Ir-respective of the execution of the if block : the result will always contain “updated”

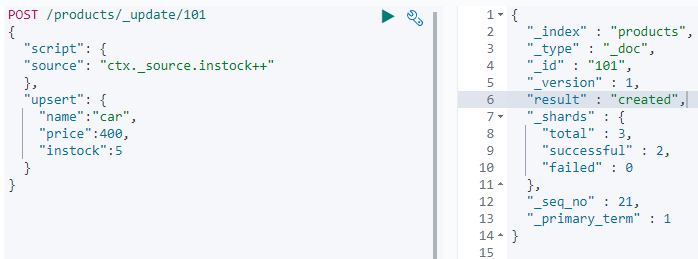
If you want to detect if nothing was changed then : follow case 1.

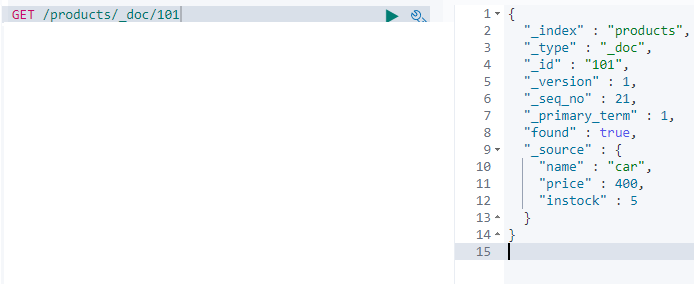
Case 3 : We can set the operation to delete which will cause the document to be deleted.  
This will set the result key to deleted with in the results.

**ctx.op = ’delete’;**

21. Updating documents using upserts

Insert if not present or else update it.

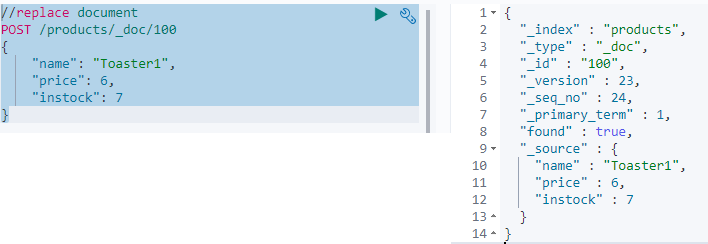


The instock is 5 after creation.  


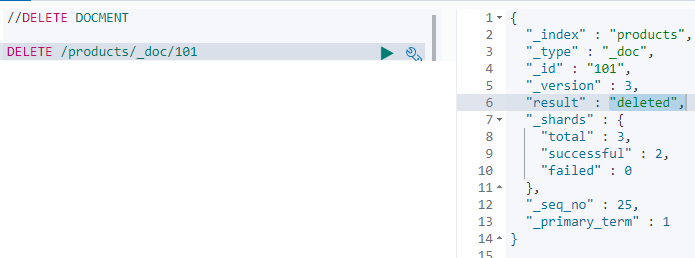
If the POST query is run again : **"result" : "updated".**

So, The script is run if the document exists else the upsert is run.

22. Replacing documents.



As the whole document was replaced if any other field existed in the old document they will not still exist.

23. DELETE documents  


24. How ES reads data ?

- its about reading a single document.

- a given node receives the read request.

- this node is responsible for co-ordinating the request – so its called the co-ordinating node.

- locate where the document is stored -> that’s done with routing.

- with Routing : Routing resolves to a shard that stores a given document.

- i.e : It resolves to a primary shard or a replication group [contains the list of primary shards + its replica groups]

- Note : If elastic search directly retrieved the document from primary shard – all retrievals will end up in the same shard – which does not scale well : INSTEAD a shard is chosen from the replication group.

- So a technique called ARS : Adaptive replica selection is used.

- ES tries to select a shard copy it believes that can yield the best performance.

- then the co-ordinating node send the read request to that shard… and so on.

25. How ES writes data

- the request is resolved to a replication group that stores/SHOULD STORE the document.

- the write is always routed to the primary shard

- It validates the field values and structure of the request.

- The write op is performed in the primary shard before it fwds to the replica shards (this could be in parallell).

- Note that the operation succeeds even if the operation cannot be replicated to the replica shards.

> Since elastic search is distributed and many tasks happen asynchronously many things can go wrong.  
> When a document is indexed – the primary shard validates the operation and indexes the document locally  
> Assume there are 2 replicas in the replication group, then the primary shards send the operation to these two replicas.  
> Let us assume the operation reaches only 1 replica shard (RS1) – as the primary shard (PS) goes down due to a hardware failure.[PS RS1 RS2]  
> When this happens ES goes through a recovery process – In which one of the replica shard will be promoted as the new Primary shard [each RG must have a primary shard.]  
> Only one replica shard indexed the new document and other did not.  
> RS 2 thinks it is up to date – but this is not the case.  
> The new document will be found only half of the time – depending on which shard serves the request.

So Elastic Search solves this using : *primary terms + sequence numbers.*

**Primary terms (PT):**A way to distinguish between old and new primary shards, when the primary shard of a replication group has changed.  
The primary term for a replication group is essentially just a counter about how many times the primary shard has changed. [1 or 2 or 3]  
In the above case : The Primary Term for the RG will be increased by 1 as the primary shard failed and one of the replication shard was promoted to be the primary shard.  
The Primary terms for all Replication groups are persisted in the cluster’s state.  
When write operations are performed : the current PT is appended to the operations that are sent to the replica shards.  
This enables the RS to tell if the PS was changed since the operation was forwarded.

**Sequence Number (SN):**  
Apart from associating each operation with a primary term a **sequence number** is given to each operation.

This is also just a counter that is incremented for each operation – until the PT changes.

The primary shard is responsible for increasing this number when it process a write request.

These sequence numbers help ES to know in which order write operations happened, on a given primary shard.

Thus instead of comparing data on the disk – we can use PT and SN to figure out which operations have already been performed + which are needed to bring a given shard up to date.

[global checkpoints and local checkpoints are also imp]  
  
**Global checkpoints** : exists for each replication group.  
**Local Check Points** : kept for each replication shard.

It is the active sequence number that all of the active shards with in a replication group have been aligned atleast up to.

This means that any operations containing a sequence number lower than the global check point have already been performed on all shards with in the RG.

If a primary shard fails and re-joins the cluster at a later point : ES only need to compare the operations that are above the global check point that it last knew about.

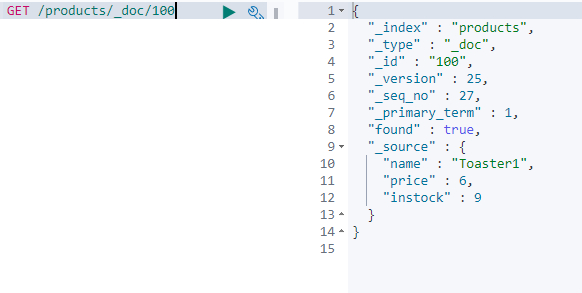
Like wise : if a replica shard fails only the operations that have a sequence number higher than its local checkpoint need to be applied when it comes back.

THIS MEANS TO RECOVER ES JUST NEEDS TO COMPARE THE OPERATIONS WHILE THE SHARD WAS GONE , INSTEAD OF THE ENTIRE HISTORY OF THE REPLICATION GROUP.

26. **Optimistic concurrency control.**

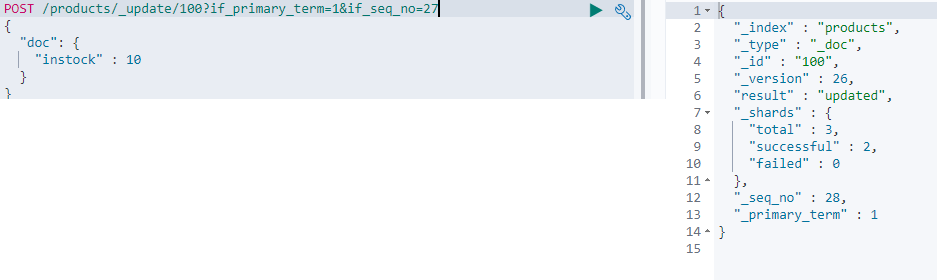
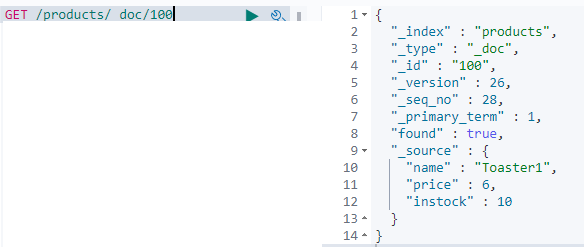
We need our update to fail if the document has been modified since we have retrieved it.  
PT and SN are of use here.

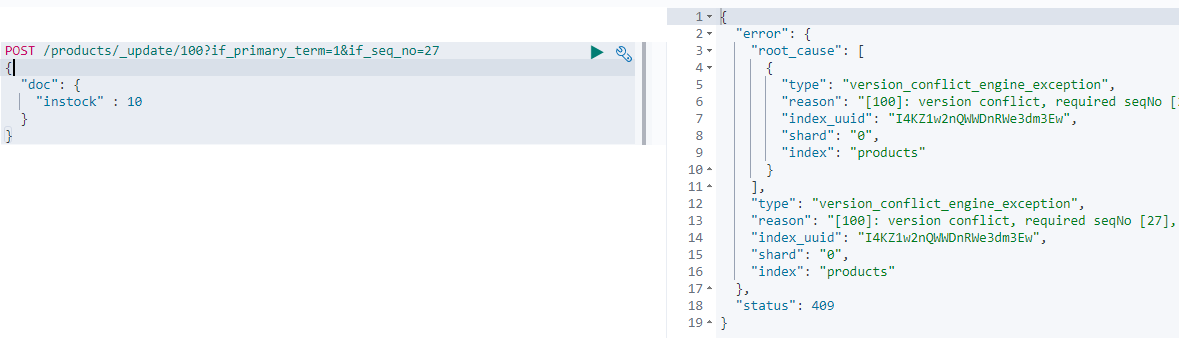
When we retrieve the product - > PT and SN are included in the results.



We take them and add them to the post request that we send to update the document.

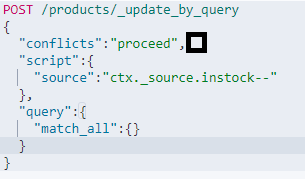
Elastic search will use these two values and ensure a document will not be over written in-advertently, if it has changed since we retrieved it.

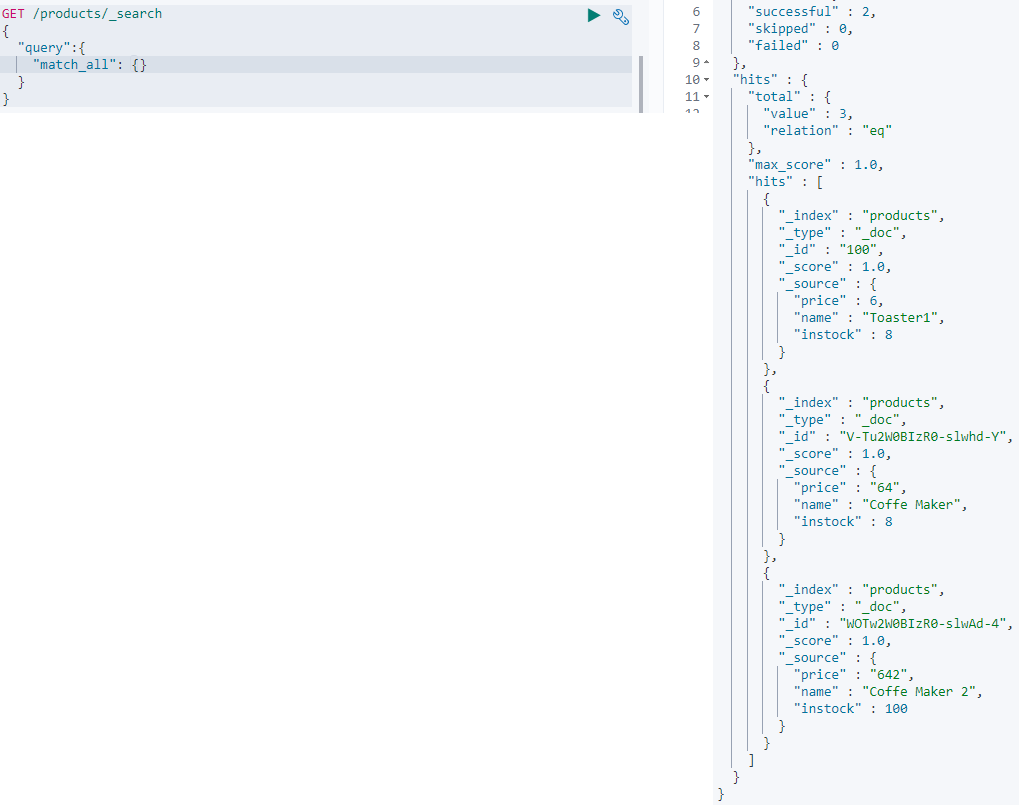
  
  


Let’s try to update with the same old seq num again  


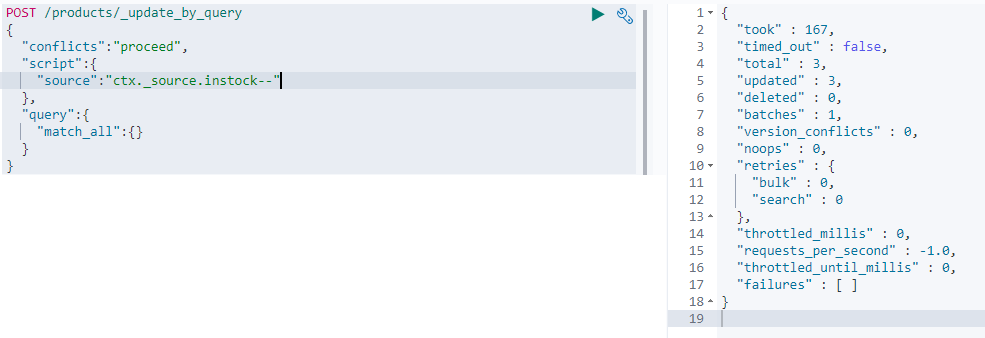
27. Update multiple documents with a query – how it works internally.

The first things that happens when an update by query request is processed :

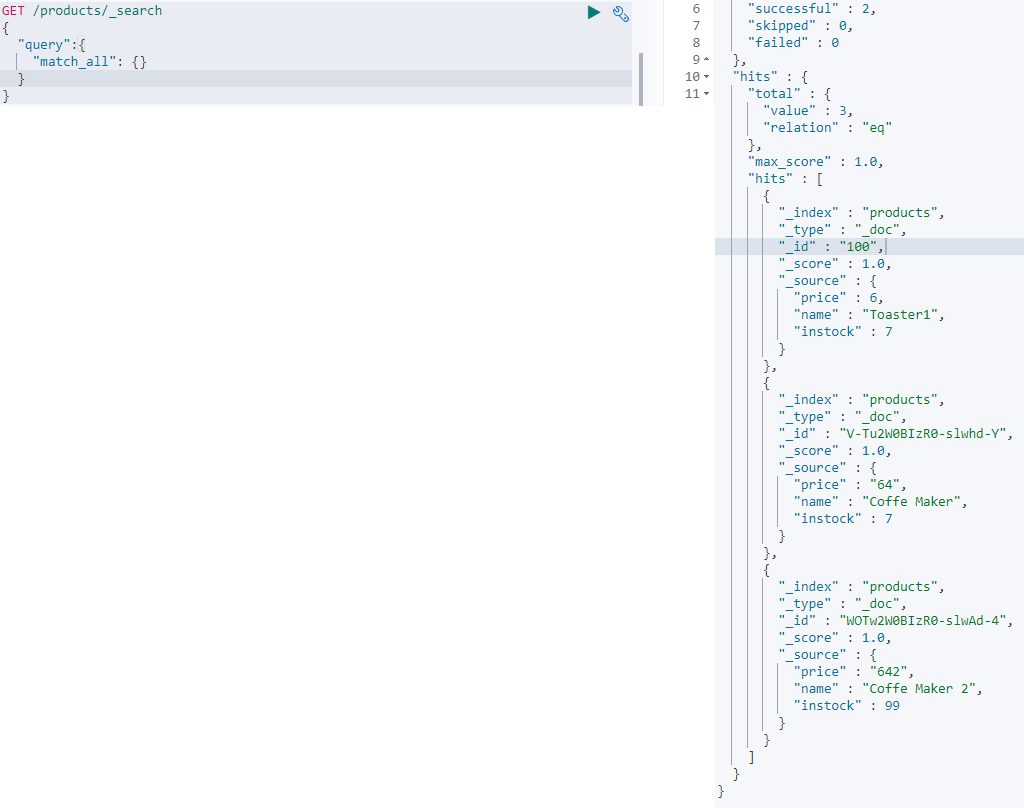
* A snap shot of the index is taken.
* A search query is sent to each of the indexes shards – in order to find all of the documents that match the supplied query.
* Whenever a search query matches any documents – a bulk request is sent to update those documents
* “batches” : How many batches were used to retrieve the documents.
* A search and bulk request are sent sequentially.
* When a number of documents are updated when an error occurs : Those documents remain updated even though the request failed.
* The queries could run successfully against a Replication Group A, But something went wrong when sending queries to Replication Group B – causing the query to be aborted.
* Any documents that match the search query are therefore not updated within the replication group C.
* The documents that were updated with in the Replication Group A will remain updated even though the query was aborted.
* **The reason why EC takes a snapshot of the index** : it is to ensure that the updates are performed on the basis of the current state of the index.
* Let us say for an index where documents are indexed, modified and deleted frequently – it is not unlikely that something has changed, from when EC received a query to when it finishes processing it.
* This is especially true when updating many documents.
* When ES is requested to update a given document, it uses the documents PT and SN from the snapshot to ensure that it has not been changed since creating the snapshot.
* If the document has been changed – there will be a version conflict causing the document to not be updated.
* This will cause the entire query to be aborted.
* The number of conflicts will be returned with the ‘version\_conflict’ key with in the results.
* If you don’t want the query to be aborted – what the below one does is it will cause the version conflicts to be counted instead of the query being aborted.  
  

28. Update multiple documents with a query. API : \_update\_by\_query  
First find all the documents in the index products  


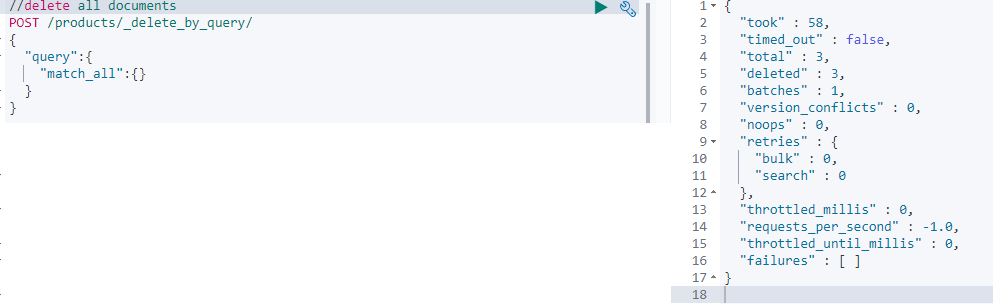
Now lets update them



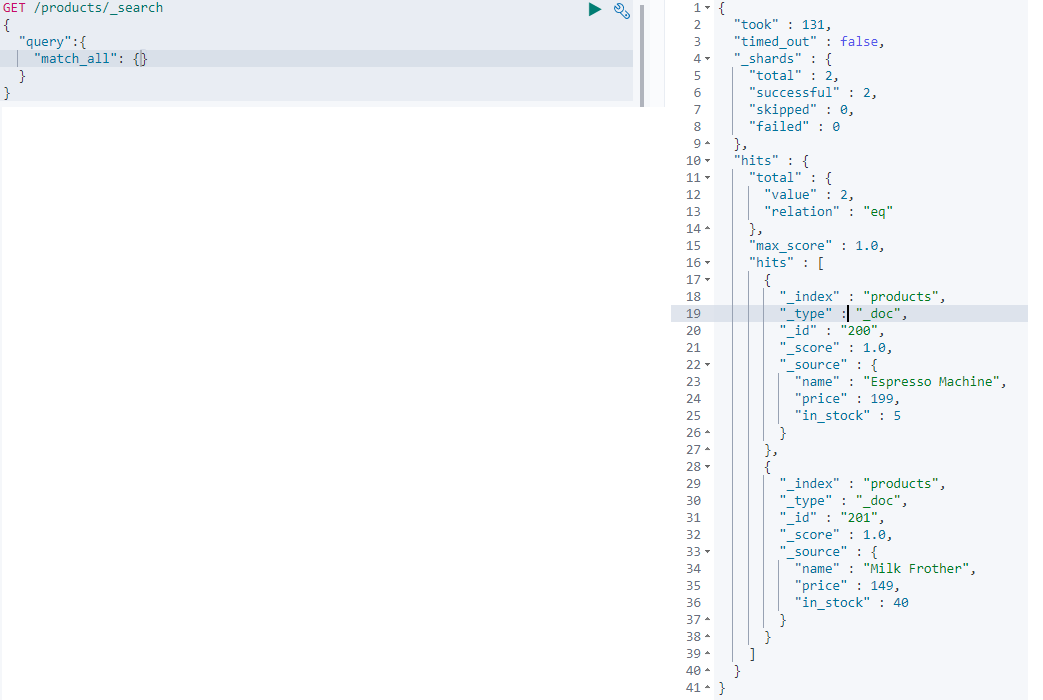
Lets see the results again.



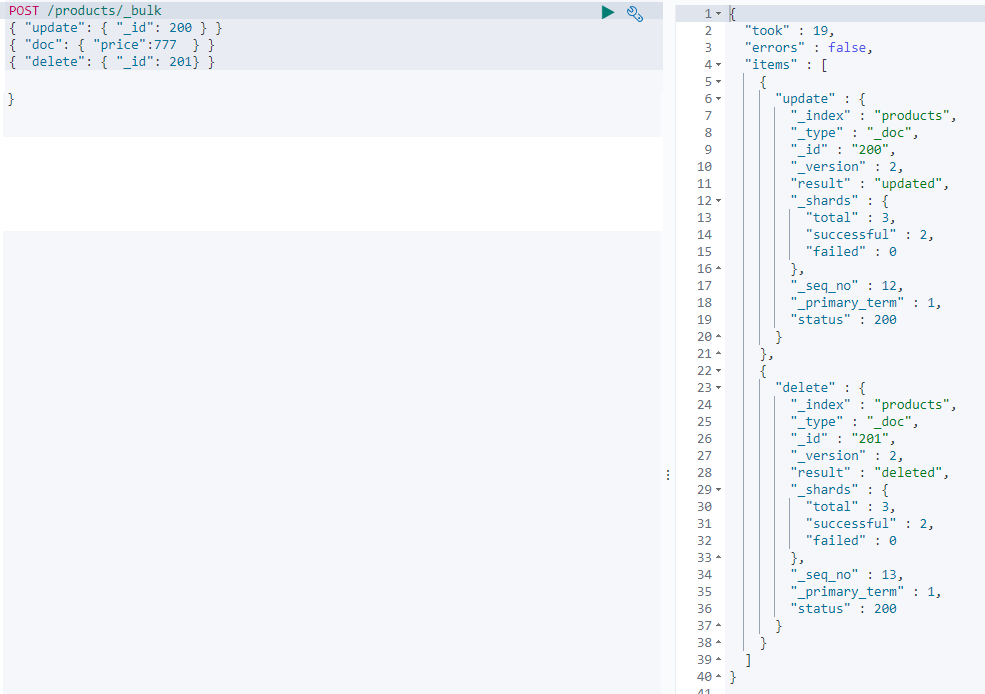
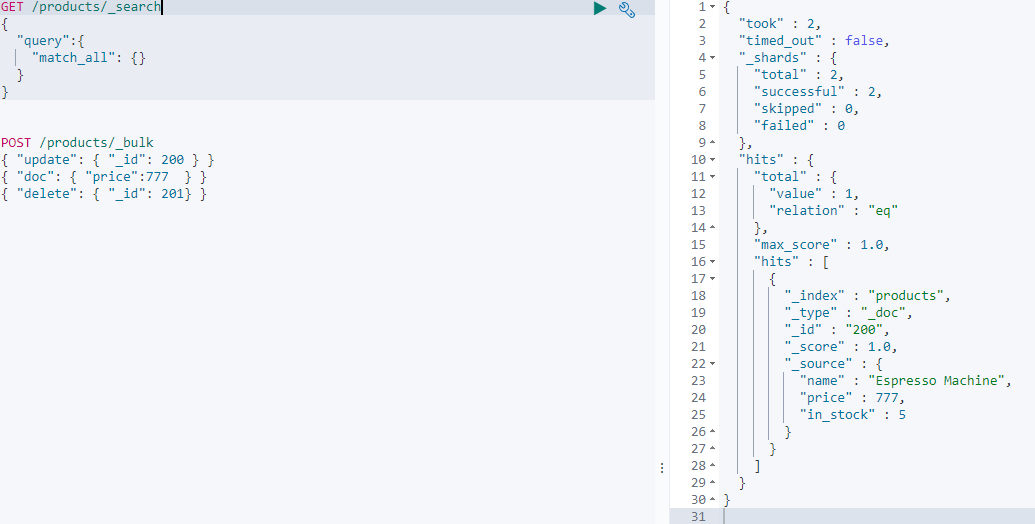
29. Delete documents based on condition



30. BATCH PROCESSING [create + index]

Endpoint : \_bulk  
Actions : index / create / update /delete  
// Create action: will fail if the index already exists  
// Index action: will create doc if it does not exist, else it replaces  
  
Lets retrieve the results of the above command using a search query.  


31. BATCH PROCESSING [update + delete]

Notice that if the index is same then – it can be given as part of the url instead as of a json object.  
The actions are all run against the products index.  
  
  
  
Run the search query again.  


Note: when using the bulk API –

The content-Type: application/x-ndjson

Each line must end with /n or /r/n including the last line [hit enter for last line as well]

If an action fails other still will be executed

The bulk is useful when a lot of write actions are to be done.   
Note: The bulk api supports optimistic concurrency control- include the if\_primary\_term and if\_seq\_no with in the action metadata.

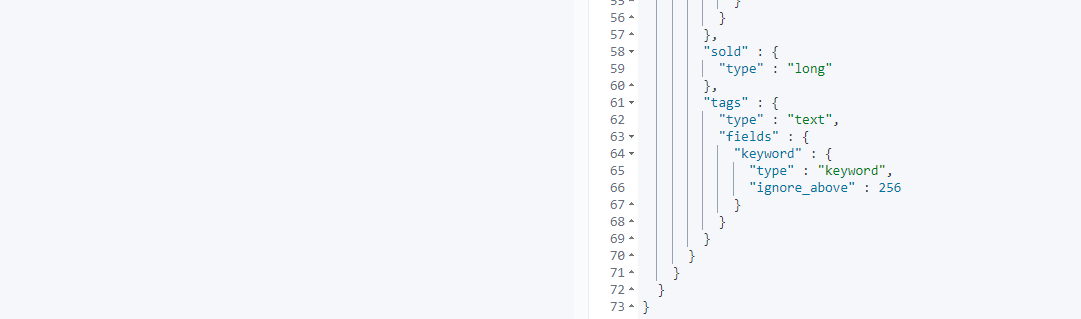
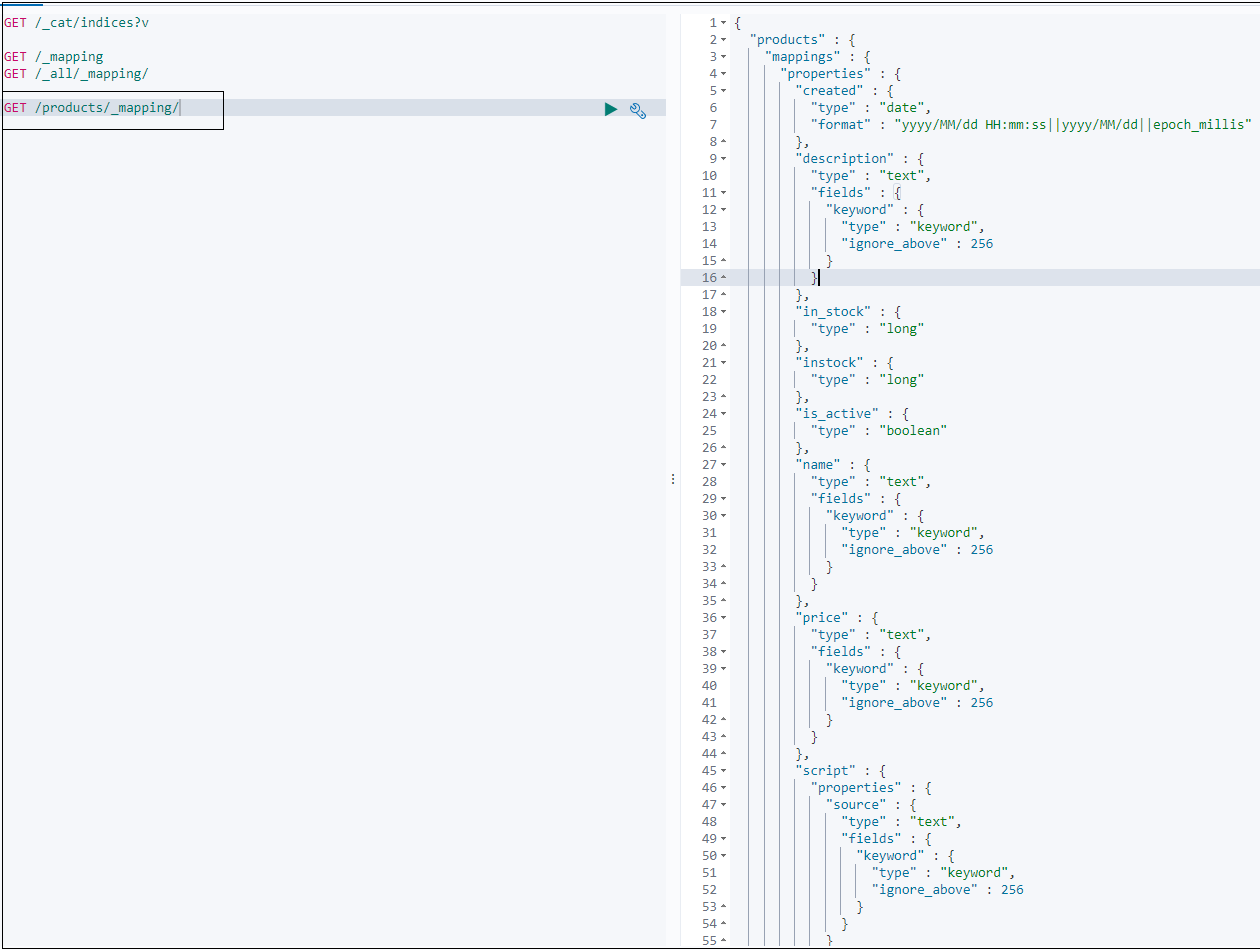
***32. Import Data into ES from a file using curl***

***curl -XPOST -u elastic: wp2ss2xAxTY5UnRUdgo2tUgW ‘http://24e374bd44bb4bf884cfd220f7f55cf0.ap-southeast-1.aws.found.io:9243/products/\_bulk?pretty -H "Content-Type: application/x-ndjson" --data-binary "@products-bulk.json"  
Note : binary : curl removes new lines and binary is used to preserve the new line.***

Above command has problems  
So used: POST /products/\_bulk command. + inserted the hwole content in the kibana editor

33. **MAPPINGS: HOW DOCUMENTS AND THEIR FIELDS SHOULD BE STORED AND INDEXED –** the point of doing this is **–** isto store andindex data in a way that is appropriate for how we want to search our data. [Something like schema for relational data bases –bad analogy ]

34. Dynamic Mapping:

ES has mapped the fields for us automatically.   


35. Meta Fields.

\_index : This field is added to document automatically.  
 It simply contains the name of the index to which a document belongs.  
 It is used internally – when querying documents with in an index.

\_id : stores the id of the document.

\_source : contains the original source document.

\_field\_names : contains the name of every field. The \_field\_names field is used to index the names of every field in a document that contains any value other than NULL. This field **was** used by exists query to find documents that HAVE or DON’T HAVE any non-null value for a particular field.  
Now the *\_field\_names* field only indexes the names of fields that have doc\_values and norms disabled. So the ‘exists’ query will not use the \_field\_names field, But it will be available for the fields which have either ‘doc\_values’ or ‘norm’ enabled.

\_routing : stores the value used to route a document to a shard. [custom routing]

\_version : used for versioning.

\_meta : may be used to store custom data, that is left untouched by elastic search.

36. Filed Data Types. – core / complex /geo /specialized

**Core** **DATA TYPES**.

Text Data Type.

* Used to index full text value such as descriptions. They are **analyzed.**
* Due to the nature of full text fields they are rarely used in sorting and aggregating.
* Text fields are stored in a way for performing optimal searches.
* i.e text fields are used for text that we want to search.

Keyword Data Type.

* Typically used for sorting and aggregating.
* They contain text but not full text : that’s becoz they are not analyzed.
* Values are stored exactly as defined at the time of adding documents to an index.
* Ex : a field for storing email addrs, category field [filter documents to find products with in a given category].
* These fields are used for values that we want to filter or use for aggregations.

Numeric Data Types.

* Float / long / short / byte/ integer/ **scaled\_float** / half\_float / double

Date Data Type.

* Can be represented as String / integer in sec / long in millisec since epoch
* The date format that should be used for fields when supplying a string value can be configured.
* Else a default format is used : which can either be a sting that optionally contains time or num of milliseconds since the epoch.
* Internally dates are stored as a long value – representing the number of seconds since the epoch.

Boolean Data Type.

Binary Data Type.

Range Data Type.

* It is used for Date values such as Date ranges or integer intervals like 10 to 20.
* You define a upper and lower boundary when indexing a document, by using the keywords gt,gte,lt,lte
* There is a query named ‘range’ which utilizes this data type.

**COMPLEX DATA TYPES.**

Object data type.

* Used for storing objects – plain json objects
* They contain nested objects as well
* Let us say you have a person object like you see here. When you index the below object you supply a normal JSON object.  
  {  
   “name”:{  
   “firstname”:”Big”,  
   “lastname”:”Bo”

}  
“Profession”:”Software Engineer”  
}

* But Elastic search flattens the object when storing. This means internally the object contains only key value pairs. And any nested objects are handled by adding dots to the key names to preserve the hierarchy of the objects.

Array Data Type.

* Any field in elastic search may contain zero or more values by default. Such as an array of numbers, string, objects etc.
* That’s possible without us having to explicitly declare this.
* You can also have an array of arrays- but note that arrays are flattened when indexed.
* All values must be of the same data type.
* [1,[2,3]] -> [1,2,3]

Array of objects.

* You cannot query individual objects independently of the other objects in the array – that’s because of how Elastic search flattens objects, - which again is because Lucene has no concept of inner objects.
* Let us say we have to person objects with in an ARRAY + Each object consists 2 properties.
* {  
  “persons”::

[

{“name”: “One”, “age”:10},  
 {“name”: “Two”, “age”:20},  
 ]  
}

* When Elastic Search indexes this array of objects, it flattens this object – Thus we have multiple values for the same keys which are persons.name and persons.age. [The field value will now be an array]
* {  
   “persons.name”:[“One”,”Two”],  
   “persons.age”:[10,20]  
  }
* Notice that the association between the objects is lost.
* If we want to search for a person name **One** age **10** – we cannot find it. The values of the objects are mixed together. There is no way of distinguishing one from the other.

Nested Data Type.

* When using Nested Data type, Each object is indexed as a hidden document. Ensuring that each object is independent.
* We do however need to use nested queries when searching through the objects.
* What happens when running nested queries is that they are executed against the nested documents as if they were separate documents, which is actually the case internally .
* The point is the documents are independent from each other – so we can query an array of objects that we probably expected in the first place – because the association between object values is preserved with the ‘nested’ data type.

**GEO DATA TYPES.**

Used for geographical data – latitude and longitude pairs.

Geo-point Data Type.

* Accepts latitude and longitude pairs in 4 different formats.
* {  
   “Location”:{  
   “lat”:12.123456,  
   “lon”:13.345678  
   }  
  }
* String with lat and lon separated by a comma  
  {  
   “location”: “12.345678,-12.345678”  
  }
* Geo hash   
  {  
   “location”: ”8ber34hhaf4425”  
  }
* Inverse of option 2 lon,lat  
  {  
   “location”: “-12.345678, 12.345678”  
  }

Geo –Shape Data Type.

* To store more complex geographical data – use this
* This data type helps you to form shapes of geographical points.
* Ex: you can store space of a geographical forest, borders of a city.

**Specialized Data Types**

Used for storing ip address, attachments etc

IP Data Type.

* You typically query fields of this type using the cidr notation

Completion Data Type.

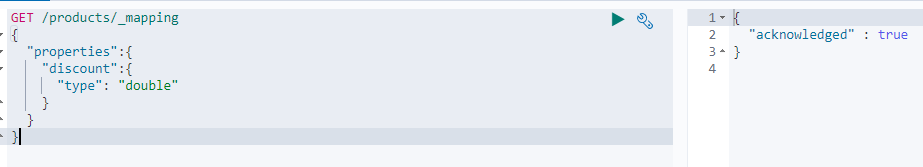
* This data type is about auto completion and search suggestions.
* That’s done with the help of suggesters – best approach.
* Using this data type enables very efficient lookups – as auto completion needs to be fast.
* Elastic search uses data structures that are slow to build but enables very fast lookups – and stores this in memory.

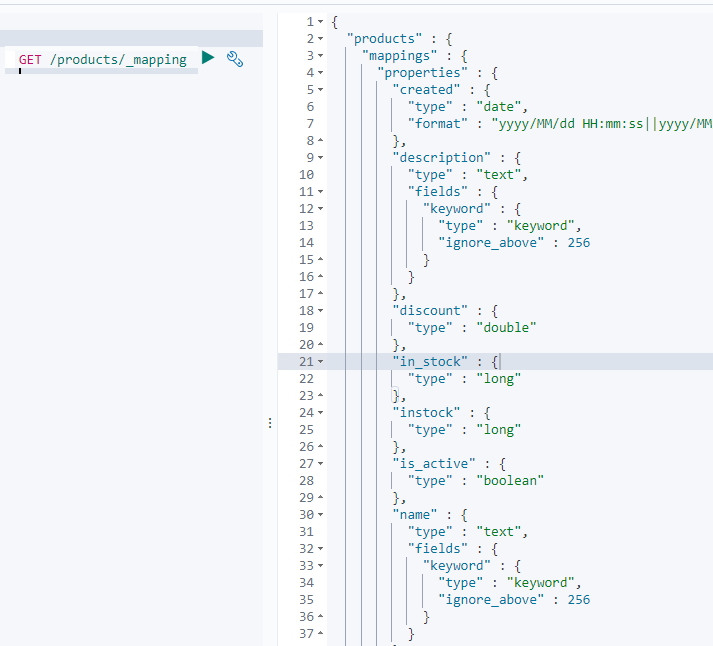
Attachment Data type.

* This data type is used for indexing documents that contain text and to make this text searchable
* Apache tika ..!?

36. **Adding mappings for new fields**

- Lets define a mapping for a new field named ‘discount’ with a data type of double

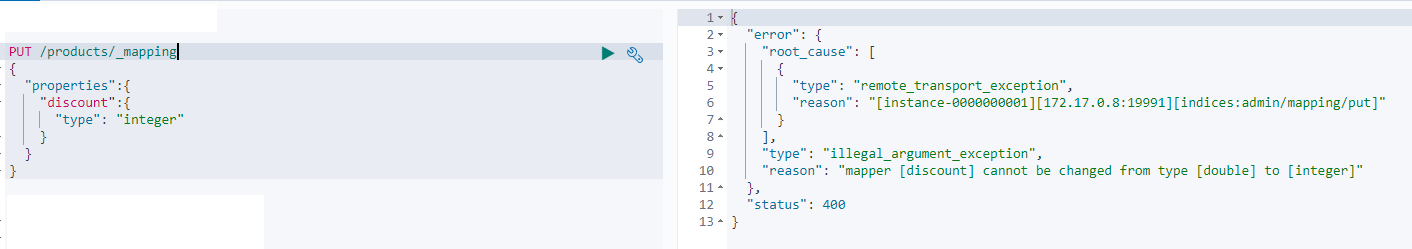
  
- Lets verify the mappings



* Now documents with a discount field can be added.

37.

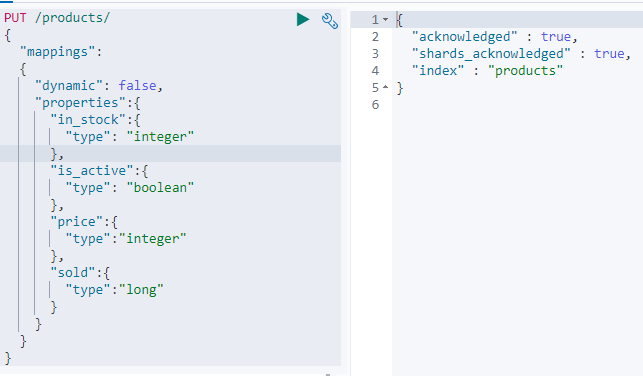
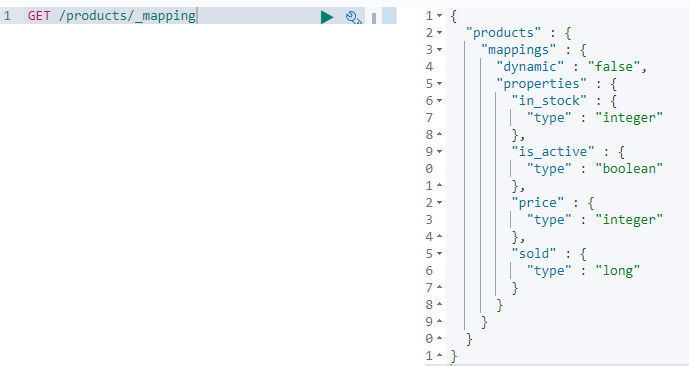
We get an error with the below query – we get an error that the type cannot be changed.  
Existing values for fields cannot be updated.



**What should be done ? [ \*\*\*\* changing existing mappings is not possible in elastic]**

* We should **delete the index**, create new mappings and re-index the data into the new index…



* Index gets created and mappings are added in the same query  
  
* Let’s first **add mappings for the simple data types**. [index will be created along with the mappings]  
    
    
  **We shall not import the data yet.**. as mappings for only the simplest fields have been added and mappings for other fields should also be added.
* There are few exceptions to the rules that mappings cannot be updated.

*You can add new properties for fields with the object data type*

*And you can also add additional mappings to existing fields.*

*An example of that is to add the* ***keyword type*** *to the text field.*

38. Properties that can be added to mappings.

Parameters that can be used to configure the mappings for fields.

**coerce parameter :** The data you index may come from various sources and it may not be in the same format you expect.ES handles this using coercion, - meaning that it convert values to the proper data type behind the scenes.  
Foe Ex : The document may contain “5” as a string for an integer field. Elastic search will try to clean up this data by coercing strings to numbers in this context.   
The above behavior can be disabled by setting the coerce parameter to false – in that case ES will reject documents that do not contain the correct data type.

**copy\_to parameter:** This parameter enables us to build a custom parameter with fields that we choose.

For Example : we can specify that the first name and the last name field values to be copied to a field called full\_name field.

{  
“first\_name”: { “type”:”text” , ”copy\_to”:”full\_name” }  
“last\_name”: { “type”:”text” , ”copy\_to”:”full\_name” }  
“full\_name : { “type”:”text” }  
}

**NOTE : When using the copy to parameter – it’s the values of the fields that are copied and not the terms that are output by the analyzer used for the field. Copied values will not show up within the \_source meta field.**

**properties parameter:**1. When adding a mapping when an index is getting created.  
2. It is used to map field mappings.

**norms parameter.**

When running search queries ES does not only determine whether or not a document matches: It also works on how well the document matches. – This is to give the user the most relevant search results first.   
{  
 “properties” :{  
 “full\_name”:{  
 “type”:”text”,  
 “norms”:false  
 }  
 }  
}

Elastic search stores some information that enables calculating relevance scores.

i.e ES stores so called normalization factors for fields that have scoring enabled.

These factors are referred to as norms. The norms parameter can be used to disable this information.

This would save disk space, but then ES looses the ability to sort documents by relevance.

For fields that are used for aggregations or just filtering out documents and there by not scoring documents – that would not be an issue.

Note that we cannot recreate norms without re-creating index.

**Format parameter**

The format parameter is used for specifying the date fields.

This can be done by specifying a custom format in the JODA format or one of the formats that are built into ES.

Custom format : “yyyy-MM-dd”, “epoch\_millis”, “epoch\_second”,……

Default format : “strict\_date\_optional\_time || epoch\_millis” [The default format accept time with an optional time OR number of seconds since the epoch]

**null\_value parameter**Replaces NULL values with the specified value.  
{  
 “properties”:  
 {  
 “discount”:  
 {  
 “type: “integer”,  
 “null\_value”: 0   
 }

}  
}

**fields parameter.**

This parameter is used for indexing a field in different ways for different purposes.  
[did you notice the mapping that ES automatically added for us based on the test data that we have imported.]



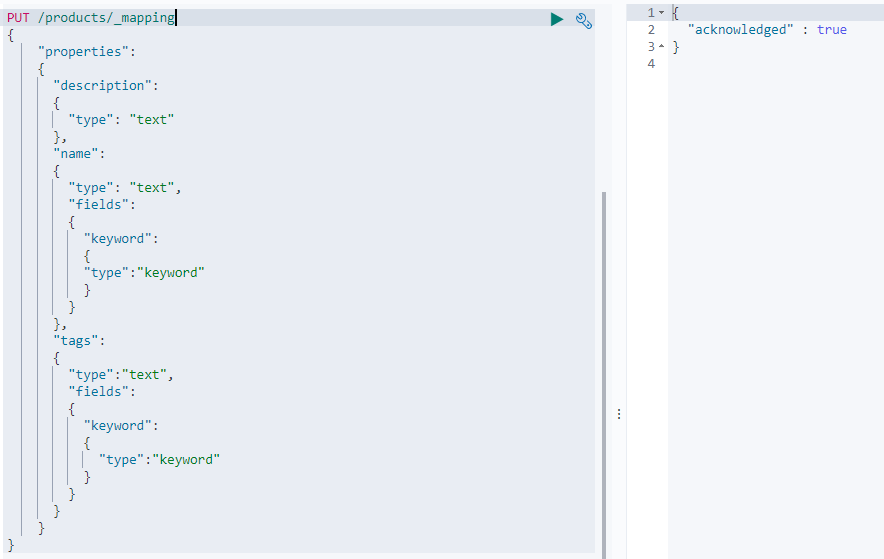
ES added an additional mapping with the “keyword” type for every “text” field.

The purpose of that is that you can use “text” fields for sorting and aggregation.

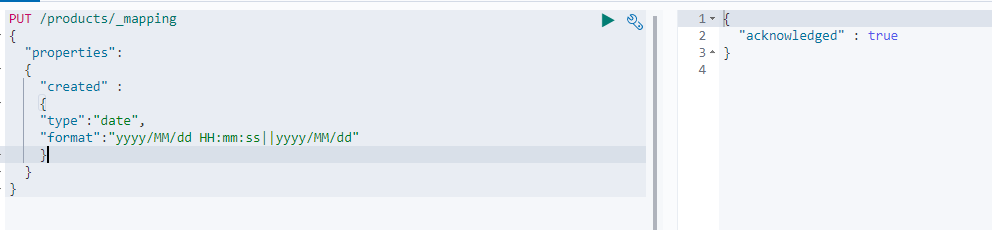
39. **ADDING MULTI-FIELDS MAPPINGS**

*Lets add more mappings for the* ***text fields*** – to do that we shall use the *fields* parameter to add additional keyword mappings and also the *properties* parameter.

Step 1:  
- if we need to add any parameters to a field say “description” you should do it with in a “description” object.  
- In this case we are not going to add a keyword mapping because that does not make sense.  
- Because we are not going to use the description field for any aggregations or any filtering that does not make any sense for a long field that contains many words.

  
Check if everything looks good.  

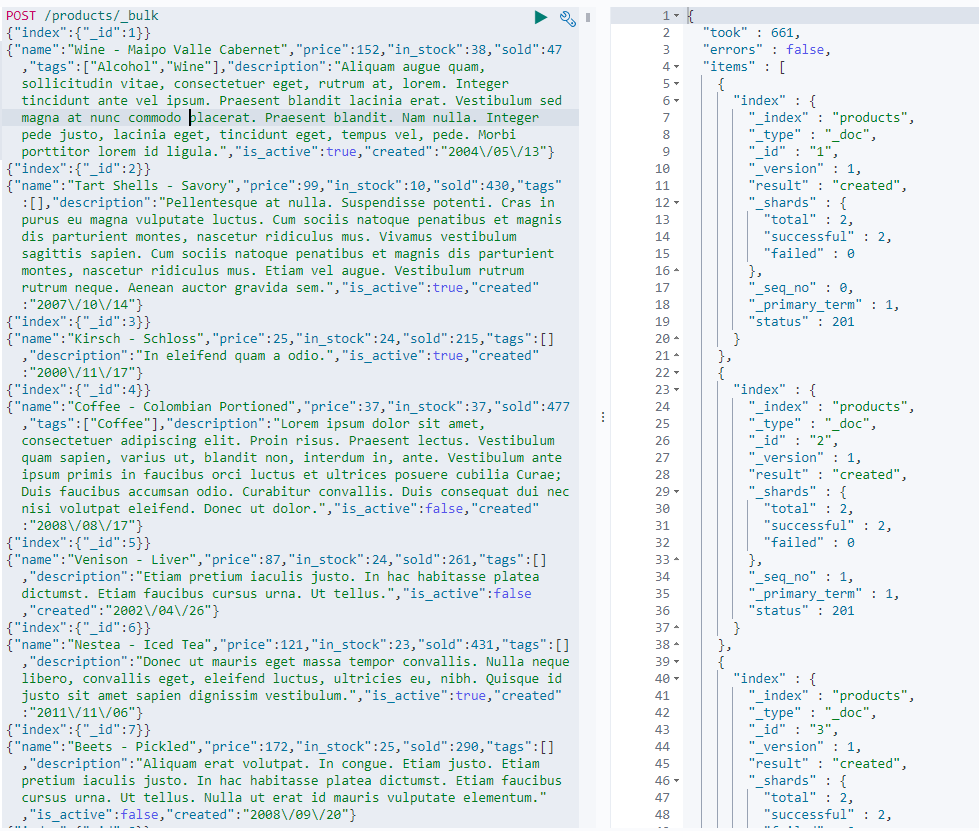

40. **Adding mapping for the created date field.**

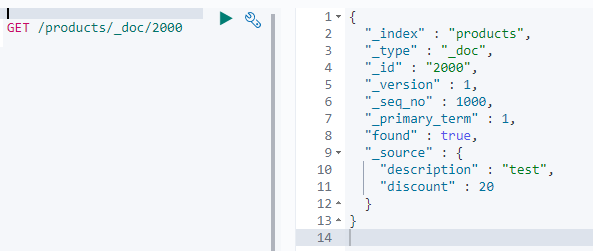

The default strict\_date\_optional\_time expects date with dashes but not slashes.

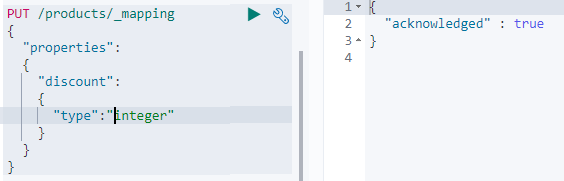
41. Add the test data from the below file. Curl can be used + \_bulk API can also be used.  
  

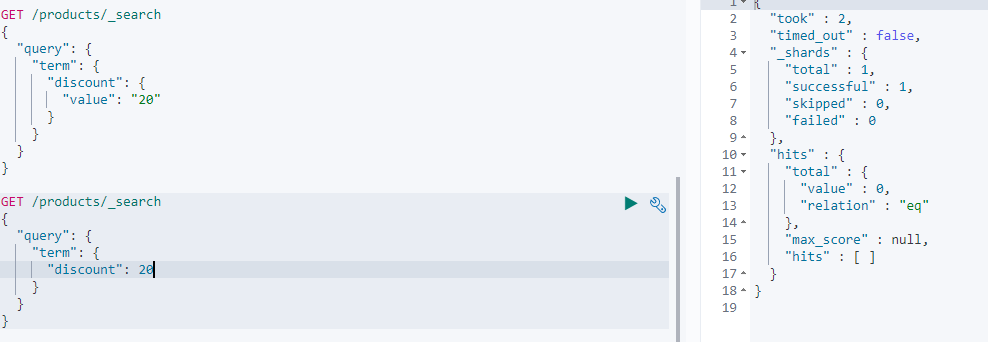

*Curl –H “Content-Type: application/json” –XPOST* [*http://localhost:9200/products/\_bulk/pretty?*](http://localhost:9200/products/_bulk/pretty?) *–data-binary “@filename.json”*



42. **PICKING UP NEW FIELDS WITHOUT DYNAMIC MAPPING**

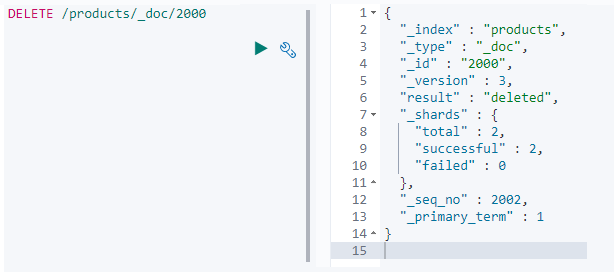
Add a mapping to the discount field – setting it to integer.  
  
  
Confirm if the document has been indexed correctly by performing a search query [search for “test” in the “description” field].  


Let try to find the document by matching documents that a value of 20 for the discount field.  


The document no longer matches.  
We have disabled dynamic mapping – through the “dynamic” parameter.  
**By doing that ES will just ignore fields for which there are no mappings.**[we can add mappings]  
The values will still be part of the \_source meta field, But the values will not be indexed, and will therefore not be searchable.

Dynamic mapping being disabled is not that uncommon, and this is because we don’t want developers adding field mappings as they wish.

**Any Work around ?**

* We could just start over and add the mapping before adding our data.
* Use update\_by\_query api  
  This api performs an update on documents – in this case we will use it to pick up the mapping that we added for the discount field
* When the below query is run : it gets a snapshot of the current state of the index, if a given document was changed between the ‘start of the query’ and ‘when the query starts to update that particular document’ – we get a conflict as the version of the document no longer matches.
* 
* The above query updated **all** of the documents in the index. It is also possible to add specific documents by adding a search query.
* The document has been re-indexed according to the new mapping, so the search query should match the new document now.  
  
* Note : Since we disabled dynamic mapping for an index : ES ignored the discount field when adding a document containing that field. The field was still part of the source meta field but was not indexed and therefore not searchable.
* This is the reason the document was not matched by the search query, even after adding the mapping.
* When adding mappings we need to refresh documents, when not using dynamic mapping, by using the update\_by\_query api – which caused the new mapping to be picked up.
* 

43. **Analyzers**

What does it mean when text is analyzed?

* The documents **full text fields** are run through an analysis process. [fields of the type ‘text’ and not the keyword fields which are not analyzed]
* It involves tokenizing text into terms, lowercasing text etc… to make text easier to search.
* You have full control over the analysis process.. as you can decide a particular analyzer.
* The results of the analysis is actually what is stored with in the index that a document is added to.
* More specifically the analyzed terms are stored in something called the inverted index.
* When we perform a search query we are searching through the results of the analysis process and not the documents as they were when we added them to the index.

44.

|  |  |  |  |
| --- | --- | --- | --- |
| Analyzer = | Character Filter + | Tokenizer + | Token Filter |

**Character filter:**  
 - **first zero or more character filters can be added** - The character filter receives the text fields original text and can then transform the   
 value by adding, removing or changing characters.  
 - Ex : Removing HTML markup.  
 **Tokenizer:**  
 - Splits the **text into individual tokens** which will usually be words  
 - If we have a sentence with 10 words, we will get an array of 10 tokens.  
 **- An Analyzer may have only one Tokenizer**. - By default a standard tokenizer is used – which uses a Unicode text –segmentation   
 algorithm [it splits by white space and removes symbols like  
 commas,semicolons,periods etc].  
 - Beside splitting standard text into tokens, tokenizers are also responsible for recording  
 the position of the tokens, including the start and end character offsets of the words  
 the tokens represent.  
 - This makes it possible to map the tokens to the original word – used in highlighting of  
 matching words.  
**Token Filters** :

- After the text is split into tokens it runs through zero or more Token Filters.  
 - A Token filter may add remove or change tokens, these are similar to a character filter,  
 but work with a token stream instead of a character stream.  
 - Lower case token filter: which converts all characters to lower case.   
 - Stop token filter: it removes common words which are referred to as stop words.  
 Ex : the a an at – they do not provide any value to a field in terms of   
 search-ability because each word gives a document very little  
 significance in terms of relevance.  
 - Synonym token filter : Which is useful for giving similar words the same meaning.  
 Ex : the words *nice* and *good* share the same semantics although they are  
 different words. By using this token filter you could match documents  
 containing the word nice even if you are searching for the word good.

45. COMPLETE EXAMPLE

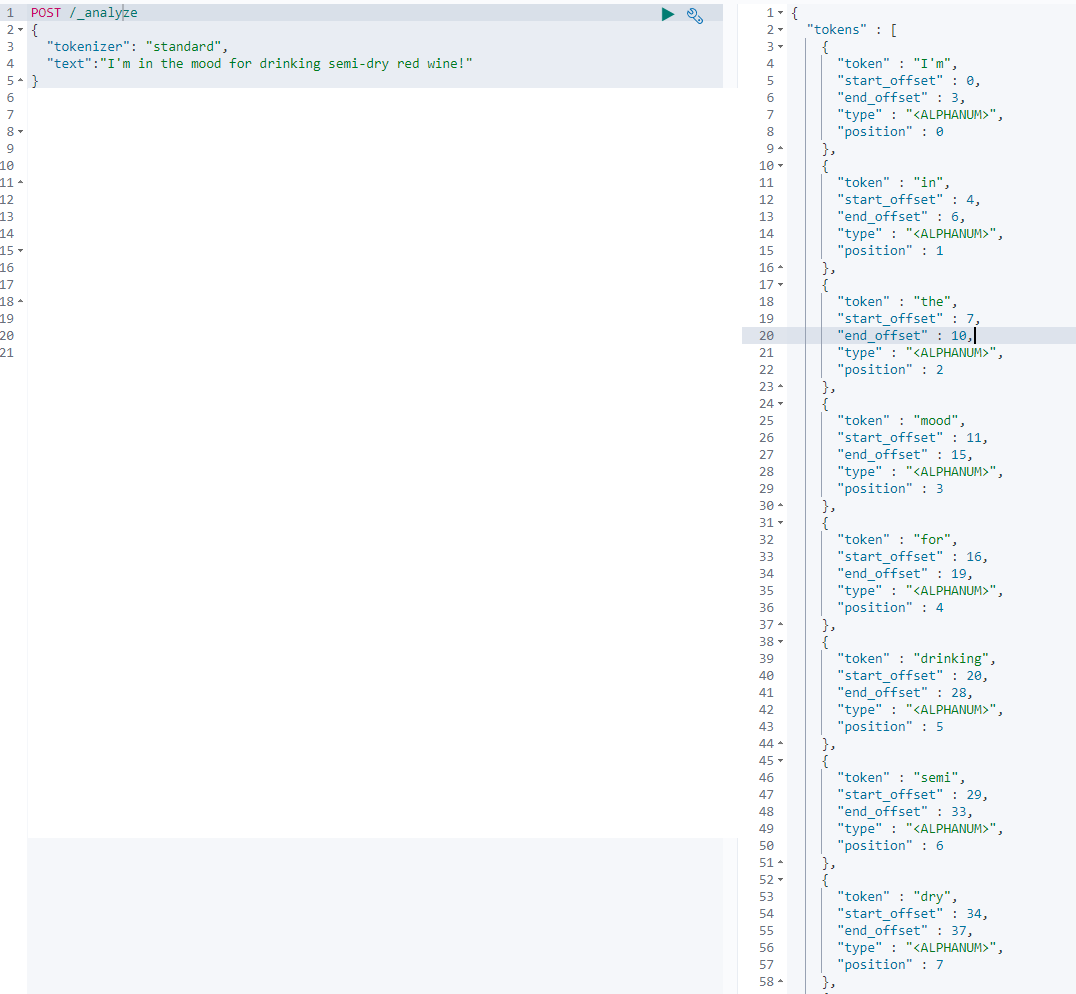
When ES detects a string field in a document it configures it as a full text field and applies the standard analyzer. With the standard analyzer there is no token filter, so the text input goes straight to the tokenizer. The standard analyzer uses a standard tokenizer, which filters out various symbols and split by white space.  
Input to tokenizer : I’m in the mood for drinking **semi-dry** red wine**!**output of tokenizer: [**I’m**, in, the, mood, for, drinking, semi, dry, red, wine ]

This array of tokens is sent to a chain of token filters.  
--- Standard + stop(disabled by default) + lowercase token filter.

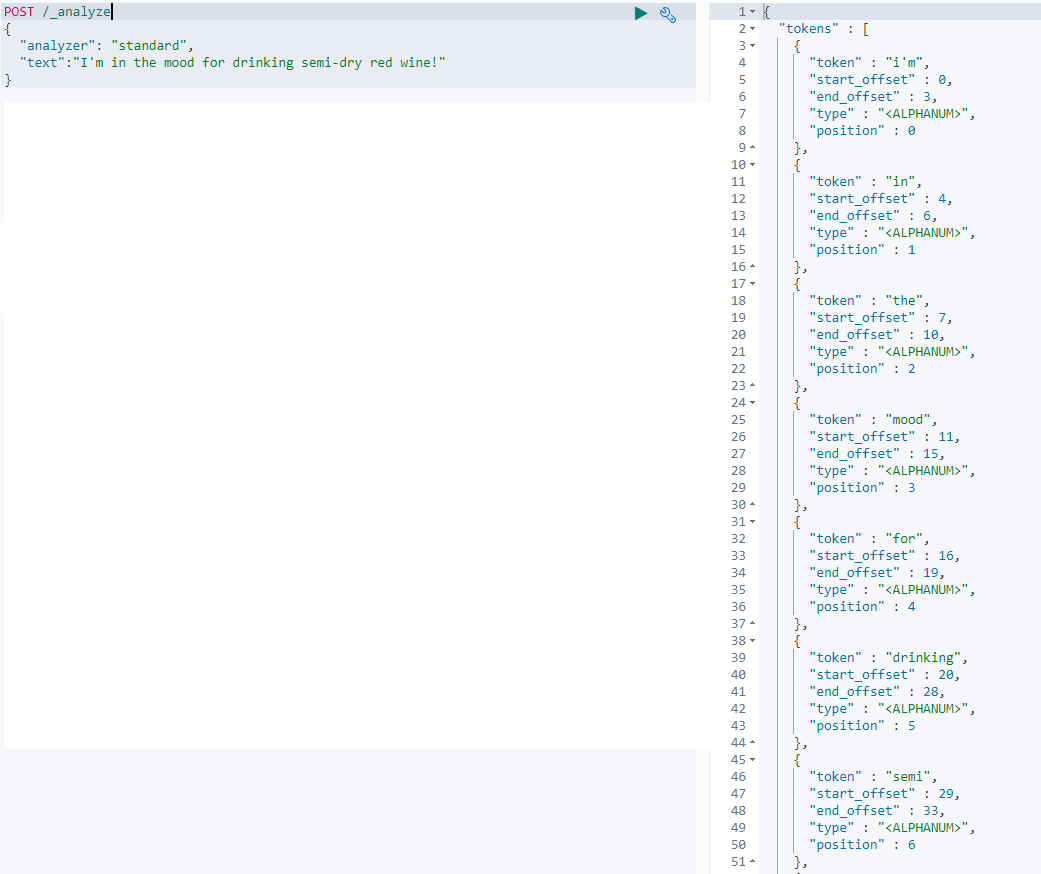
There is a analyze api which can be used to test the result of applying Character filters, tokenizers, and token filters and analyzers as a whole.

46. USING THE ANALYZE API with the standard analyzer

- since the standard analyze*r does not use a character filter* the first step is the tokenizer  
- apart from the tokens we get some additional information from the analyze api..like the character offsets.  
- If we don’t specify a tokenizer ourselves then a standard tokenizer is used.  
- *filter* key for token filer and *char\_filter* for character filter.




47 : WHAT HAPPENS UNDER THE HOOD – understanding the inverted index.

* What happens with the results of the analysis process?
* They are stored with in the inverted index.
* The purpose of an inverted index is to store text in a structure that allow for a very efficient and fast full text searches.
* When performing full text searches we are actually querying the n-inverted index – not the json document that we defined when indexing the documents.
* The cluster will have atleast 1 inverted index – there will be an inverted index for each full text fields per index.
* So if you have an index containing documents that contain 5 full text fields, you will have 5 inverted indices.
* An inverted index consists of all of the unique terms that appear in any document covered by the index.
* For each term, the list of documents in which the term appears is stored.
* ***Essentially an inverted index is a mapping between terms and which documents contain those terms.***
* Since the Inverted index works at the document field level, and stores the terms of a given field, it does not need with different fields.

“The best pasta recipe with pesto” + “The delicious pasta carbanara recipe”

The following table shows what the inverted index looks like.  
The terms from both of the titles have been added to the index.

|  |  |  |
| --- | --- | --- |
| Term | Document #1 | Document #2 |
| Best | X |  |
| Carbanara |  | X |
| Delicious |  | X |
| Pasta | X | X |
| Pesto | X |  |
| Receipe | X | X |

For each term we can see which document contains the term – which enables ES to efficiently match documents containing specific terms.  
A part of what makes this possible is that the terms are sorted.  
The terms with in the index are the results of the analysis process that we saw earlier, most symbols have been removed and tokens have been lowercased. This obviously depends on the analyzer.

Performing a search involves a lot of things such as relevance.  
The first step of a search query is to find the documents that match the query in the first place.  
This is how the inverted index is used while performing search queries.

Inverted Index also holds information that is used internally such as for computing relevance.  
Ex : number of documents containing each term   
 number of times a term appears in a given document   
 The average length of a field etc.

(Stemming of words and synonyms will also be applied to the inverted index.)

|  |
| --- |
| **CHARACTER FILTERS** |

48.

OVERVIEW OF THE BUILT IN CHARACTER FILTERS.

There are 3 character filters available.  
 HTML strip character filter.  
 - Used for striping html from text + decode html entities.  
 - Ex : getting rid of html from comments.

Mapping Character filter.  
 - This filter replaces values based on a supplied list of values and their replacements.  
 - Ex : we could define certain characters to be replaced by smiley faces.  
 Pattern Replace Character filter.  
 - It is similar to Mapping character filter, Except that it does the matching based on regular  
 expressions.  
 -It also allows you to access the matched values with capture groups and use those with in the   
 replacement values.  
 - In this example we have a regex that matches a series of characters potentially followed by a  
 hypen   
 - Ex :   
 Pattern : ([a-zA-Z0-9]+) (-?)  
 Replacement : $1

|  |
| --- |
| **Tokenizers** |

49.

OVERVIEW OF THE BUILT IN *Tokenizers*

Tokenizers are grouped into three main categories.

|  |  |  |
| --- | --- | --- |
| WORD ORIENTED TOKENIZERS | PARTIAL WORD TOKENIZERS | STRUCTURED TEXT TOKENIZERS |

**WORD ORIENTED TOKENIZERS:** This group of tokenizers are typically used for tokenizing full text into words. So these tokenizers work at the word level, thus the o/p will be easily readable by humans.  
 1. Standard Tokenizer: it splits text into terms based on white spaces and certain symbols. It also removes certain symbols which will not be part of the output.  
Example:   
Input to tokenizer: I’m in the mood for drinking **semi-dry** red wine**!**Output of tokenizer: [**I’m**, in, the, mood, for, drinking, semi, dry, red, wine]  
  
 2. Letter Tokenizer: it splits text into terms whenever it encounters a character that is not a letter.  
Example:   
Input to tokenizer: I’m in the mood for drinking semi-dry red wine**!**Output of tokenizer: [**I,m**, in, the, mood, for, drinking, **semi, dry**, red, wine ]   
  
 3. Lower case Tokenizer: It works like the letter tokenizer, but additionally it lowercases terms.  
Example:   
Input to tokenizer: I’m in the mood for drinking semi-dry red wine**!**Output of tokenizer: [**i,m**, in, the, mood, for, drinking, **semi, dry**, red, wine ]   
  
 4. Whitespace Tokenizer: It divides text into terms when encountering whitespace characters. This tokenizer does not split by symbols like the other tokenizers.   
Example:   
Input to tokenizer: I’m in the mood for drinking semi-dry red wine**!**Output of tokenizer: [**I’m**, in, the, mood, for, drinking, **semi-dry**, red, wine**!**]

5. UAX URL Email Tokenizer: It works the same way as the standard tokenizer. But it preserves email addresses and URLs.  
Example:   
Input to tokenizer: Contact us at [abc@def.com](mailto:abc@def.com) or 12345  
Output of tokenizer: [Contact, us, at, [abc@def.com](mailto:abc@def.com), or, 12345]

**PARTIAL WORD TOKENIZERS:** Thesetokenizer**s** are used for breaking text into small fragments. This is used for partially matching terms.  
 1. N-Gram Tokenizer: This tokenizer breaks text into terms similar to the standard tokenizer + and then emits N-grams(it’s like a sliding window that moves across a word) of the specified length.  
Let us say you configure an N-gram with min-characters of 2 and max-characters of 10, Then each N-Gram will be between **2** and **10** characters in length.  
Example:   
Input to tokenizer: “Red wine”  
Output of tokenizer:   
Step 1: The tokenizer splits the input text into words using the white space character separating the words. Then N-grams are generated for the first token.  
First ‘2’ characters are taken as it’s the min-characters configured that should be included and then it increases the number of characters till it reaches the ‘end of the term’ or ‘the max-char count ‘10’ ’.  
**[Re,Red]**  
Step 2 : Then the ‘cursor’/’start index’ moves to 1 position forward [from R to e in the word Red] and then starts to generate the N-Grams from the ‘e’ character.  
**[ed,]**

Step 3 : The tokenizer moves the cursor 1 position forward to the letter ‘d’, but no N-Grams can be generated, So the cursor moves to the next word ‘wine’ and starts generating N-grams.  
**[wi,win,wine,in,ine,ne]** 2. Edge N-Gram Tokenizer: It’s similar to the n-gram tokenizer except that it emits n-grams of each word – beginning from the start of the word. i.e N-Grams always start from the beginning of the terms.  
Example:   
Input to tokenizer: “Red wine”  
Output of tokenizer: [ Re,Red, wi,win,wine]  
**Application** : this edge n-gram tokenizer has often been used for auto-completion.[better to use suggesters – mostly for auto-completion]  
  
**STRUCTERED TEXT TOKENIZERS:** These are used for email addresses, zip codes, identifiers and more  
 1. Keyword Tokenizer: All it does it take the input and return it as 1 token  
 2. Pattern Tokenizer: It uses a regular expression to ‘match token separators and use those to split text into terms’ or ‘use capturing groups and use the matched text as terms’  
 3. Path Tokenizer: This tokenizer splits hierarchical values [eg: file system paths] by a path separator and emits a token for each component. This tokenizer is used when the input is a tree structure. We can configure the de-limiter that is used to split the input which is ‘/’ by default. We can also specify a replacement value.  
Example:   
Input to tokenizer: /path/to/some/directory  
Output of tokenizer: [ /path, /path/to, /path/to/some/, /path/to/some/directory ]

|  |
| --- |
| **TOKEN FILTERS** |

50.

**Note :** **The standard tokenizer** is used to split text into tokens, before applying a token filter.  
 **Standard Token Filter:** It does not do anything, just acts as a placeholder for future versions.  
Example:   
Input to standardtokenizer: I’m in the mood for drinking **semi-dry** red wine**!**Output of standardtokenizer: [I’m, in, the, mood, for, drinking**, semi, dry**, red, **wine]**Output of token filter: [**I’m**, in, the, mood, for, drinking, semi, dry, red, wine]

**Lowercase Token Filter:** It lowercases terms  
Example:   
Input to standardtokenizer: I’m in the mood for drinking **semi-dry** red wine**!**Output of standardtokenizer: [I’m, in, the, mood, for, drinking**, semi, dry**, red, **wine]**Output of token filter: [i**’m**, in, the, mood, for, drinking, semi, dry, red, wine]   
  
**Uppercase Token Filter:** It uppercases terms  
Example:   
Input to standardtokenizer: I’m in the mood for drinking **semi-dry** red wine**!**Output of standardtokenizer: [I’m, in, the, mood, for, drinking**, semi, dry**, red, **wine** ]Output of token filter: [ I’M,IN, THE,MOOD,FOR,DRIKING….. ]

**NGram Token Filter:** If you wantuse adifferent tokenizer other than ngram tokenizer then you can use this token filter to do the same job.  
**Edge NGram Token filter:** this behavior is also same as the edge n-gram tokenizer.

**Stop Token filter**: removes stop words.  
Example:   
Input to standardtokenizer: I’m in the mood for drinking **semi-dry** red wine**!**Output of standardtokenizer: [I’m, in, the, mood, for, drinking**, semi, dry**, red, **wine]**Output of token filter: [I’m, mood, drinking, semi, dry, red, wine]

**Word delimiter token filter**: This token filter splits words into sub words based on a number of rules that can be enabled or disabled. It splits into terms at every non alpha numeric character.  
This token filter splits – when encounters a hyphen  
 – when case transitions from lowercase to upper case or vice versa.  
 – The filter also splits when switching from letters to numbers or vice versa.  
 – If an apostrophe is found at the end of the word, these two are also removed.  
  
Example:   
Input of token filter: [Wi-Fi,PowerShell,CE1000,Andy’s]  
Output of Token filter : [Wi,Fi,Power,Shell,CE,1000,Andy]

**Stemmer token filter:** It stems words based on a default language + it reduces words to the base form – to make sure that a document is matched regardless of which form a word is in.  
Example:   
Input to Token filter: [I’m, in, the, mood, for, drinking, semi, dry, red, wine]   
output of Token filter: [I’m, in, the, mood, for, **drink,** semi, dry, red, wine]

**Keyword marker token filter:** If you want to prevent certain words from being stemmed, it can be do with this filter. We just need to provide a list of words to protect them and the stemmer filter will protect it.

Example:   
Protected Terms: [drinking]  
Input to Token filter: [I’m, in, the, mood, for, drinking, semi, dry, red, wine ]  
output of Token filter: [I’m, in, the, mood, for, **drinking,** semi, dry, red, wine ]

**Snowball token filter**: Another token filter used for stemming. It enables u to make use of stemming algorithms implemented in **snowball** (which is a string processing programming language, which is used to implement stemmers).

**Synonym token filter**: This token filter is used to handle synonyms. If we want to define the word delighted as something similar to happy. And therefore we don’t care which ever word the user searches for and we want the document to match regardless of whichever words it contains.[which can be accomplished by supplying a configuration file with words that have the same meaning]  
Example:  
Input to token filter : [I, am, very , happy]  
output to token filter : [I, am, very , happy/delighted]  
When run through the synonym token filer synonyms will be injected as TERMS .  
In reality the terms are not stored with a slash in the inverted index. But the synonyms are stored in the same position as the original terms  
Trim Token filter: which trims any trailing or leading whitespace.  
Length Token filter: Which removes token that are either too large or short based on min and max lengths  
Truncate token filter: to truncate tokens to a maximum length you specify.

51. Default Analyzers.  
 Analyzer: It is an orchestration of CF + Tokenizer + TF.

**Standard Analyzer**: It divides text into terms using word boundaries. It Removes Punctuation + lowercases terms and optionally removes stop words.   
Standard Tokenizer + Standard token filter, Lowercase Token filter, and optionally stop token filter.  
Input: “I’m in the mood for drinking semi-dry red wine!”  
Output: [ i,m,in,the,mood,for,drinking,semi,dry,red,wine ]

**Simple Analyzer:** It onlyconsists of a *lower-case tokenizer*. It splits input into terms whenever it encounters characters that are not letters + lowercases terms (this is normally done by a token filter)  
Input: “I’m in the mood for drinking semi-dry red wine!”  
Output: [i,m,in,the,mood,for,drinking,semi,dry,red,wine]

**Stop Analyzer:** It worksthe same way as a simple analyzer + it removes stop words.It consists of a lowercase tokenizer + stop token filter  
Input: “I’m in the mood for drinking semi-dry red wine!”  
Output: [i,m,mood,drinking,semi,dry,red,wine]  
Note that this does a similar job as Standard Analyzer + Stop Token filter  
  
**Language Analyzers:** These analyzers are used for analyzing text in specific languages. They provide an easy way of enabling stemming stop words and more – without having to define these parts explicitly.  
 English Analyzer: The standard tokenizer is used.  
 Input: “I’m in the mood for drinking semi-dry red wine!”  
 Output: [I’m,mood,drink,semi,dry,red,wine]  
 in, the, for : stop token filter  
 drinking – drink : stemmer token filter

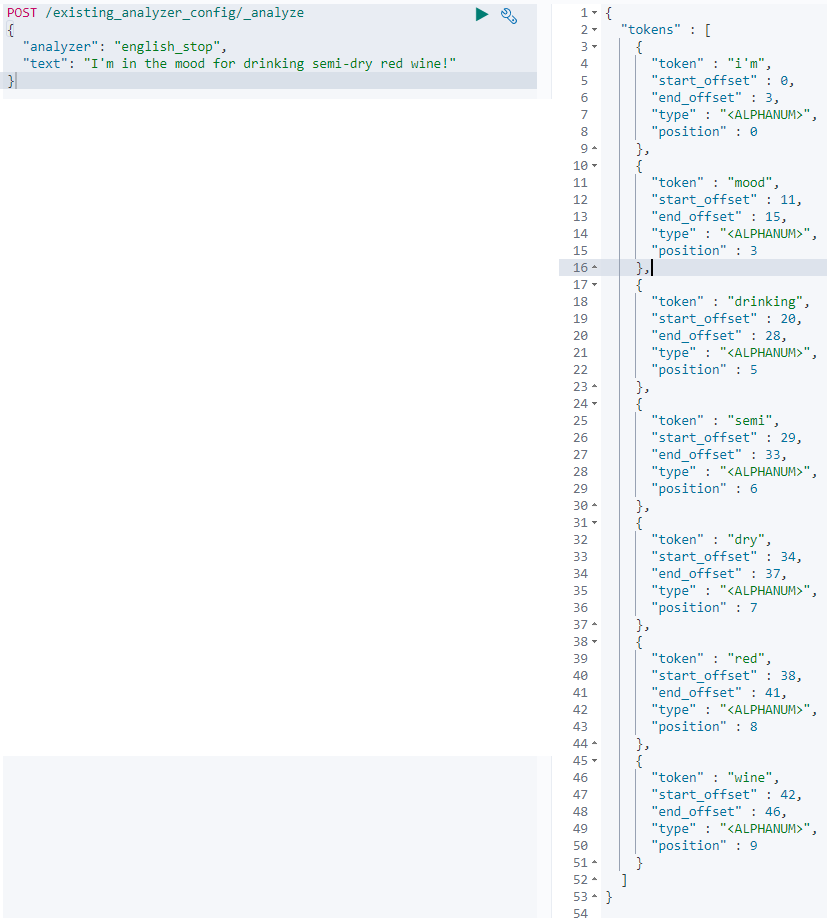
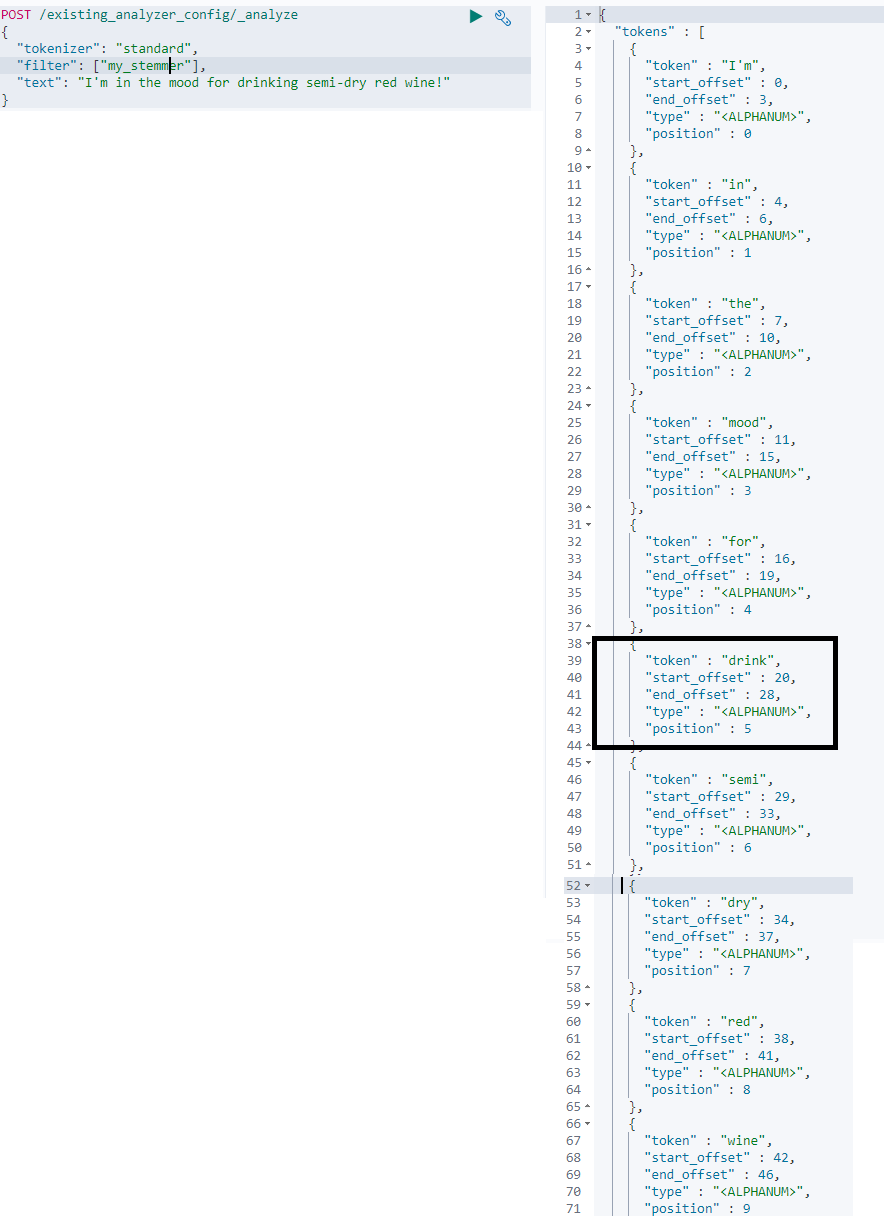
**Keyword Analyzer:** It takes all of the input and returns it as a single term something that it is done by keyword tokenizer.

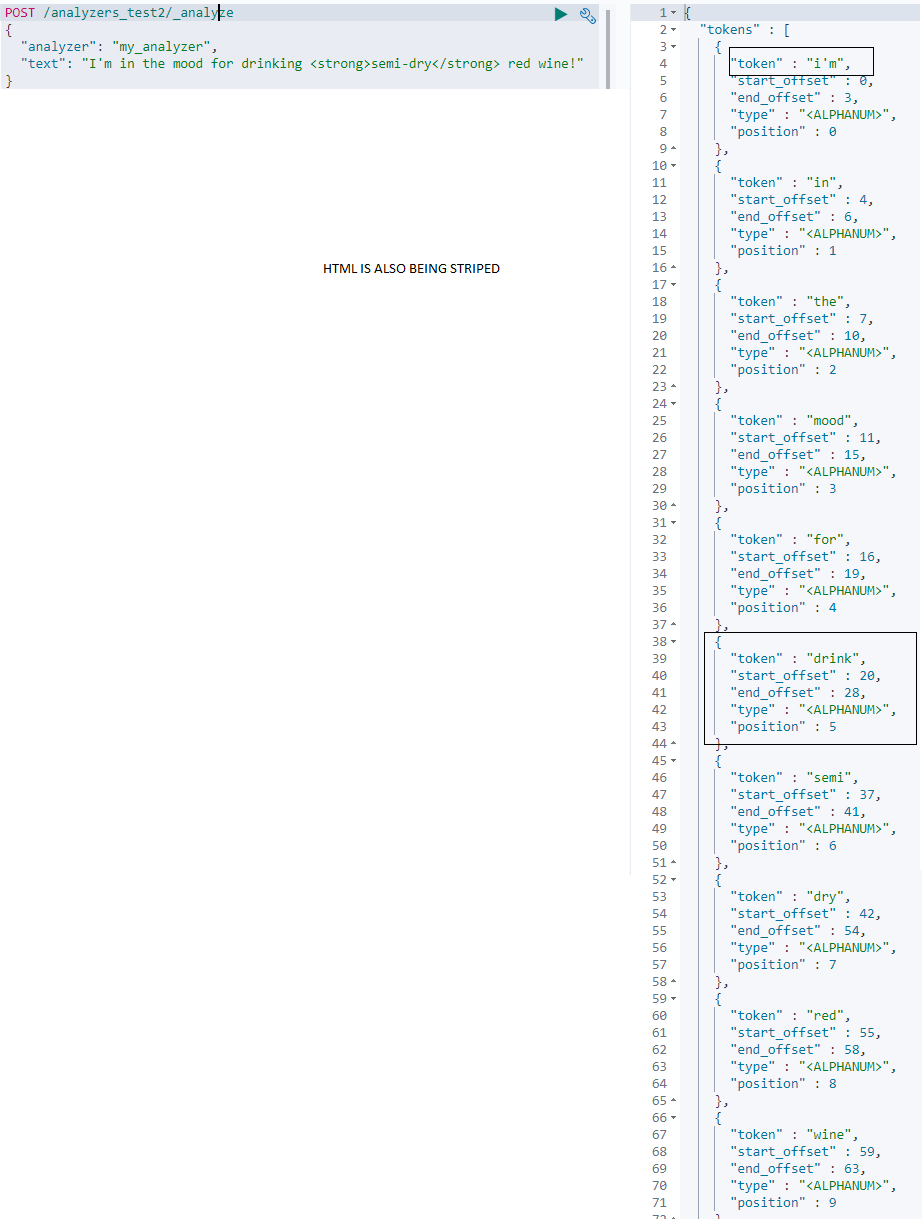
**Pattern Analyzer**: It splits text into tokens by matching token separators with the supplied regular expression, which is done with the help of pattern tokenizer + it uses lowercase token filter + optionally uses a stop token filter.  
Input to analyzer: “I, like, red, wine”  
output of analyzer: [I, like, red, wine]

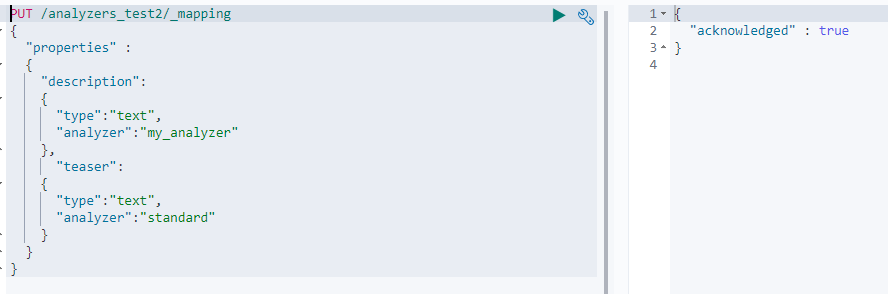
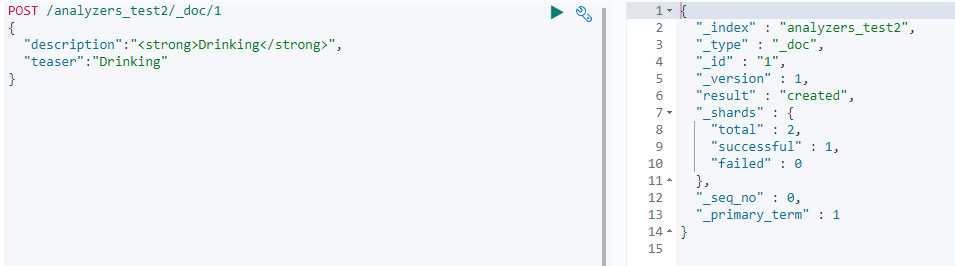
**Whitespace Analyzer:** It simply splits text into token when encountering a whitespace character, which uses white space tokenizer for that.  
Input: “I’m in the mood for drinking semi-dry red wine!”  
Output: [**I’m**,in,the,mood,for,drinking,semi-dry,red,wine!]

52. CONFIGURE THE BUILT-IN ANALYZERS, TOKEN FILTERS and MORE.

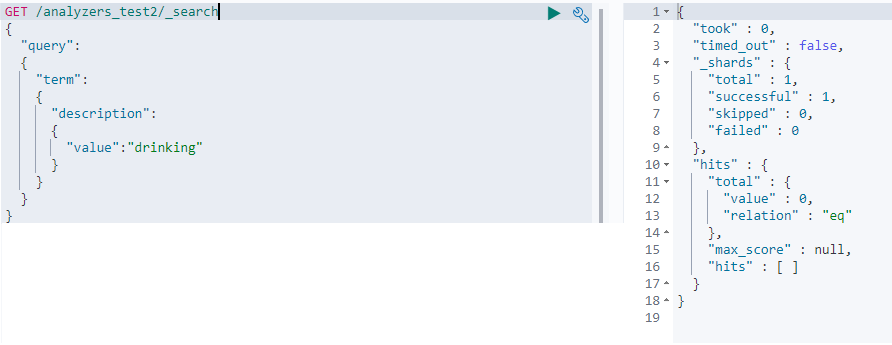
We shall configure the standard analyzer – to remove stop words, which causes it to enable the stop token filter.

We shall define an analyzer at index creation time, with a “settings” object.  
  
We are using a custom analyzer which uses the standard analyzer, and we are configure it through parameters. – Now we can use this custom analyzer in field mappings.  
+  
We configure a token filter as well. [ “tokenizer”, “char\_filter” keys can also be used to configure a tokenizer and a character filter ]  
  
Test if the analyzer works:  
  
Check if the token filter works.  


53.   
Creating a custom analyzer from scratch.  
  
  
Verify if the analyzer works via the analyze api.  
  
[There is a problem with the ‘standard’ token filter above – so remove it and everything works fine.. AND NOTE THAT WE ARE USING A DIFFERENT INDEX]  
  
  
As the analyzer is good to go. Next use it in field mapping.

54. USING ANALYZERS IN FIELD MAPPINGS.  
  
Below we added mappings for two fields.  
  
Add a document to this index.  


Perform a search query: search for ‘teaser’ field having a value drinking.  
After we run the query we can see that the document being matched.  

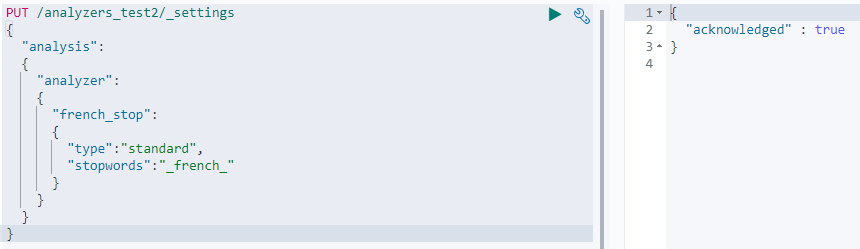

If we change the query to search the ‘description’ field instead.  
The document no longer matches the query.  
coz : description field uses “my\_analyzer” which uses an English stemmer.  
Thus we can say that our analyzer is indeed doing its work when indexing new documents.

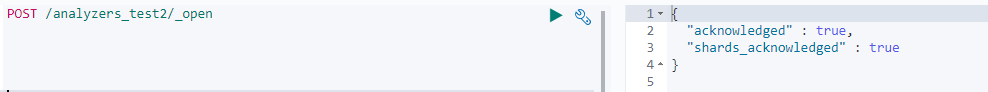
55. Adding analyzers definitions to an **existing** index.

First we have to close an index by using the *close* API.  


If a search request is sent to this api : below is the result  


Let’s add an analyzer to this **index** using the *\_settings* api.



Let’s re-open the index- to send read and write requests once again.  


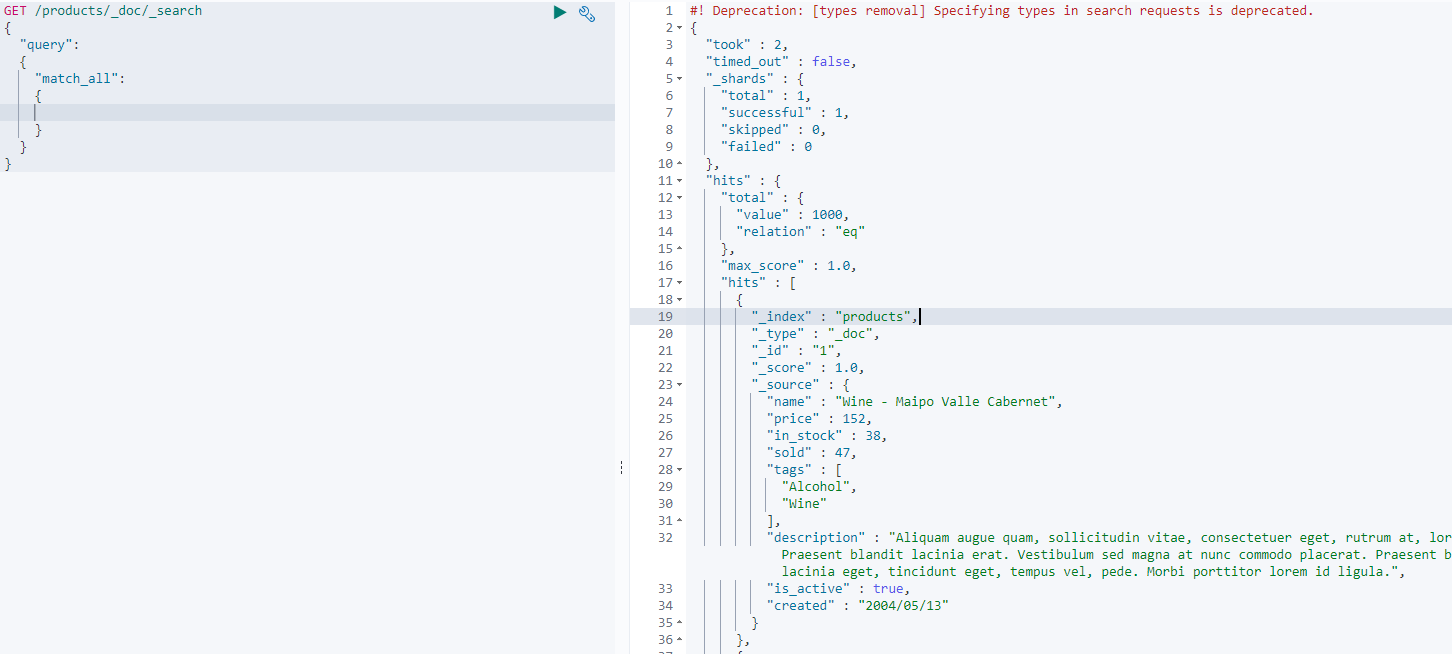
NOW IF WE NEED TO USE THIS ANALYZERS FOR AN EXISTING MAPPING, WE CAN NOT, AS WE NEED TO DELETE AND CREATE THE INDEX ALL OVER AGAIN, AND RE-INDEX THE DATA AFTER HAVING ADDED THE CORRECT MAPPING.

It is not a good practice to remove stop words. As it did not help much in relevance scores. But as *ES* has evolved, so has its relevance algorithm which handles the stop words in a netter way.

56. SEARCHING

Performing Search using query DSL: we provide a json object in the request body instead of in the request URI.

There are two main groups of queries in query DSL.  
1. Leaf queries: these queries search for values with in particular fields.  
2. Compound queries: They consist of multiple leaf queries compounded using a bool query.  
Ex : 1000 docs matched

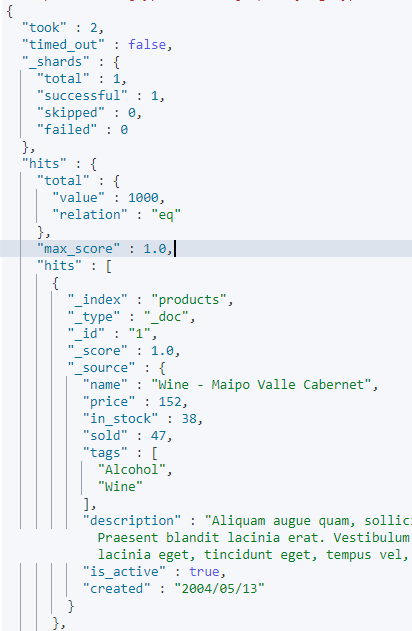


57. how elastic search searches for data.  
From Developer point of view: A client communicates with Elastic search cluster by sending search queries over http. The cluster does the searching by using the index and the search query that is mentioned in the http request + responds with the results.  
Digging little more: Suppose there is a cluster which consists of three nodes, containing 1 index distributed across 3 shards A,B,C. Each shard has 2 replicas. So each replication group consists of a primary shard and 2 replicas.  
  
CLUSTER

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | SHARD A | | REPLICA C1 | | REPLICA B2 |   3 - Nodes | |  | | --- | | SHARD B | | REPLICA A1 | | REPLICA C2 | | |  | | --- | | SHARD C | | REPLICA B1 | | REPLICA A2 | |

When a client sends a search request to the cluster, which ends up on a node containing shardB.  
This node is the so called co-ordinating node. [This node broadcasts the requests to all shards in the index that the query refers to] This node is responsible for sending queries to other nodes + assembling the results + responding to clients. By default every node in the cluster may act as a co-ordinating node and may receive http requests.  
  
Since the co-ordinating node itself contains a shard, which should be searched, the node will perform the query itself.

Note : When you retrieve a single document by its ID – In this case the request is routed to the **appropriate** shard, instead of being broadcasted to all of the index’s shards.

58. Explaining Search Results  
  
took : number of milli seconds the query took to execute  
timed\_out : flag indicating if the search request timed out or not.  
\_shards : this obj contains the total number of shards that was searched + the number of shards that  
 completed successfully or failed.  
hits: it contains the search results  
 total property : total number of documents that matched the search criteria.  
 hits property: its an array containing the matched documents.  
 max\_score property : contains the highest score for any of the matched documents.by default  
 matches are sorted by relevance score so the first document contains this  
 number.  
\_score : Each matching document contains this \_score property, indicates how well the document  
 matched the search query.  
  
  
  


59. Understanding relevance scores.

- it depends on the search query + the algorithm can be changed.  
- TF/IDF : Elastic search used this algorithm until recently. : Term Frequency / Inverse Document  
 Frequency  
- Okapi BM25 : used now.

Some of the factors used when calculating scores (**TF/IDF algorithm**).

TF : Term Frequency  
It looks at how many times a given term appears in the field that we are searching for a particular document. The more times the term appears the more relevant the document is.

IDF : Inverse Document Frequency  
This refers to how often a term appears with in the index.  
The more often the term appears, The lower the score and relevance.  
The logic here is if the term appears in many documents, then it has a lower weight.  
This means that the words that appear many times are less significant.

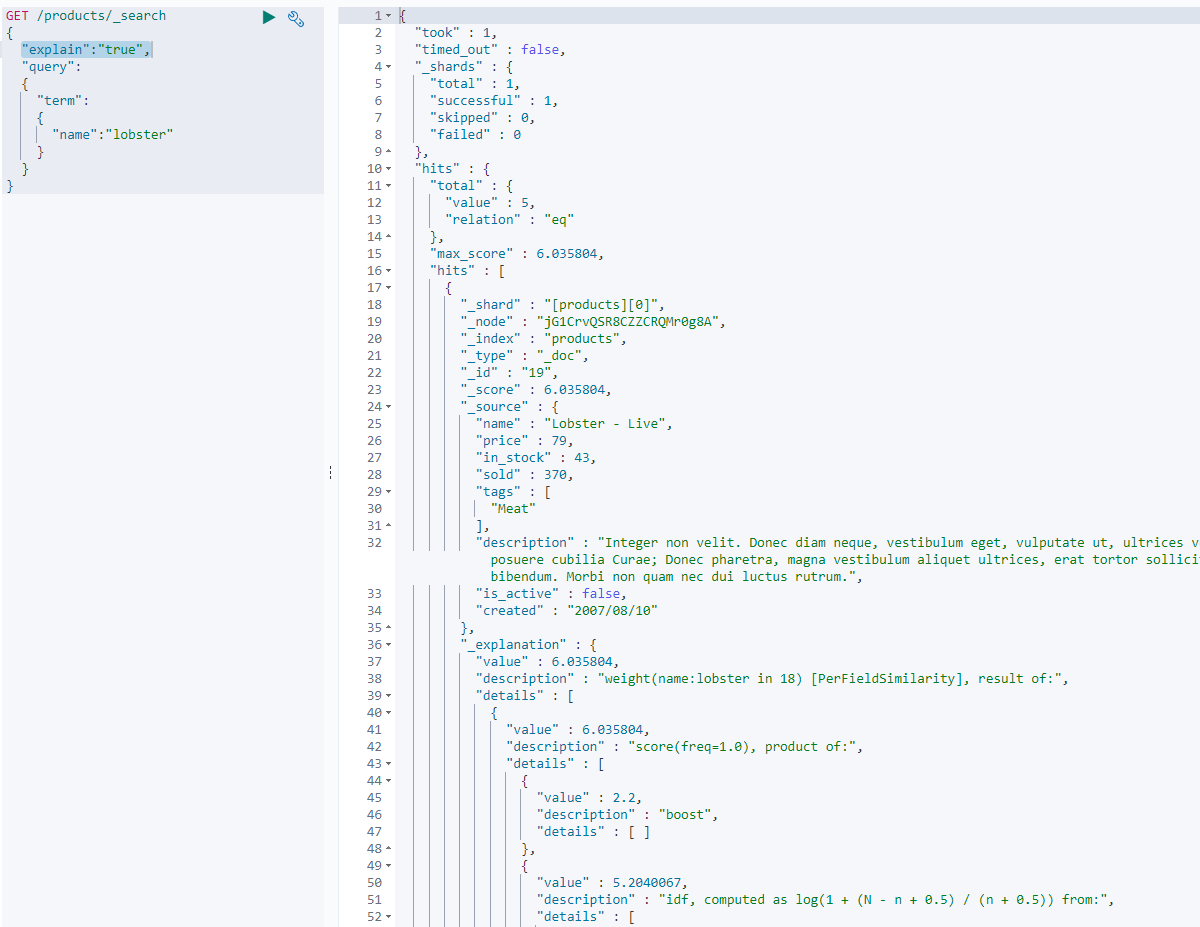
Field-length norm.  
This simply refers to how long the field is.  
The longer the field the less likely the words within the field are to be relevant.   
Ex : The term salad in a 500 character description is more significant that if it appears in a 5000 character description.  
Therefore a term appearing in a short field has more weight than in a long field.

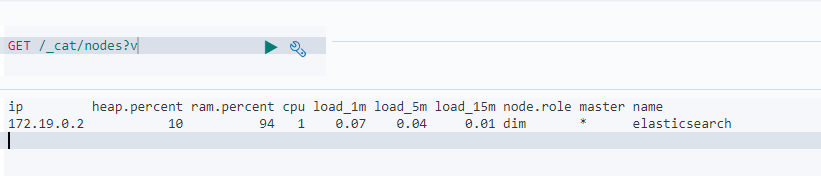
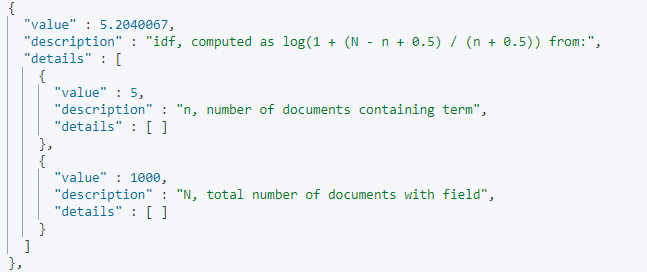
When a document is indexed/updated the TF + IDF + field-length norm are calculated and stored.

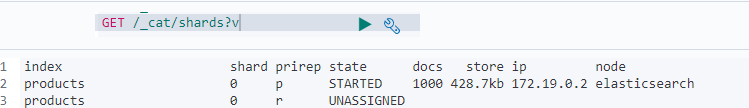
**BM25 algorithm.**stop words : these are the words which appear many times in documents and provide little input as to how relevant the document is in relation to a search query.  
Earlier stop words were removed when analyzing text fields – the reason being that they did not provide any clues in calculating the relevance.[This is the reason the stop token filter being disabled by default for the standard analyzer]

The relevance algorithm needs to handle this, otherwise we would see the weight of stop words being boosted artificially for large fields that contain many of them.

BM25 solves this problem of better handling of stop words: non-linear term frequency saturation.  
If a term occurs 5-10 times in a field the document has more relevance than if it occurs 1-2 times.  
But if a terms appears 30 times or 3000 times the document has the same relevance score. Exponential curve.  
  
Field-length norm (BM25 improves): Instead of treating a field in the same way across all documents, the BM 25 algorithm considers each field separately. It does this by taking the average field length into account. This means that it can distinguish a short title field from a long title field.

60. Elastic Search will return detailed information on how it calculated the score for each matching document.  
  
  
  
  
  

  
  
Notice the value of ‘N’ : 1000.   
Sometimes it could 200 or 300.This is because of the 3 factors mentioned earlier TF / IDF / FLN, which are calculated on a per shard basis..  
This means when determining how many documents contain a given term for instance, then this will based on the documents that are stored with in this shard.

61.

Debugging un-expected search results.  
If you are wondering why a document matched or did not match a query..!  

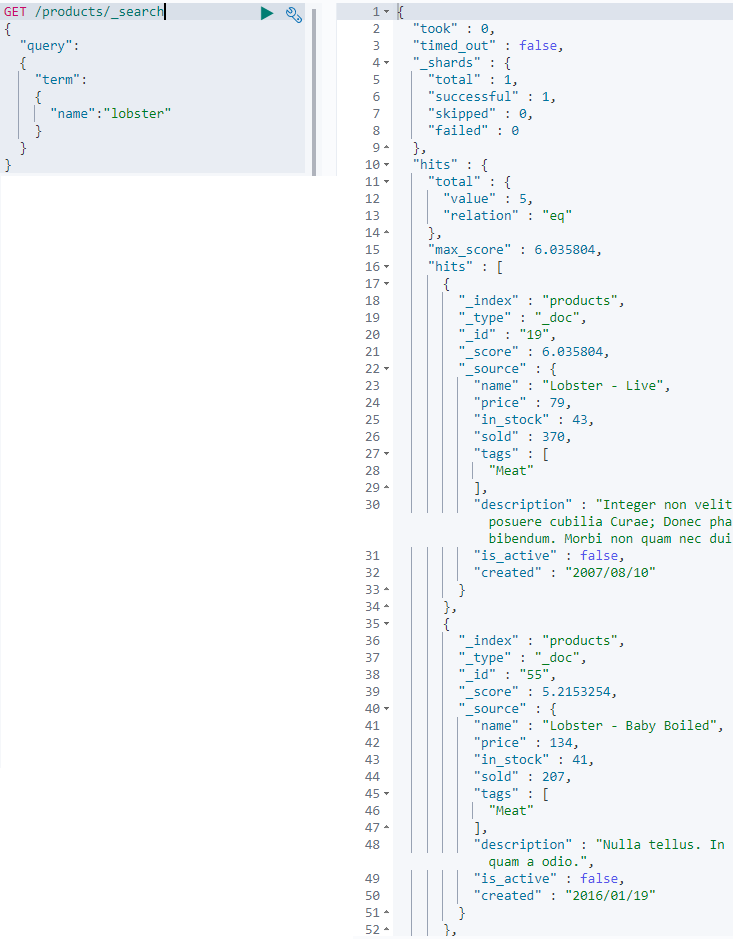

62. QUERY CONTEXT  
  
A query clause can be executed in 2 different contexts.  
Query context: When used with in a query context, we are essentially asking ES the question how well documents match this query. ES will still decide whether or not documents match in the first place, but a relevant score will also be calculated. This is what we have seen so far as we have nested the ‘term’ query clause within the query object.  
Filter context: When adding a query clause with in a filter context, we ask elastic search to document match this query class. i.e documents that do not match the query clause will not be part of the results. This part is the same with the query context..but the important difference is that with filter context, ES does not calculate relevance scores, that is coz it is a Boolean evaluation, either a doc matches or it does not. If it does it will be part of the results else not. The filter context is used for filtering data such as dates,status,ranges..  
So the difference between a Query Context and a Filter context is that when adding a query clause with in the query context the relevance scores are calculated, which is not the case for filter context.

So which context should you add a query context to : it all depends on whether or not you want the query clause to affect the relevant score of matching document – if you want that add the query clause with in the query context i.e with in the query json object.

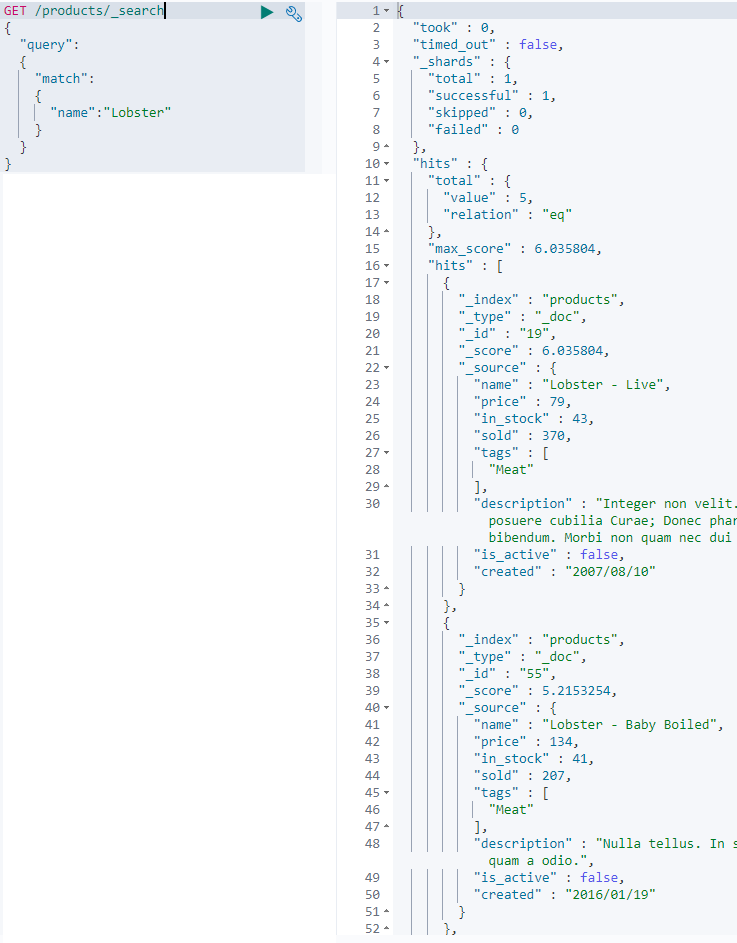
An example : to search documents for the term ‘salad’ with in the title field. If you want documents where the term appears more than once to be MORE RELEVANT + all of the other relevance factors to be taken into account – then go for the query context.  
 If you just want to match any document regardless of how well they match – then you can use the filter context.

63. Types of queries while searching for data.

Full text queries vs term level queries.  
  
We have three search queries which are below. Although they look similar, ‘documents’ will not match for all of them.  
  
Term Level queries : They search for exact matches and are not analyzed   
 – so the casing of letters matters.  
 The term Lobster is capitalized with in the document – but still it matches.  
 The reason why this matches is that we are searching the inverted index itself but  
 not the document itself.  
 So, when we search for the term ‘lobster’ in all lower case letters, this term gets  
 looked up in the inverted index. There we have a match because the documents  
 text field went through the standard analyzer (which has a token filter for  
 lowercasing letters – which is why the term gets stored in all lower case letters  
 in the inverted index.)

  
  
The L in lobster is capitalized in the second query. The same as above is not for the second query below.  
Since the term with in the inverted index is in all lowercase letters, as a result of the analysis process – the query does not match the document.  
  
Below is a (full text) **match** query looking for a capitalized word.

Why does the below query match when we are searching for a capitalized term, and the term with in the inverted index is lower cased. – The reason is unlike term level queries full text queries are analyzed.  
What this means is that the SEARCH query goes through the same analysis process as the documents text fields did – for this particular query this means that the capitalized term lobster is lowercased as part of the standard analyzer and compared to the term with in the inverted index, because the term with-in the inverted index went through exactly the same analysis.  
Thus   
Term level queries – search for exact values and are not analyzed.  
Full text queries – Are analyzed, using the same analyzer (default behavior) that is defined for the field that is being searched.



Term level queries are not useful for performing full text searches – because the inverted index will often differ from the search query – and therefore yield unexpected results.  
Instead Term level queries are better suited for matching enum values, numbers, dates etc. and not sentences – SO WHY WERE THINGS DESIGNED THIS WAY.???

(In the above case) If performing a full text search a user would expect the document to match regardless of the casing.

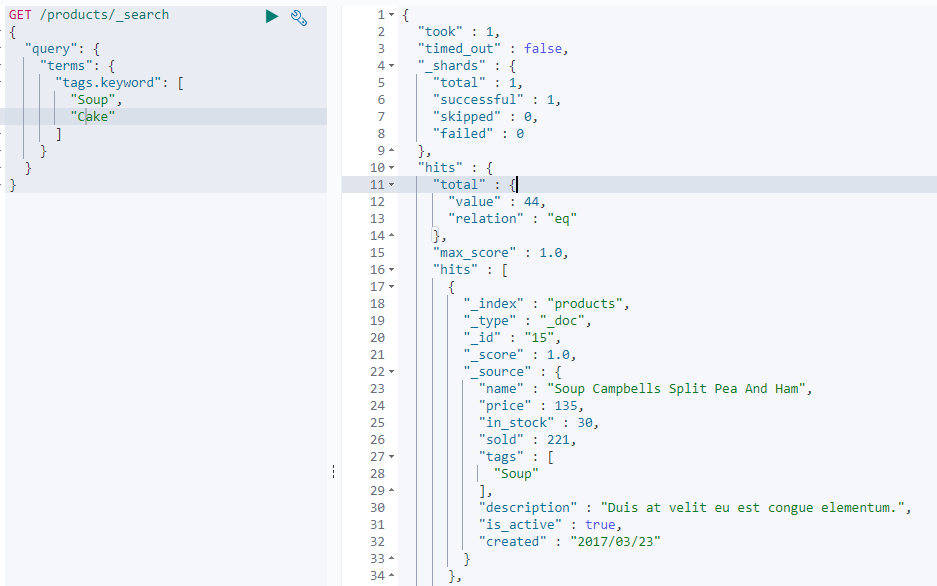
Perhaps you want to apply stemming to account for different variants of the same word such as jump or jumping, which share the semantic meaning.  
Or perhaps we want to make use of synonyms to make our searches more intelligent, so to make both of these to work - the query needs to use the same analyzer that was used during index time for the documents.

Again, this is not the same for term level queries because they match exact values.

64. TERM LEVEL QUERIES. [use this query for searching exact matches – like number field such as dates]

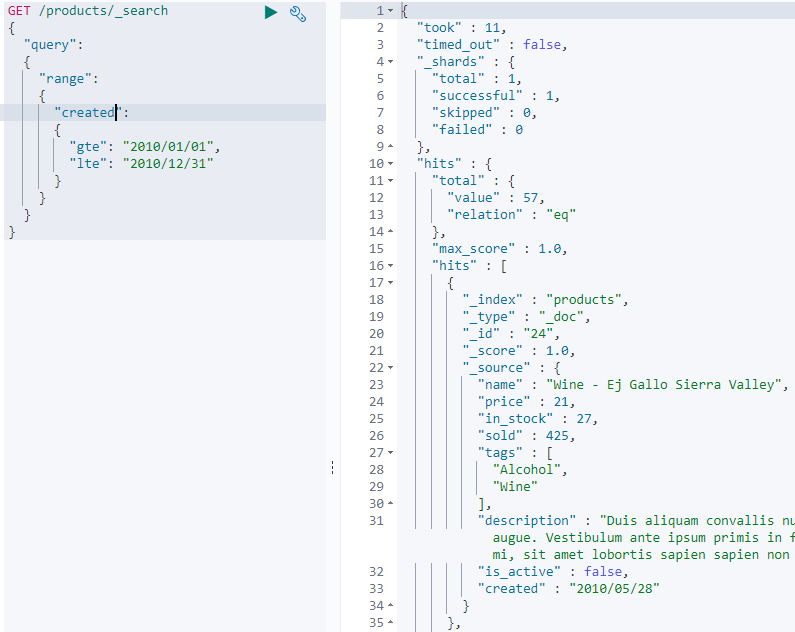
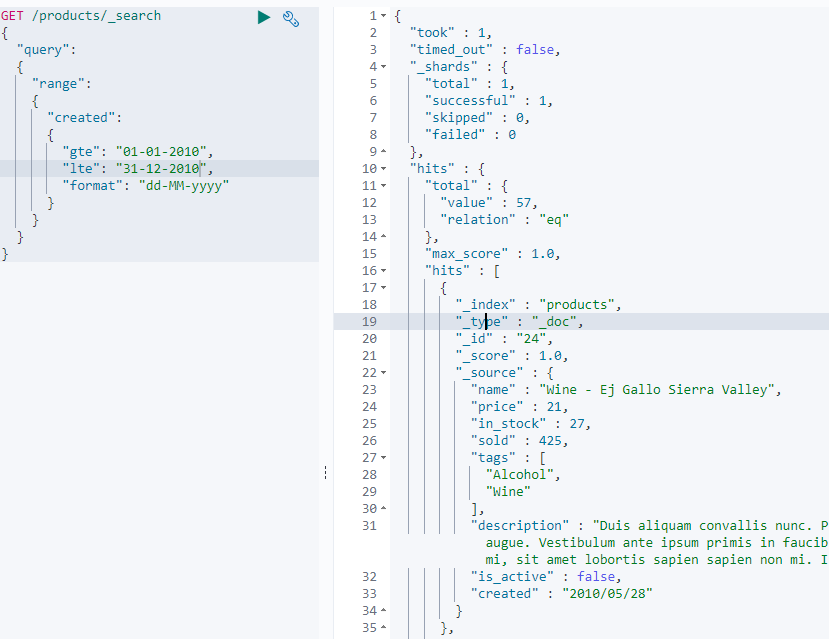


65. SEARCHING FOR MULTIPLE TERMS USING THE TERM QUERY.  
  
The document will match if it contains any of the supplied values, with in the field that we specify …  
*ES maps a text field as both a full text field AND a keyword field automatically.   
ES does this with dynamic mapping. Such fields are multi fields.  
Since we are using a term level query we want to query the keyword mapping as we are looking for exact values.*



66. RETREIVING DOCUMENTS BASED ON ID’S  


67. MATCHING DOCUMENTS WITH **RANGE** VALUES using the range query.  
  
Apart from matching specific values for documents, we can also match documents if a value is within a range. For example we might want to find documents that are almost out of stock – then we should probably order more of them soon.

  
  
Let’s use the range query for date fields - and find out the products added between particular dates.  
  
  
  
The date format we specified is the default one that ES uses, when you add documents to an index.  
It’s also possible to specify an another format for the dates that we define – within the query by adding a format key to the query. This is useful if you need to specify a different format with in the query.  
  


68. Using relative dates (date\_math)  
Anchor date : The anchor point is the point in time we want to use as the basis for the date calculation, i.e the starting point.  
It can be in two formats the keyword now or the date string as shown below.  
If we are using a date string we need to use two pipe symbols to indicate the end of the anchor date.  
After the || double pipe symbol comes the special form of math expression in which we can make use of date and time units – being months, days, minutes etc.  
The math expression is relative to the anchor date we specify.

a date – 1y = 2019/01/01||-1y

a date – 1year and 1 day = 2019/01/01||-1y-1d



We can round of dates as well by appending a forward slash and then a date and a time unit.

A query may supply a date including a time value – in which case you could round the time of mid-night by appending /d to the date expression.  
In general values are rounded down – but in the context of the range query the rounding depends on the parameter (gt,gte,lt,lte) that the date is added for..!

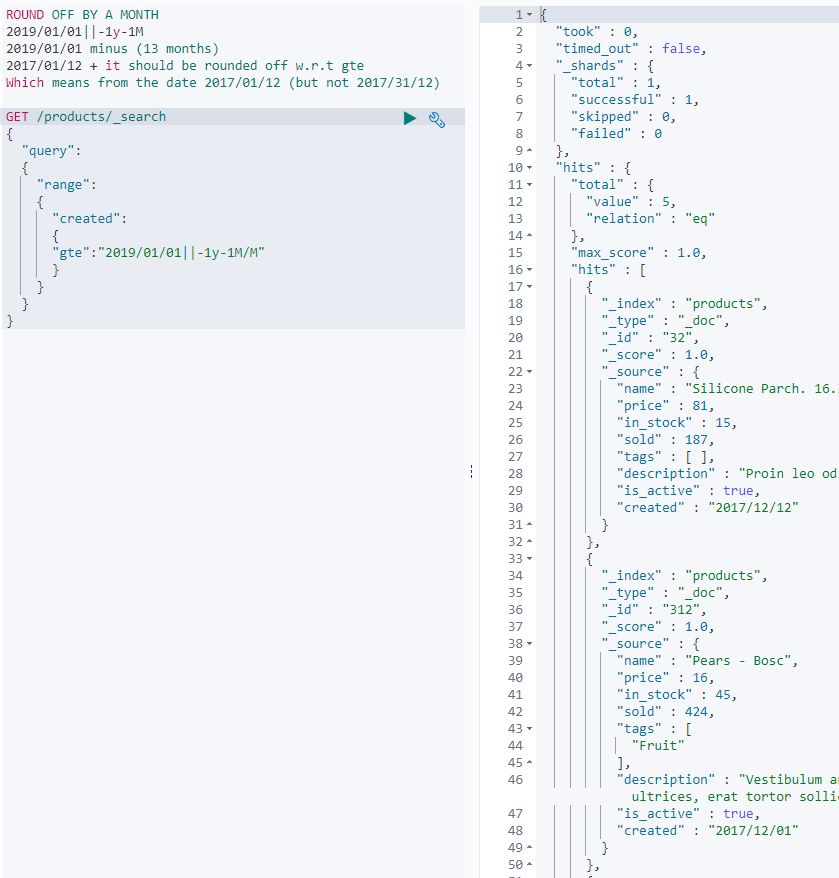
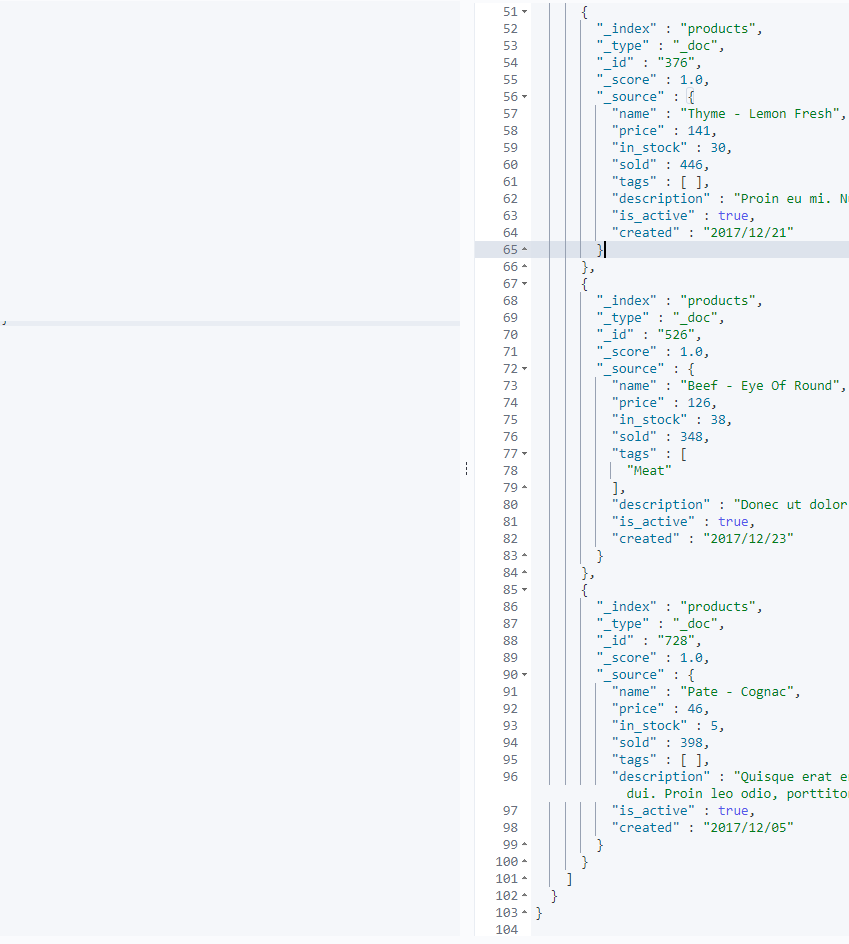
The following table which includes an overview of the range operators along with the rounding direction. + An Example of how the date would be rounded by month

|  |  |  |  |
| --- | --- | --- | --- |
| 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 |

Usually we would round by day – meaning to midnight.  
But this example is rounding off to the nearest month – as it’s easier to show.

Note that even though these examples only specify dates : Elastic search will still round to time values as well internally. This is because ES stores and calculates dates based on the number of milliseconds since the epoch.

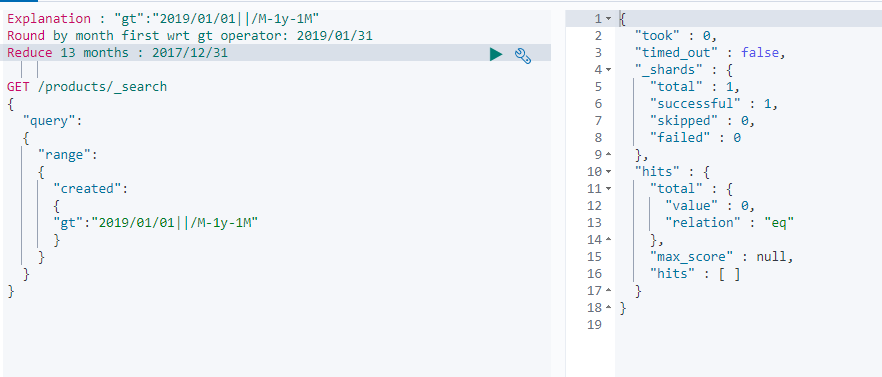
This means that when rounding up, the time would be set to one millisecond before midnight and when rounding down it would be set to midnight all zeros.

Note : The rounding does not necessarily have to be placed at the end of the expression – we can place it at the beginning if we want the rounding to happen before adding or subtracting from the anchor date.

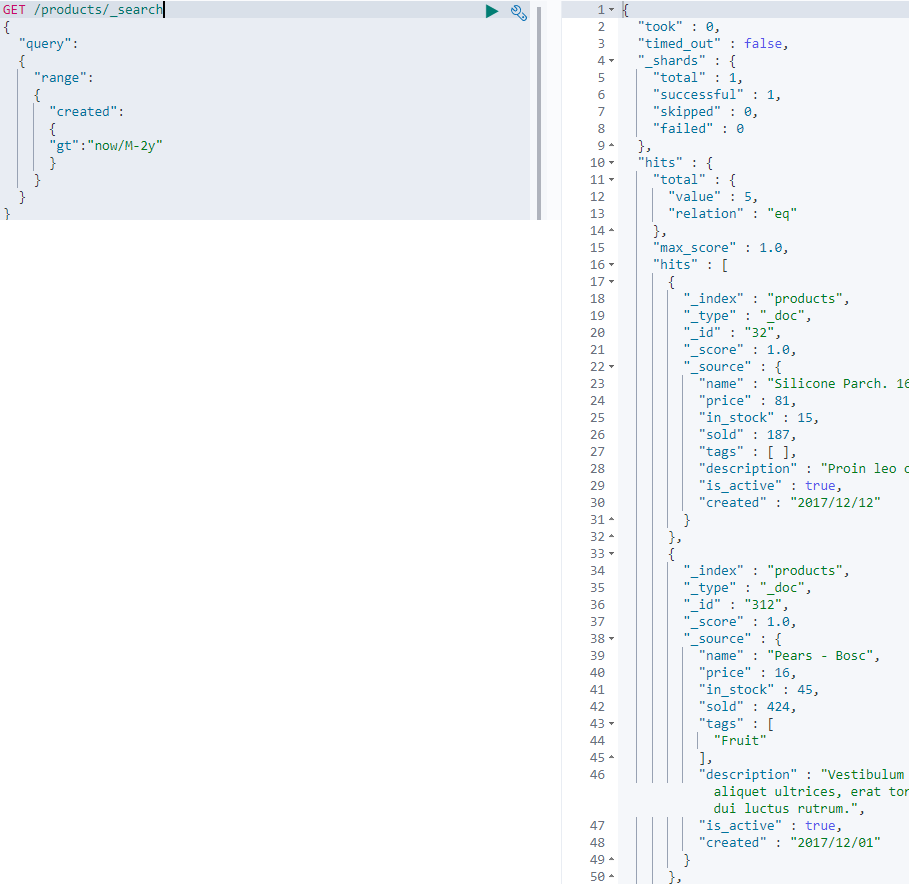
In the below example we are rounding by month before subtracting 13 months.  
Since we used the gte operator the value is rounded down.

If we have used the gt operator the value will be rounded up.



[ NOTICE THE OUTPUT OF THE ABOVE "gte":"2019/01/01||-1y-1M/M"   
 -----there is not document created after December 31 2017, so the immediate above command is correct.]

USING THE KEYWORD NOW – LETS MAKE THE DATE CALCULATION RELATIVE TO THE CURENT TIME STAMP.

Now=11/13/2019  


69. MATCHING DOCUMENTS WITH NON-NULL VALUES

We shall try to match a document which has at least one non-null value – using a query named ‘exists’.

A non-null value is any value that is not null.

In php – an empty string is a null, unless we do a strict check.

This is not the case in Elastic Search: meaning that a field containing an empty string would be matched with the ‘exists’ query.

Then what about an empty array: **That would not match an exists query clause**. Because an array with no value does not satisfy the definition of the exists query. Which is that the fields should have atleast one non-null value.

As the tags field is an array of string we shall use the exists query for matching documents – that have atleast 1 tag.  

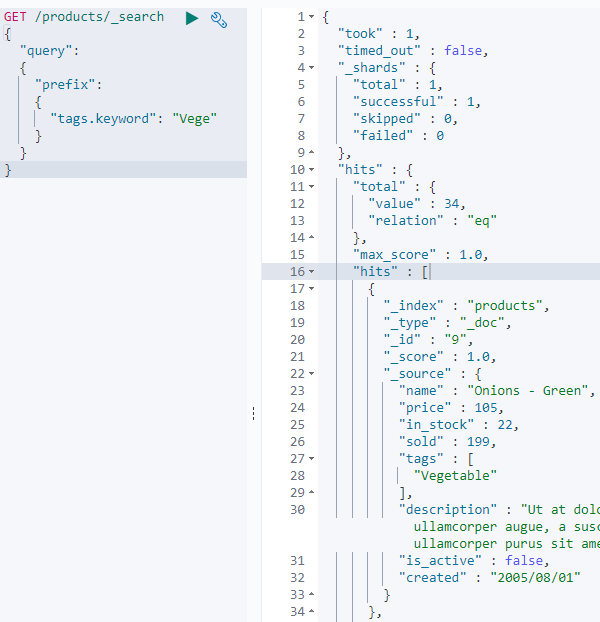

So how does the query work : the meta field **\_fields\_names.**Elastic search uses this meta field internally when finding documents that contain a non-null value for a given field.  
**null\_value** *mapping parameter* : this also has an effect on the *exists* query. i.e if you specify a null value the document will be matched by the query – because the document will no longer have a null value.

70. MATCHINGN BASED ON PREFIXES

Apart from matching documents that contain a given term for a field with the **term query**, You can also look for terms that begin with a given prefix with a query called *‘prefix’*.

i.e the prefix query matches documents that contain a term with in a given field that begins with a given prefix.

The query is not limited to searching fields containing a single term, but will also match a document that has 10 words with in a field and if any of the terms begin with a specified prefix.

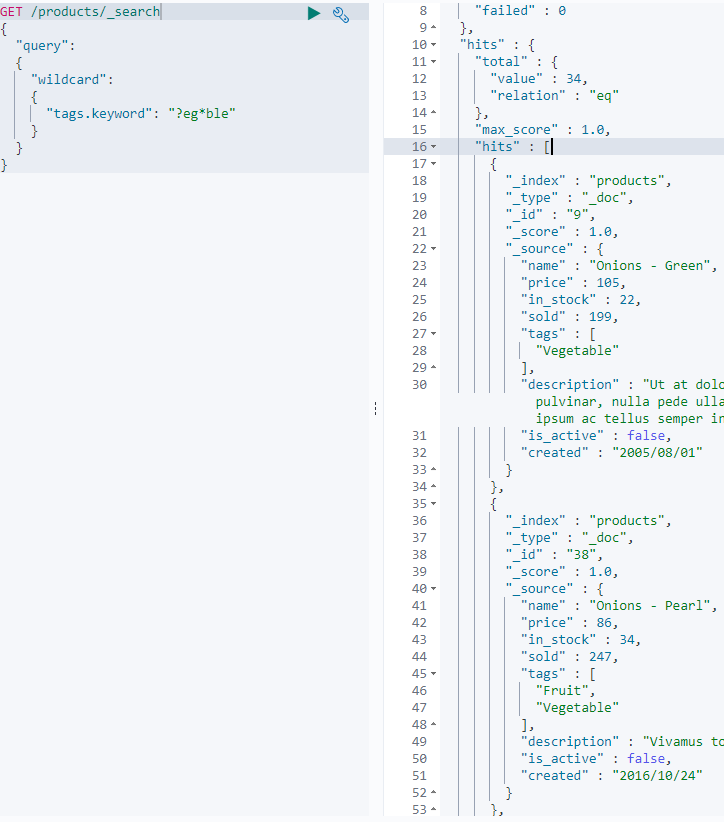


[ We use the *tags* field and more specifically the *keyword* mapping]

71. SEARCHING WITH WILD CARDS.

Elastic search supports a variety of dynamic queries one of which is the wild card query.

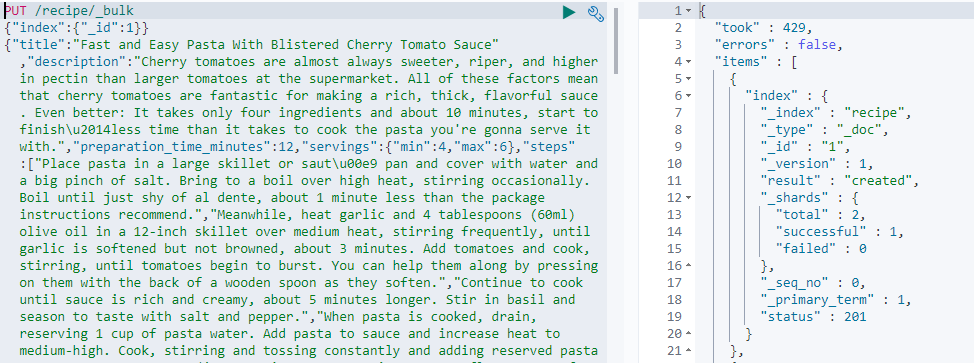
-> ? and a \* can be used. **\* matches any character** and **? matches** **any single** character.

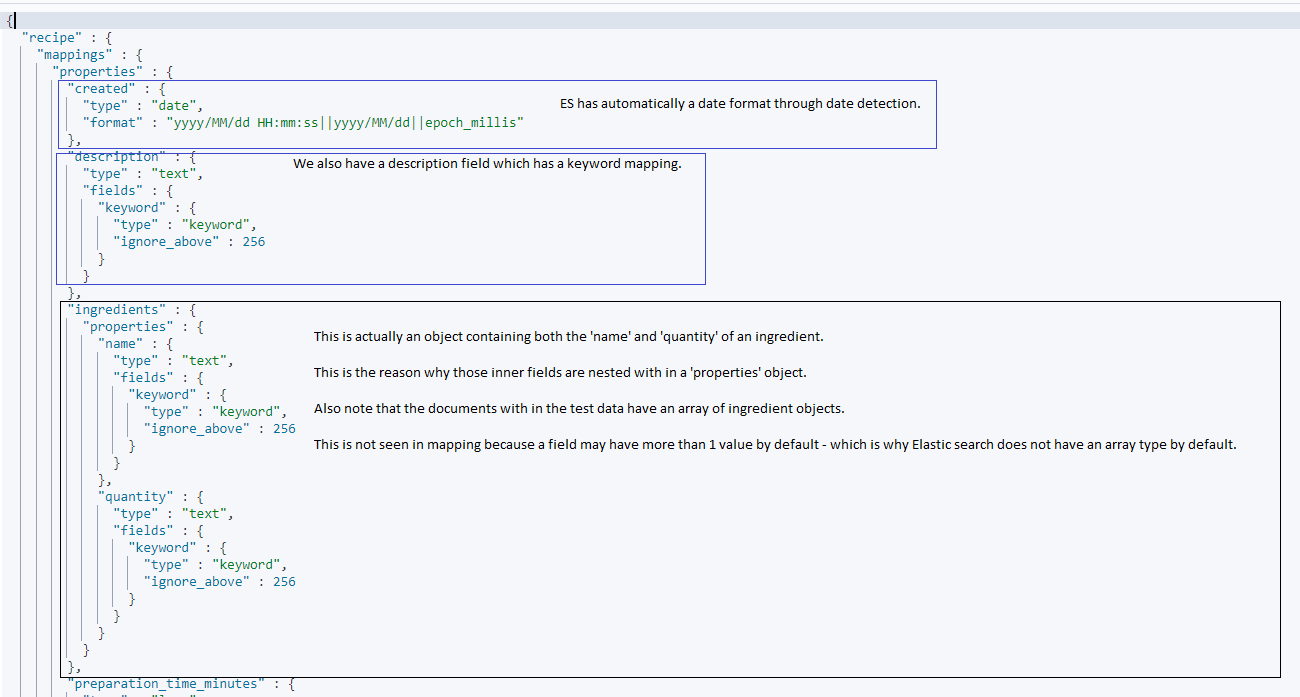


**Placing these two characters in the beginning of the term can lead to extremely slow queries.**

72.searching with regular expressions- can be ignored.

73. INTRODUCTION TO FULL-TEXT QUERIES.

In the earlier section we went through term level queries – looking for exact matches.  
  
  
  
{  
"title":"Fast ...............................Tomato Sauce",  
"description":"Cherry ............................... serve it”,  
"preparation\_time\_minutes":12,  
"servings":{"min":4,"max":6},  
"steps":["...............................passing extra Parmesan at the table."],  
"ingredients":[  
 {"name":"Dry pasta","quantity":"450g"},  
 {"name":"Kosher salt"},  
 {"name":"Cloves garlic","quantity":"4"},  
 {"name":"Extra-virgin olive oil","quantity":"90ml"},  
 {"name":"Cherry tomatoes","quantity":"750g"},  
 {"name":"Fresh basil leaves","quantity":"30g"},  
 {"name":"Freshly ground black pepper"},  
 {"name":"Parmesan heese"}  
 ],  
"created":"2017\/03\/29",  
"ratings":[4.5,5.0,3.0,4.5]  
}

We imported the data with out adding mappings – letting Elastic search take care of the mappings  
  




74. MATCH QUERY

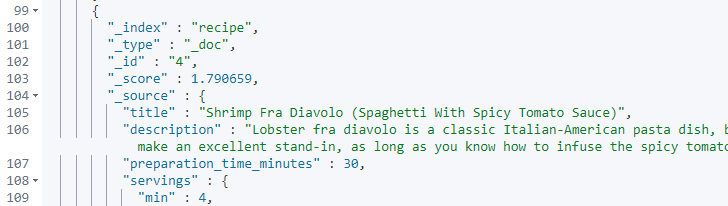
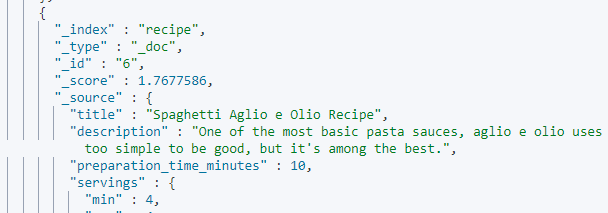
- very powerful and simple query and it allows for very flexible matching.

- very useful for searching queries that users enter in google.

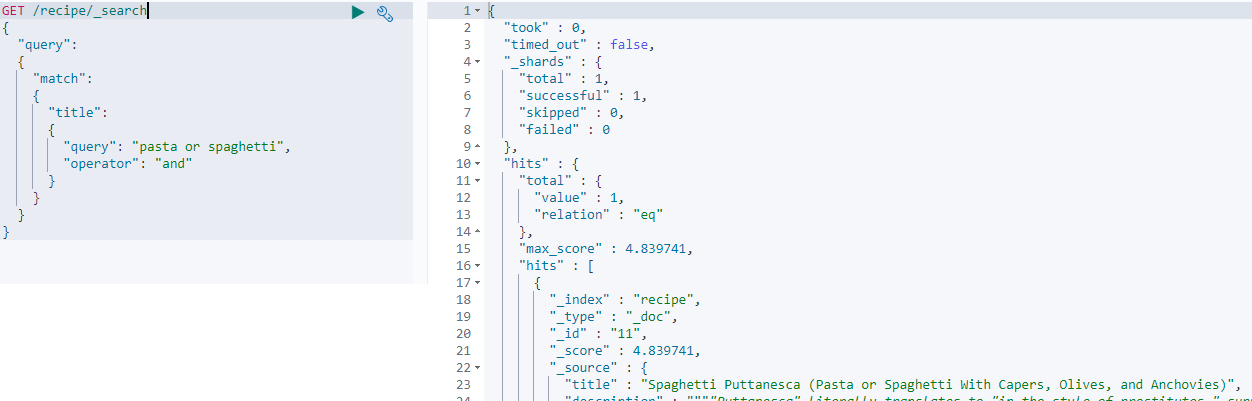
- when dealing with search queries that come straight from a text input the match query is better suited.

**Below query** : The first result contains the term spaghetti twice and pasta once which gives it a high relevance score. As you go to the other results below the title only includes the term spaghetti and none of the other terms.

The reason for that is that the match query is a Boolean query.  
 What happens internally is that the terms are used for Boolean query with the default  
 operator of OR (over simplified here.).  
 Since the default operator is OR all terms need not have to appear in the field they are   
 searching. But the more number of times they appear the higher the relevance. This is   
 useful becoz we do not want to require filler words such as ‘the’ ‘with’ etc to be present,  
 As those words are not significant in terms of relevance.  
 One of the factors of relevance namely the Inverse Document Frequency – this factor reduces   
 the importance and relevance boost for words that appear many times with in an   
 index.Which implies that these filler words will not cause any trouble with regards to   
 relevance.  
 Therefore you would want to leave the Boolean operator at the default value.

**(74.1)** CHANGING THE BOOLEAN OPERATOR in the above query from default OR operator to AND.  


None of the documents match: With the above query, all of the terms with in the query must be present with in a document’s title field. Even a document which was highly relevant and had both the terms pasta,spaghetti in the title field did not match.  
  
LETS GET RID OF A FEW TERMS AND TRY THE QUERY AGAIN.  


This above query matches one document that has the title field that has all the three terms mentioned in the query.

LETS REMOVE THE OR WORD FROM THE QUERY AND TRY.  
  

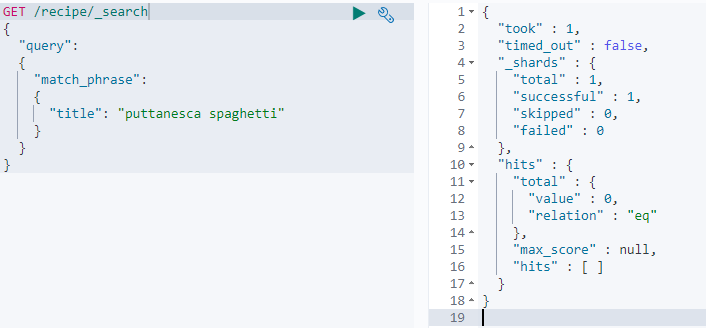

The document still matches as both **terms** pasta , spaghetti appear with the title field.

Note that the query we enter for the match query is analyzed by using the analyzer that is specified with in the field mapping.(This does happen for the term level queries and they look for exact matches +they are not analyzed.)

75. Match Phrases : i.e TERMS IN A SPECIFIC ORDER. Using match\_phrase.

This query matches phrases i.e sequence of terms.

With the match\_phrase query the order of terms matches.  
  


So both *spaghetti* and *puttanesca* must appear in the *title* field in the order as mentioned in the query, and with no other terms in between for the query to match. And so the below query does not show any documents.  


76. SEARCHING MULTIPLE FIELDS – multi\_match  
  


The query will search both the *title* and the *description* field.

You could use the *type* option here but let us see what happens by default.

Suppose we are searching for two fields for the terms *pasta* and *spaghetti* – a document will be more relevant if both the terms appear in the same field instead of 1 term in each field.

By default documents containing the terms in any of the fields are matched.  
But the relevance score from the best matching field is used.  
This means if one of the field contains both terms and the other field only contains one of them – then the relevance score for the first field is used for that **document.**

77. INTRODUCTION TO COMPOUND QUERIES.

We shall use Boolean login when constructing search queries - by using compound queries.

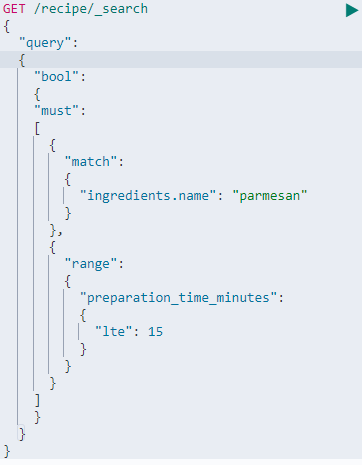
Querying with Boolean login – it is similar to the where clause but it also deals with relevance scores.

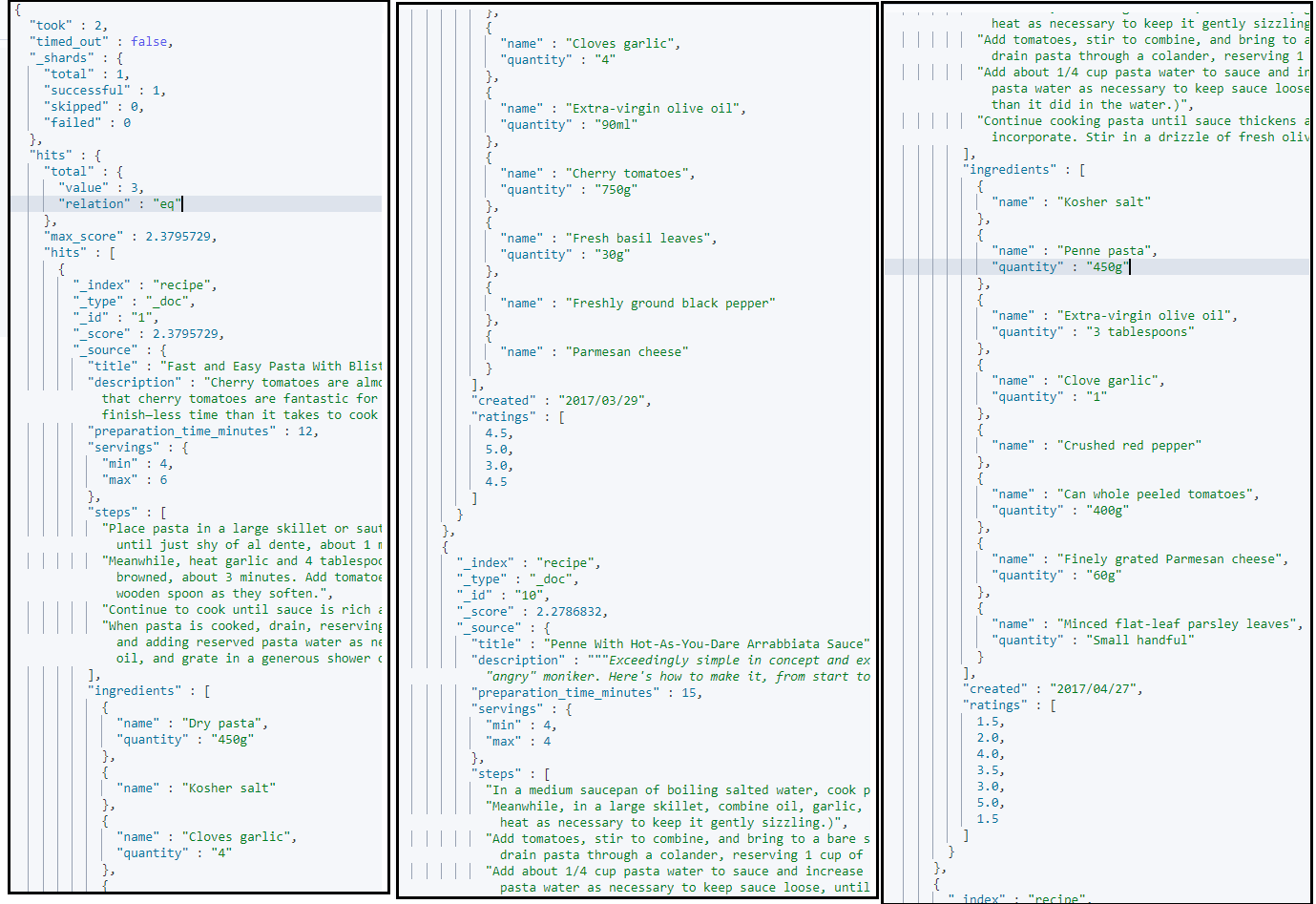
Recap - The context in which a query can be run **query** context or a **filter** context.

With in a query context – relevance scores are calculated and documents are ordered by how well they match the given query.

In a filter context – It is only determined whether or not a document matches a query AND NOT how well it matches a query.

The bool query can be used with both the contexts – but we shall see the usage in a query context.

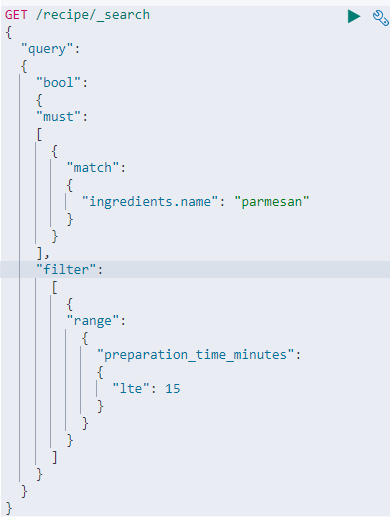
Example 1: We want to match recipes that contain ‘*parmesan*’ as an ingredient *And* we want the preparation time to be 15 minutes or less – we can achieve that by using a *key* names *must* [a array of *query clause* objects].  
  


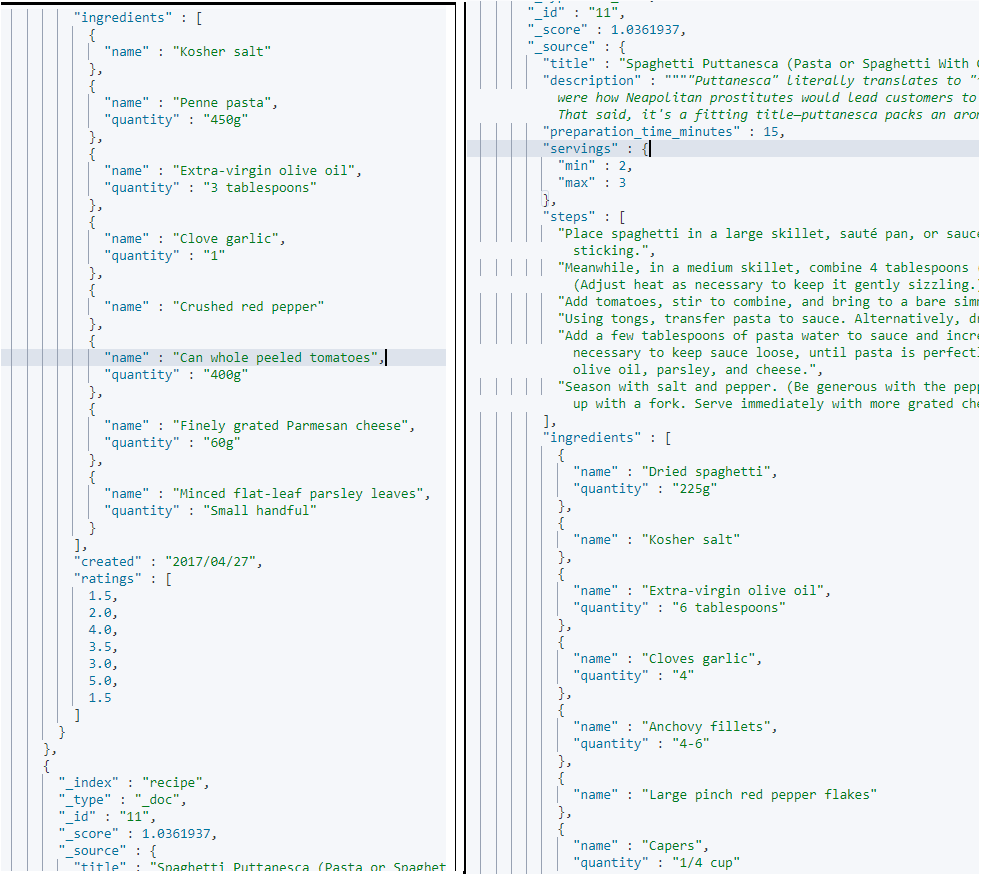
What does this actually mean: the queries nested with in the must key must be satisfied for a document to match the query. We must also note that the queries with in the must key contribute to the relevance score. This means that while a document must be matched by all queries with in the must key / must array - it is also scored by how well it matches the queries[this happens based on various relevance scores discussed earlier.].  
  


Results : We can see that the recipes do indeed contain parmesan and have a preparation time of less than or equal to 15 minutes.

CAN WE IMPROVE OUR QUERY ? : As it makes no sense to calculate how well a document satisfied the range query. It really a yes or no question. So instead of keeping the query with in the must array – we should move it with in the filter object. [while elastic search is smart enough to apply a constant score of 1 to the range query while is it placed with in the must array , the filter object has an edge in terms of performance. As said the filter object is for queries that either match or don’t match – i.e there is no such thing as how well they match. More precisely queries with in this object are run in a filter context which is why scoring is ignored.]

Caching is where filter queries have an edge because the query clauses can be cached for sub sequent queries. This is because the result of the query clauses directly depend on the query and the documents – there is no notion of how well the documents match. This makes the results much easier to cache.

LETS MOVE THE RANGE QUERY TO THE FILTER OBJECT.  


We will get the exact matches as before but with scores 1 less.  
  
…………………  


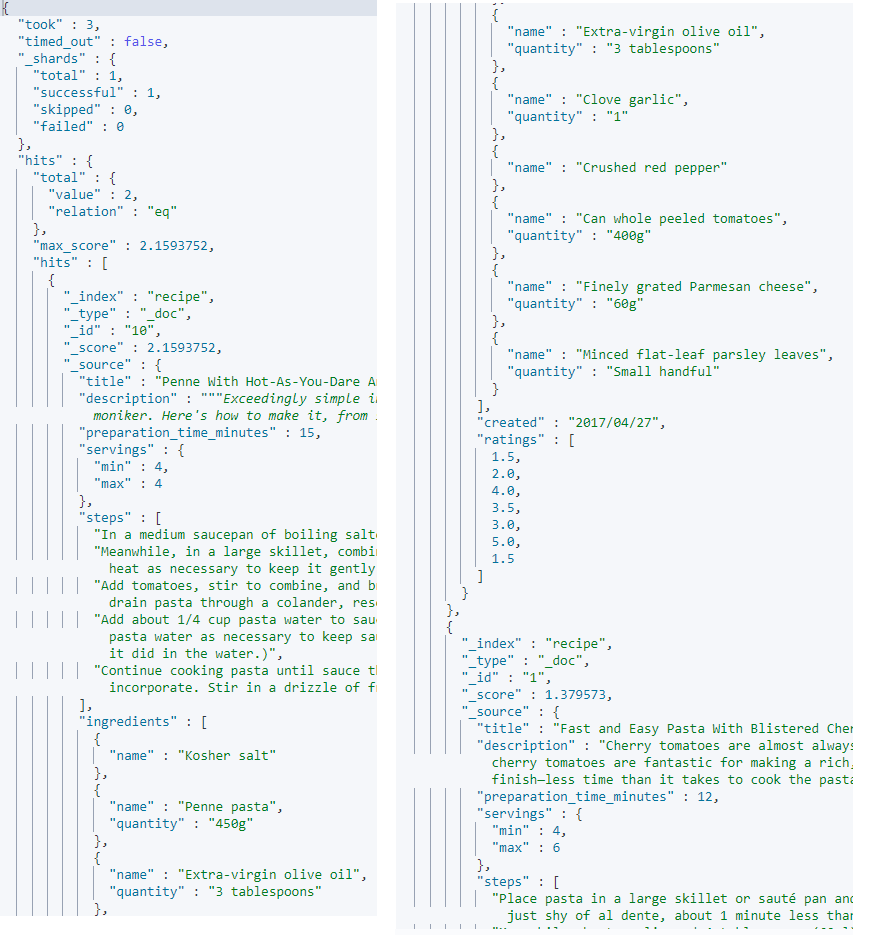
LET US SAY IF WE WANT TO REFINE THE QUERY : by removing recipes that contain tune….!  


The query clauses within the must\_not object are executed in a filter context – exactly as with the filter object. Scoring is therefore ignored and a score of zero is returned for all documents.

As with the other queries with in the filter context, queries with in the must\_not object are also considered for caching, which ES handles for us.

LETS CHANGE THE QUERY A LITTLE BIT MORE: I WANT TO RANK RECIPES THAT CONTAIN PARSLEY HIGHER AS I REALLY LIKE PARSLEY.  


The specialty of ‘should’ is that : the queries with in it boost a score if they match, but they are NOT required to match. So a recipe that contains parsley is likely to score higher than a recipe with out parsley.

This is no **guarantee**, as it depends on how well other queries match too….So you can think of an object where you can define preferences. That is we prefer recipes that contain parsley but you won’t require them to contain parsley.  
  
   
  
The reason that the ‘should’ object is a bit special is that its behavior depends on boll query as a whole and what else it contains.

If the bool query is in a query context and contains a must or a filter object : then the should queries do not need to match for a document to match the bool query as a whole.

If this is the case - the only purpose of the should query is to influence the relevance scores of matching documents.

On the other hand – if the bool query is in a filter context OR if it does not have a must and filter object then atleast one of the should queries must match – that is because should queries are intended for boosting documents matching a set queries, but the documents should still satisfy some constraints., else every single document in an index will be matched.[you can control this with the min\_should\_allowed parameter- which determines how many should clauses must match via numbers or percentages.]

OK LETS COME BACK TO THE BEGINNING AMD TRY ANOTHER QUERY.

Match documents that contain pasta as an ingredient and optionally parmesan so a must query and a should query respectively.  
  
s

The above query matches recipes which contain a ‘pasta’ ingredient and it gives a relevance boost to recipes with ‘parmesan’.  
  

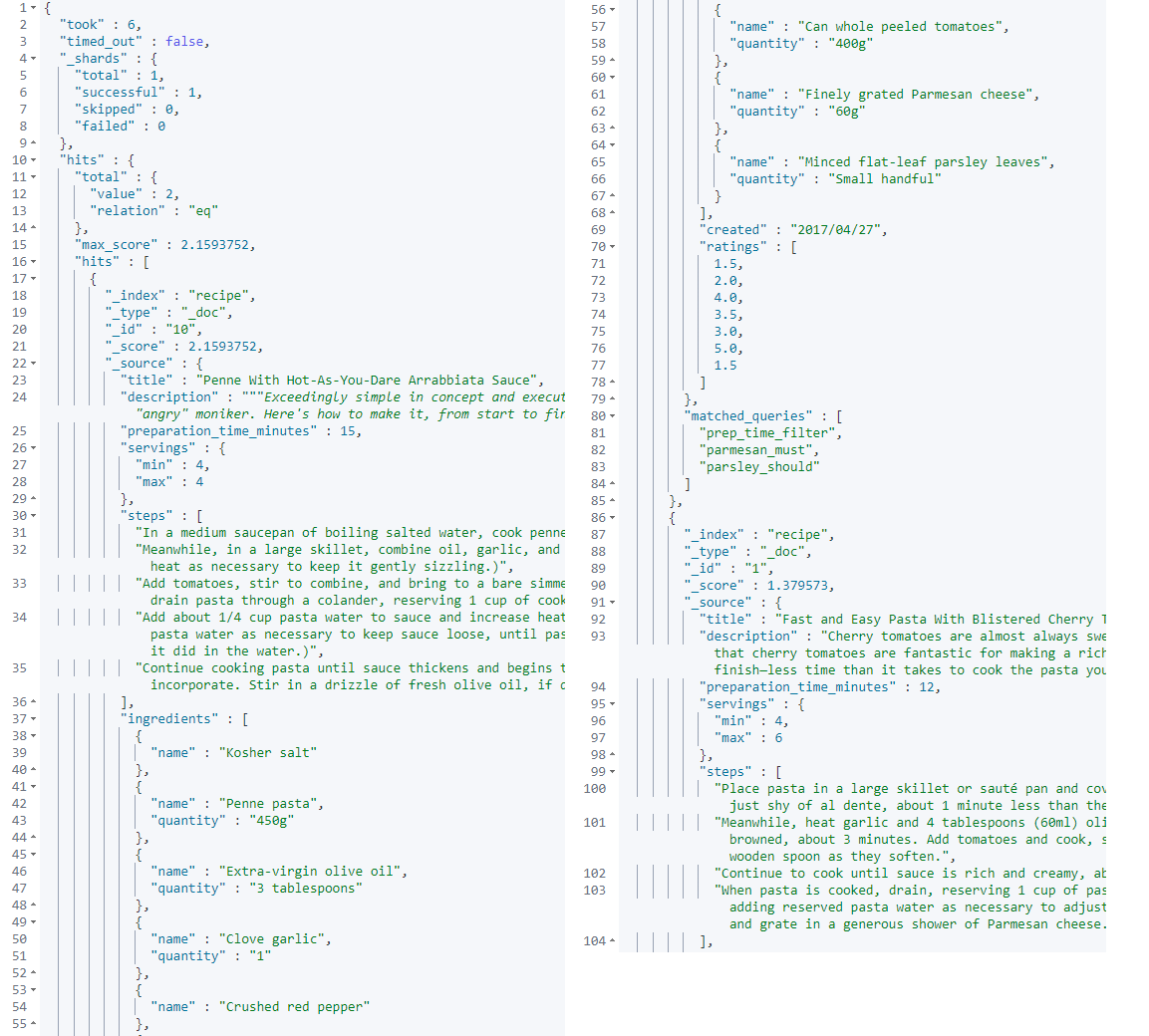

LETS ALTER THE QUERY AGAIN : BY REMOVING THE must OBJECT.  


We get the same number of matches : Since we don’t have a *must* or a *filter* object : atleast one should query must match. And in this case we have only 1 query clauss i.e the recipes MUST contain parmesan.

Removing the must object means that the parmesan ingredient went from **being** optional and boosting the relevance score to **being** required.

78. DEBUGGING BOOL QUERIES WITH NAMED QUERIES.

\_explain api : useful in understanding why a document did or did not match a query.  
\_named queries: Something similar to above can be done for bool queries. This helps us to see which query clauses a document matched. We don’t obviously need this for the ‘must’, ‘must\_not’ and ‘filter’ queries – because these behave the same across all documents.   
 It becomes useful when we want to see which ‘should’ clauses a document matches and try and make sense of the relevance scores.  
  


RESULT:   
  
  


matched\_queries : this contains an array with the query names that the document matched.

The first match in the above list : matched both the *filter and must* query clauses.

And it also matched the should query.

The second document matched only the mandatory query clauses and not the *should* query which gives a relevance boost to recipes containing parsley.

This is how you can use named queries to debug bool queries to get some insights about which parts of a bool query a given document matched.

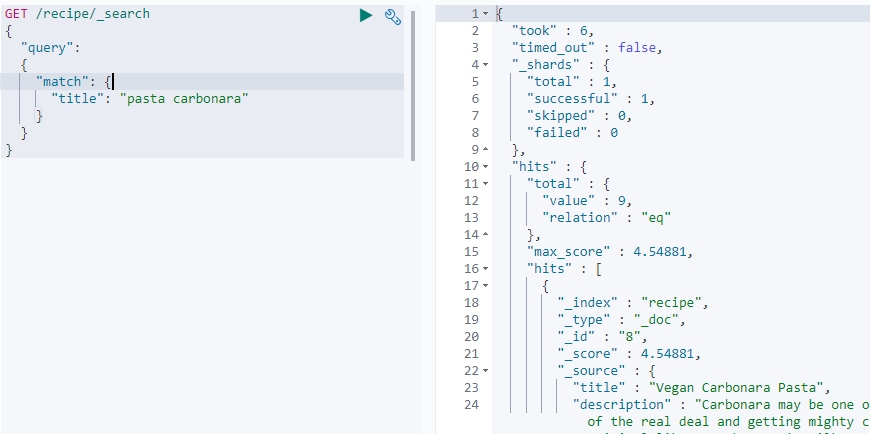
79. HOW THE MATCH QUERY WORKS.

Note(74.1) : the *match*  query uses a default Boolean operator of OR which can be changed to AND.



* Even though the match query is not categorized as a compound query, the match query actually constructs a bool query internally.
* It does this as part of the analysis process after the query has been analyzed by the fields analyzer.
* The tokens resulting from the analysis process are added to a bool query as term queries.
* i.e the match query is a convenient wrapper around the bool query that simplifies writing common queries.
* This eventually means that – you can through a search query directly at Elastic Search instead of having to tokenize the query first in your application to construct a bool query.

Lets confirm that with an example.



Note that the default operator for the match query is or – so this above query translates into a *bool* query with **two** *query* clauses one for *each term with in the match query*.



The above two queries do exactly the same thing. The relevance scores match.

NOW LET US USE THE ‘MATCH’ QUERY AND CHANGE THE BOOLEAN OPERATOR TO ‘AND’.



The equivalent of this query : It is a bool query with each term translated into a term query with in a must object.  
  


Just as with the bool query the match query also accept the ‘*minimum\_should* *\_match*‘ where you can configure how many of the should clauses should match.

This parameter is passed to the bool query during the analysis process.

But how does all of this work – a match query is analyzed and a term query isn’t – the term query is constructed based on the results of the analysis process. So the match query goes through the appropriate analysis process which ‘tokenizes the query / lowercases the letters / or whatever the analyzer is configured to do ‘

So, the terms that are used for the term queries, have therefore been analyzed and therefore match what is stored with in the inverted index.

NOW LET US TEST this by capitalizing the letter P in Pasta in the *match* query.



WE still get the same results because the query is analyzed and part of that process is to lower case the query.

NOW LET US capitalize the letter P in Pasta in the *bool* query.



Now we no longer get any matches.

This means that even if we supply the *match* query with capitalized words – it’s the result of the analysis process that is used for constructing the bool query.

In this example the analysis process two terms – pasta and carbonara.

Which are then lowercased by a token filter.

The resulting terms are then used for the *term* queries with in the bool query.

So we are essentially searching analyzed data with an analyzed query.

And this is the reason we get the same matches.

RECAP:

* When you send a match query to ES, it analyzes the query and adds a term query clause to a bool query for each term that comes out of the analysis process.
* If the Boolean operator is OR which is the default – then the terms are added a *term* query clauses with in the *should* object – meaning that atleast one of them should match.
* If the Boolean operator is set to AND – then the *term* queries are added with in the must object- meaning that all of them should be present for the given fields for the documents to match
* Since the query is analyzed it’s the resulting terms of the analysis process – that are used for the term query clauses.
* This means that the query has gone through the same analyzer as what is stored with in the inverted index.
* But what if match query only receives a query consisting of a single term, in that it does not make much sense to use a bool query – so instead a single term query is used. i.e a term query that is not nested with in a bool query.
* THIS IS HOW THE MATCH QUERY WORKS INTERNALLY

80. INTRODUCTION: Joining Queries.

* We shall look at querying relationships between documents.

Nested Query : The way of joining data, can be done using this query.

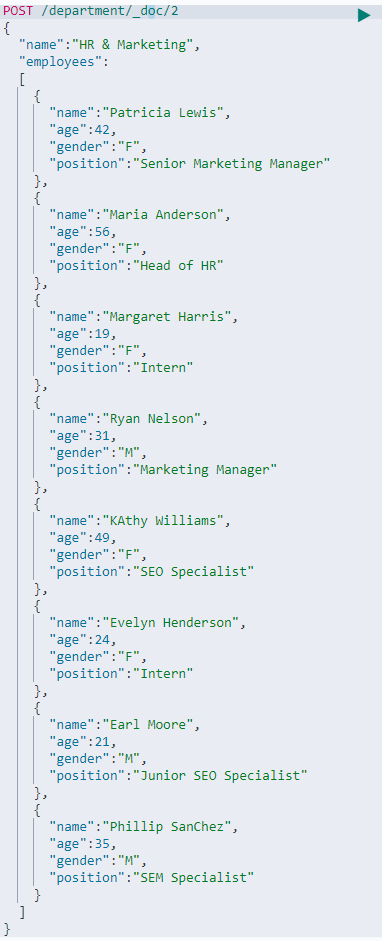
Nested data Type : This is used together with the nested query.   
 This type is used for arrays of objects where you want to maintain the   
 relationships between object properties. i.e many to 1 relationships.

If a field is just mapped as an array of objects, An object is not   
 independent – because all of the object properties are mixed together when   
 stored by ES.



The *Employees* field type is of type nested as it will contain an array of objects.

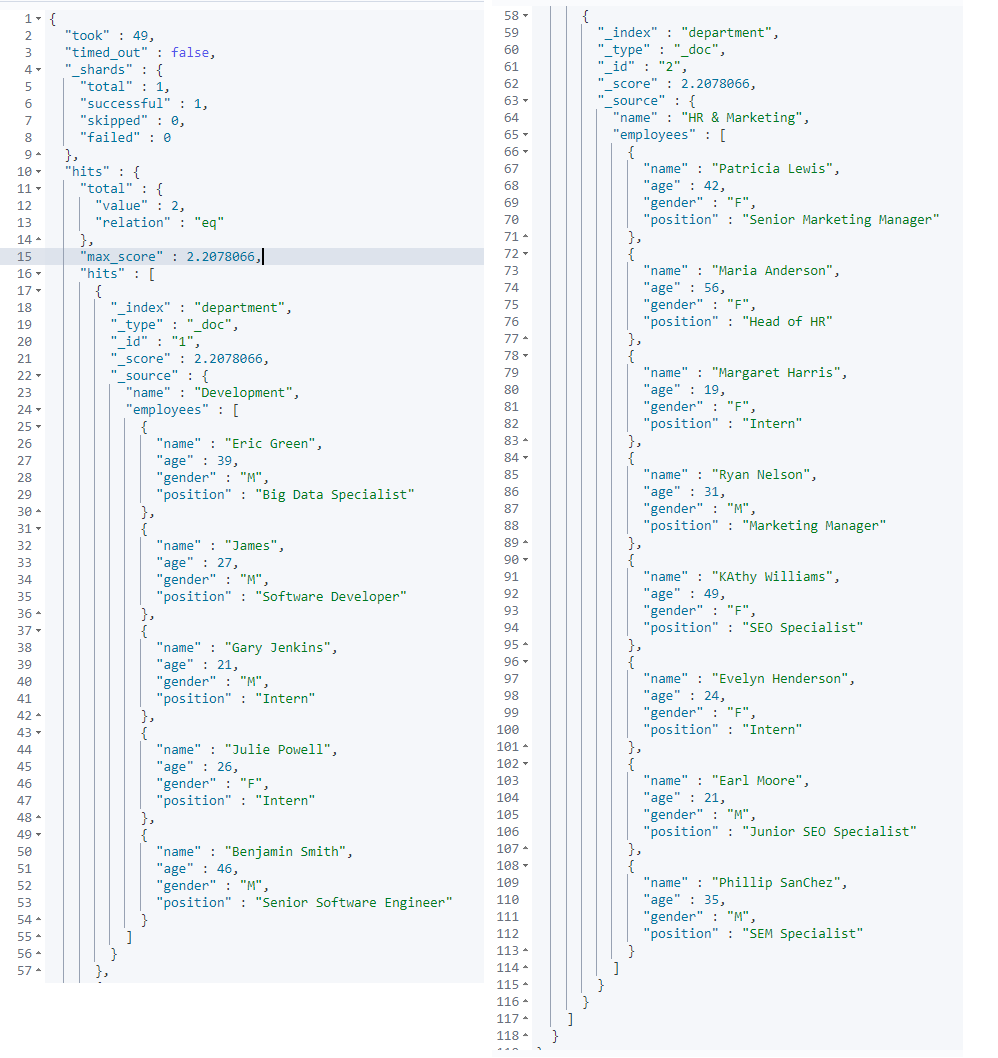
Add a first document with a *development* department and its employees.  


Add a second document with a *Hr and Marketing* department and its employees.  


Let us query the nested filed i.e the employees field : for female interns.So we have to write 2 queries and wrap them with in a bool query.



Running the above query will yield no results as nested fields cannot be queried that way – and they must be queried using a specialized query type called *nested.  
*

Result:  


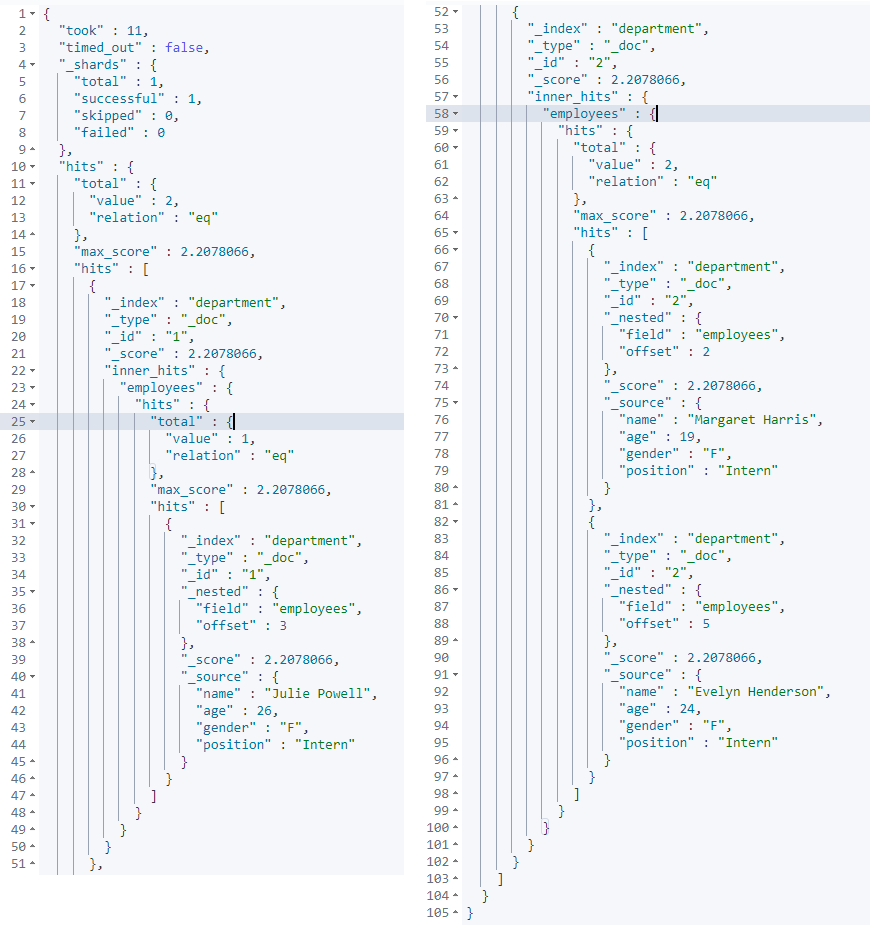
* Both departments are now matched. ***Which departments/documents have employees that are both ‘F’ and ‘Intern’***
* This nested query is used in combination with the nested field data type, and enables you to query each object independently.
* More specifically you can query each object as if it were an independent document – which is actually how objects are stored independently.
* A problem with this approach is that each employee is stored with in that department document.
* This increases the complexity of keeping the employees updated.
* How would it be if an employee was a document itself and that we had a way of establishing a relationship between the *department* document and the *employee* document? [Kind of the same thing you would do in a relational database with the foreign key], we can actually do this with a ‘joint’ field.

81. Inner Hits.

- we did not find out which employees caused the document to match from the above results.

- we do this with a feature called ‘inner hits’ and it can be used with the ‘joint’ data type.

- How to ask ES to include the nested inner hits.  
  
  
RESULTS:



82.