Elastic Search

ElasticSearch is a real time distributed and open source full-text search and analytics engine developed in Java under Apache License version 2.0. It is an Apache Lucene-based search server developed by Shay Banon. It is accessible from the RESTful web service interface and uses schema-less JSON documents to store data.

## General

1. ElasticSearch is scalable up to petabytes of structured and unstructured data.
2. ElasticSearch uses denormalization to improve search performance.

## Key Concepts

| Node | It refers to a single running instance of ElasticSearch. Single physical and virtual server accommodate multiple nodes depending upon physical resource like RAM, processing power etc |
| --- | --- |
| Cluster | It is a collection of one or more nodes. Cluster provides collective indexing and search capabilities across all nodes for entire data. |
| Index | It is a collection of different types of documents and their properties. Index also uses concepts of shards to improve performance. Example: A set of documents contains data of social networking problems. |
| Document | It is a collection of fields in a specific manner defined in JSON format. Every document belongs to a type which resides inside an index. |
| Type | It refers to a collection of specific Documents. It is also known as ‘Table’ in RDBMS. |
| Field | It refers to a particular key of a JSON. It is also known as ‘Column’ in RDBMS. |
| Shards | Indexes are horizontally subdivided into shards. Each shard contains all the properties of documents but contains less number of JSON objects than index. Horizontal separation makes shard an independent node, which can be stored in any node. Primary shard is the original horizontal part of an index and then these primary shards are replicated into replica shards. |
| Replicas | ElasticSearch allows users to create replicas of their indexes and shards. Replicas improves data availability in case of failure and increases search performance by carrying out parallel search operations. |

## Advantages

1. ElasticSearch is developed in Java which makes it compatible on almost every platform.
2. ElasticSearch is real time, in other words after one second the added document is searchable in this engine.
3. ElasticSearch is distributed, which makes it suitable for scale and integration in any big organisation.
4. Creating full backups are easier by using the concept of gateway which is available in elastic search.
5. ElasticSearch uses JSON objects as responses which makes it possible to invoke ElasticSearch with a large number of programming languages.
6. Almost all types of document are supported except those which doesn’t support text rendering.

## Disadvantages

1. ElasticSearch has problems with split brain situations.(Nodes shutdown in one machine may affect other sometimes)
2. ElasticSearch does not have multi language support in terms of handling request and response data (only possible in JSON).

**Architecture of Elastic search**

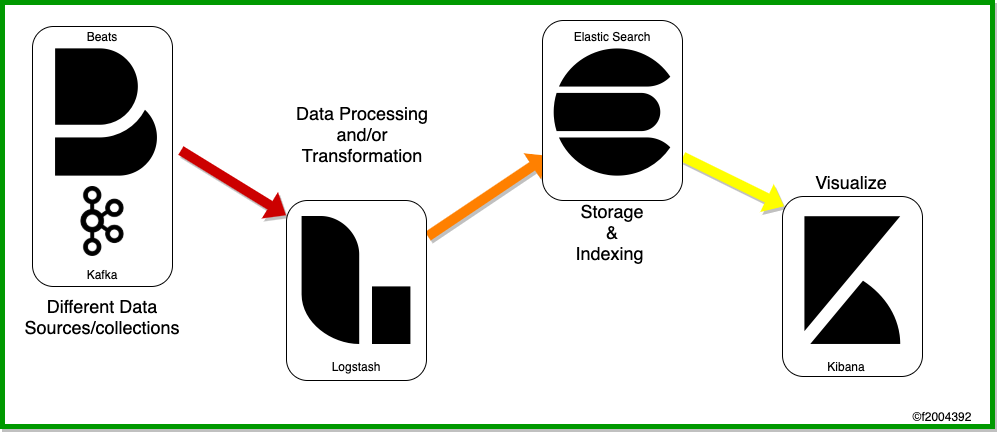
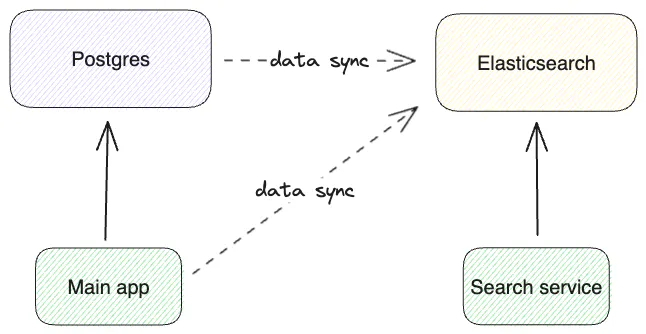
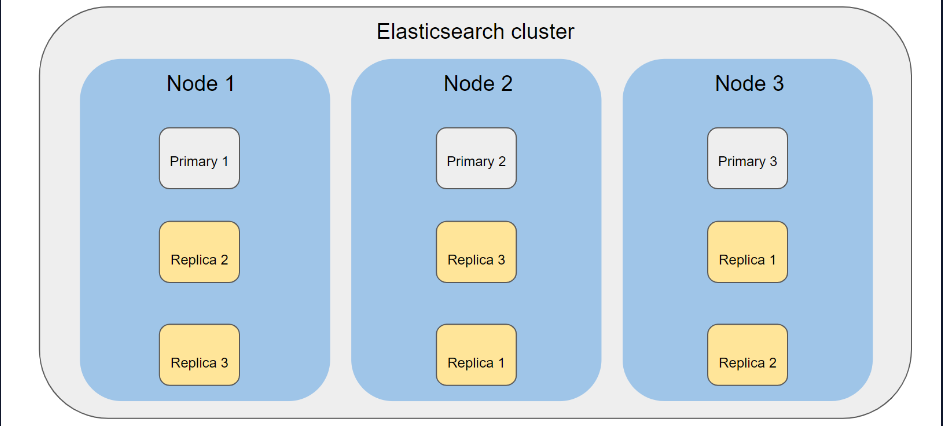


Fig:- **ELK stack working**

**Elastic search for search functionality**

****

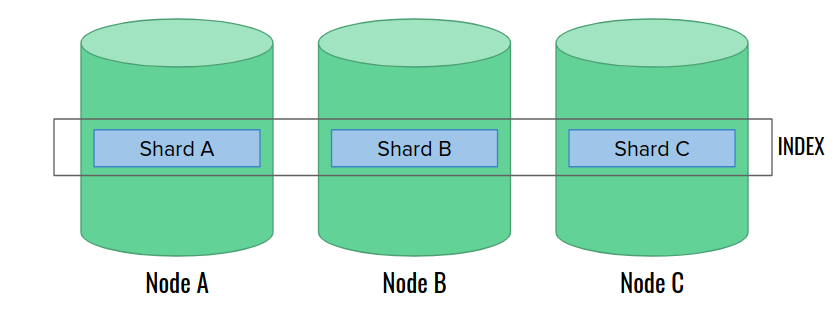
**CLUSTER, NODE**

## 

## CLUSTER :- Collection of nodes that are similar

## Node :- Single node that can be in different machines, helps in data safety and data loss,and balances request(Load Balancer)

**SHARDING**

****

* Divide of single index(Product) to multiple places data integrity, data safety , backup and scalability
* Product(1000 GB) , Node(500GB) , in this case product can be divided into multiple shards and can be stored.

**Replication Groups**

****

* Store a replica of shard that is in another node to node. Even if the machine that contain Node(A) crashes, its replica is stored in node(B), so data will never be loss
* Replica will not be stored in same node

**Roles of Nodes** (only Map roles for large DataSets)

1. **Master** Node :- create, delete,indices ,If multiple nodes present, voting will happens to choose master node*(node.master=true)*
2. **Data** Node :- Node to store data, perform search query, modification of data
3. **Ingest** Node :- Process data before storing data(Like logstash,for simple use case)
4. **Coordination** :- Load balancer for large cluster ,distributed queries
5. **Machine Learning** :- only for running machine learning jobs
6. **Voting** Only :- participate in voting master node

**Create**:

PUT /products/\_doc/1

{

"name": "Product X",

"price": 100.00,

"description": "A great product"

}

**Read**:

GET /products/\_doc/1

**Update**:

POST /products/\_doc/1/\_update

{

"doc": {

"price": 120.00

}

}

**Delete**:

DELETE /products/\_doc/1

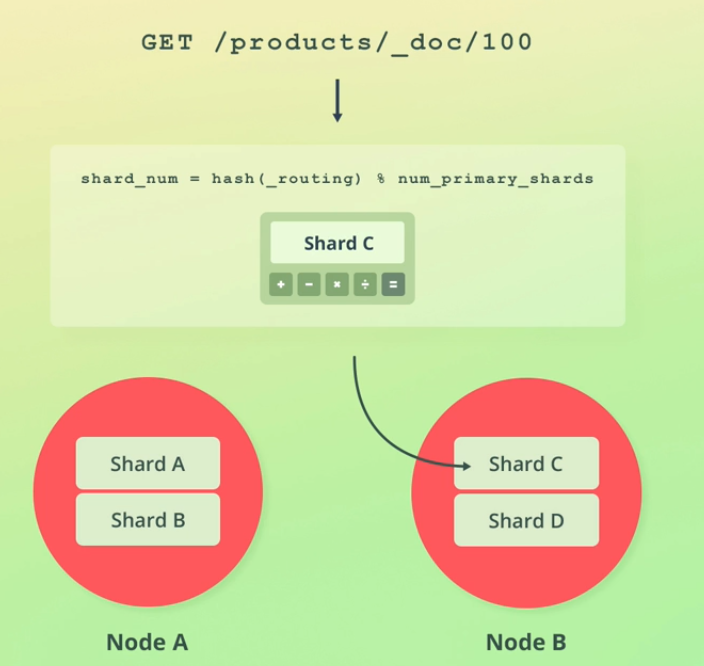
### **Bulk Operations:**

Elasticsearch also supports bulk operations to perform multiple CRUD actions in a single request. This is more efficient when dealing with several documentsOST /\_bulk

{ "index": { "\_inde

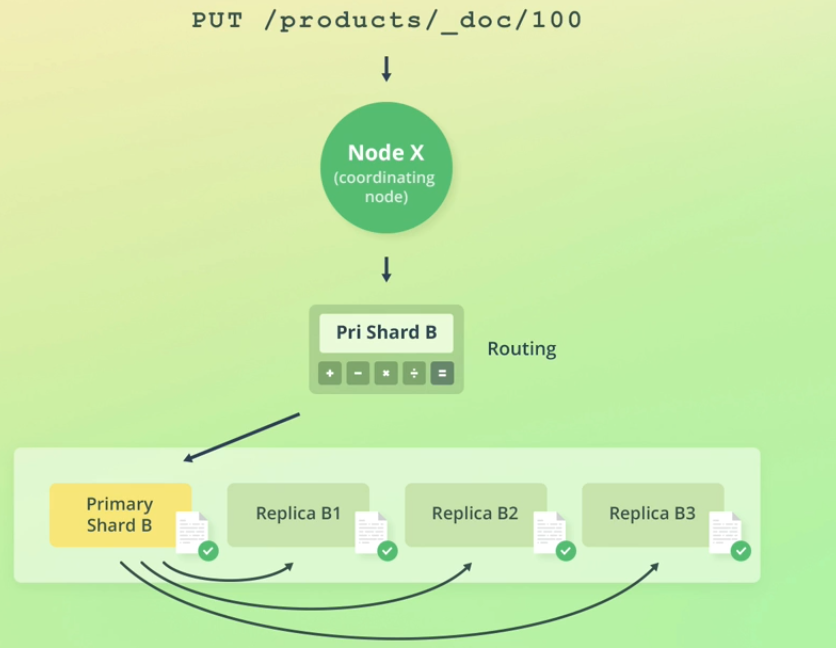
| POST /\_bulk  { "index": { "\_index": "products", "\_id": 200 } }  { "name": "Espresso Machine", "price": 199, "in\_stock": 5 }  { "create": { "\_index": "products", "\_id": 201 } }  { "name": "Milk Frother", "price": 149, "in\_stock": 14 } |
| --- |

**Routing** (How ES finds documents once they have been indexed(in different shards in different nodes which are in different machines?)

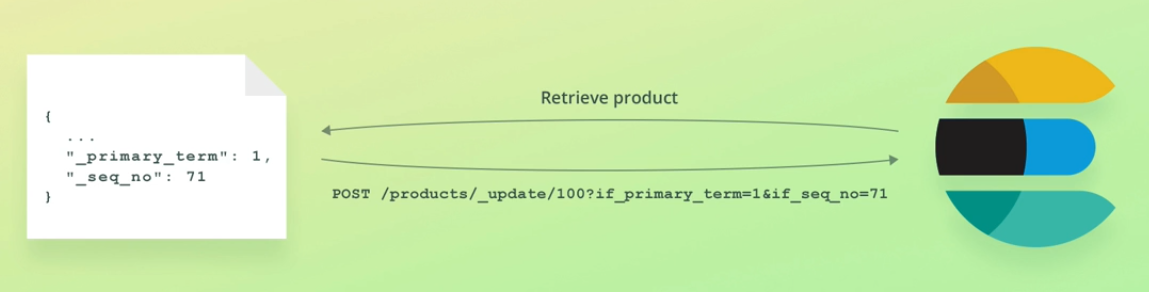


hash(\_routing) = \_id is hashed

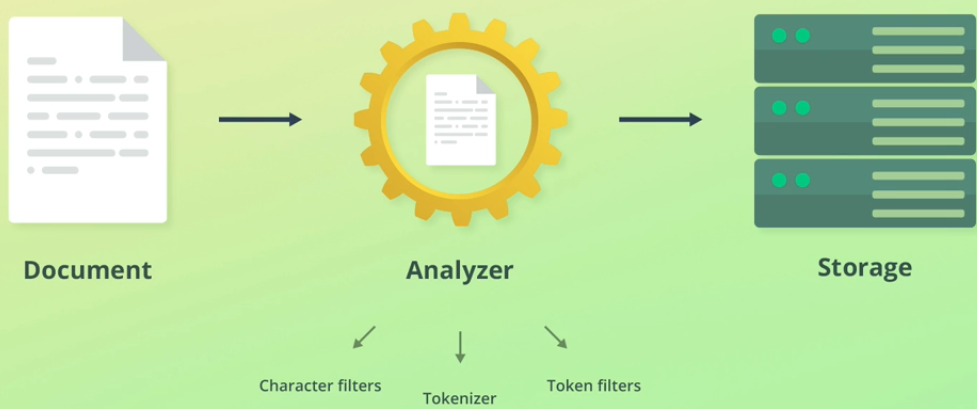
**How does ElasticSearch write data?**

****

**How to handle concurrency issues in elastic search?**

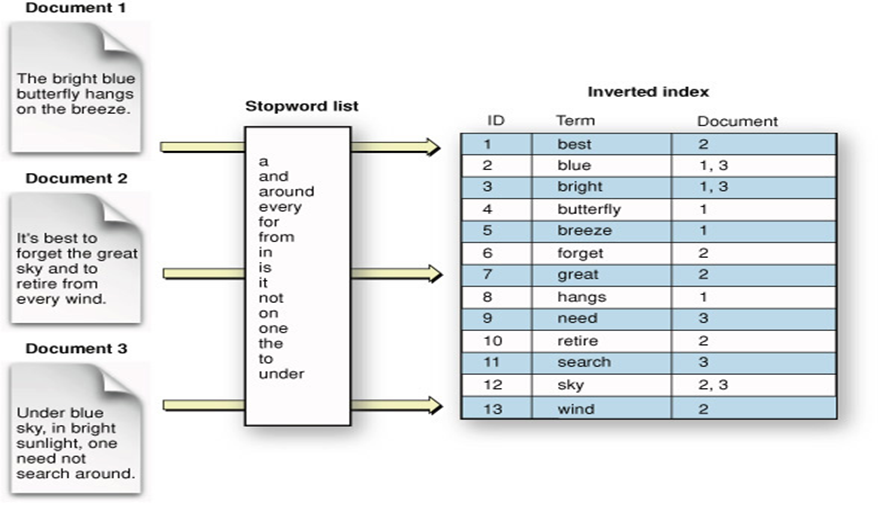
****

1. Retrieve \_primary\_term and Sequence\_term
2. Send them with post /update request
3. This way if anyone changes that data(***\_primary\_terms and \_seq\_no will change***) in between then it will not update data as primary\_terms and seq\_no will change if any operation happens in it.

**Analysis (**Result stored in different Data structure for efficiency)  


****

**INVERTED INDEX (**How text fields are stored and can be searched efficiently?**)**

****

* Each document runs through Analyzers and tokenizers and data are stored according to that
* So while searching a certain text that will also go through those analyzers and search which document matches those text(String)
* If 10000 documents are already stored in Index and then you configure/Add a new tokenizer then it will create issues as the present document was saved using the old tokenizers.

**Data Types in ElasticSearch**

**(**Short,Integer,short,text,float,date,double,long, nested,parent-child**)**

**Object data type(Inner Object): -**

****

**For this type(ARRAYS) :-** AND == OR , becoz,data will same propertyName will store in each array, so how to resolve this issue (Nested Data Type // Parent-child)

**Nested Data type**

| **Mapping** | **Data** |
| --- | --- |
| **{**  **"Person":{**  **"properties":{**  **"name" : {**  **"type" : "string"**  **},**  **"car":{**  **"type" : "nested"**  **}**  **}** | **{**  **"name" : "Zach",**  **"car" : [**  **{**  **"make" : "Saturn",**  **"model" : "SL"**  **},**  **{**  **"make" : "Subaru",**  **"model" : "Impreza"**  **}**  **]}** |

**Note:- Internally,** Each car is a separate document, and if there are 10 cars then 11 documents will be stored(10 cars + 1 root document)

**How Dates work in ElasticSearch?**

All the date formats are converted to epoch time under the hood while saving in elastic search i.e

"2015-01-01" , "2015-01-01T12:10:30Z" , 1420070400001

Will be stored as 14007040001(epoch time)  
 **Mapping in ElasticSearch**

**Inner Object**

* Easy, fast, performant
* Only applicable when **one-to-one** relationships are maintained
* No need for special queries

| ﻿// categories/my-category  {  "title": "my category",  "required\_member\_level": 123,  "date\_created": "2016-11-16T00:00:00Z",  "Products.name": ["Ergonomic Keyboard 11", "USB Camera" ],  "products.price": [ 100.0, 34.99]  } |
| --- |

**Nested**

* Nested docs are stored in the same Lucene block as each other, which helps read/query performance. Reading a nested doc is faster than the equivalent parent/child.
* Updating a single field in a nested document (parent or nested children) forces ES to reindex the entire nested document. This can be very expensive for large nested docs
* "Cross referencing" nested documents is impossible
* **Best** suited for **data** that **does not change frequently**

| **MAPPING** | **NESTED DATA** |
| --- | --- |
| {  "person":{  "properties":{  "name" : {  "type" : "string"  },  "car":{  "type" : "nested"  }  }  }  } | {  "name" : "Zach",  "car" : [  {  "make" : "Saturn",  "model" : "SL"  },  {  "make" : "Subaru",  "model" : "Impreza"  }  ]  } |

**Parent/Child**

* Children are stored separately from the parent, but are routed to the same shard. So parent/children are slightly less performance on read/query than nested
* Parent/child mappings have a bit extra memory overhead, since ES maintains a "join" list in memory
* **Updating a child doc** does **not affect** the **parent** or any other children, which can potentially **save a lot of indexing** on large docs
* Sorting/scoring can be difficult with Parent/Child since the Has Child/Has Parent operations can be opaque at times

**Example** :- **Person** has **multiple** **Homes** in **different States**

| {  "mappings":{  "person":{  "name":{  "type":"string"  }  }  }  } | {  "homes":{  "\_parent":{  "type" : "person"  },  "state" : {  "type" : "string"  }  }  } |
| --- | --- |

**Denormalization**

* You get to manage all the relations yourself!
* Most flexible, most administrative overhead
* May be more or less performant depending on your setup

**Question** :- Let's say you want to change the mapping of an already existing index (change text to long) how to do that?  
**Ans** :- Not directly possible because of the already stored data are saved in are indexed in different data structure so  
**RE-INDEX -** It will create new index and copy the data into that index

POST \_reindex

{

**"source"**: {

**"index"**: **“index\_product"**

},

**"dest"**: {

**"index"**: **"index\_new\_product"**

}

}

**Multi-value Map**

| Text can be mapped with both text and keyword(exact match) at same time(it will make two inverted Indexes) one for text, one for keyword(performance less) If exact match needed(EMAIL) use keyword field. | "city": {  "type": "text",  "fields": {  "raw": {  "type":"keyword"  }  } |
| --- | --- |

**Stemming and stop words**

Words that will not be indexed (a,an the,that) are stop words. They will be discarded while indexing the document text fields.(Analysers do that for you)

**SEARCHING**

**Term -Level Queries**

*Case sensitive*, *Not Analysed*, and used for *exact match* of the *term*.

| Basic Term Search | Case-insensitive Term search | Multiple Term search |
| --- | --- | --- |
| GET /products/\_search  {  "query": {  "term": {  "tags.keyword": "Vegetable"  }  }  } | GET /products/\_search  {  "query": {  "term": {  "tags.keyword": {  "value":Vegetable",  "case\_insensitive":true  }  }  }  } | GET /products/\_search  {  "query": {  "terms": {  "tags.keyword": ["Soup", "Meat"]  }  }  } |

**Range Searches**

| Basic Range search | Date searches | Specify UTC offset |
| --- | --- | --- |
| GET /products/\_search  {  "query": {  "range": {  "created": {  "gte": "2020/01/01",  "lte": "2020/01/31"  }  }  }  } | GET /products/\_search  {  "query": {  "range": {  "created": {  "gte": "2020/01/01",  "lte": "2020/01/31"  }  }  }  } | GET /products/\_search  {  "query": {  "range": {  "created": {  "time\_zone": "+01:00",  "gte": "2020/01/01 01:00:00",  "lte": "2020/02/01 00:59:59"  }  }  }  } |

**Prefix, wildCard ,Expression**

**Prefix:-** Check First Starting terms of the text field

**WildCard** :- match certain pattern

? = match one character (only one allowed after)

\* = match multiple characters afterwards

| **Prefix** | **?** | **\*** |
| --- | --- | --- |
| GET /products/\_search  {  "query": {  "prefix": {  "name.keyword": {  "value": "Past"  }  }  }  } | GET /products/\_search  {  "query": {  "wildcard": {  "tags.keyword": {  "value": "Past?"  }  }  }  } | GET /products/\_search  {  "query": {  "wildcard": {  "tags.keyword": {  "value": "Bee\*"  }  }  }  } |

## 

| Regex: | Query by Existence |
| --- | --- |
| GET /products/\_search  {  "query": {  "regexp": {  "tags.keyword": {  "value": "Bee(f|r)+"  } } }} | GET /products/\_search  {  "query": {  "exists": {  "field": "tags.keyword"  } }}  **SQL** :- Where IS NOT NULL |

## **FULL TEXT MATCH**

1. Query *Unstructured Data*, (For *Long Texts* like *Description*, blog)
2. Analysed (Indexed values)

**MATCH QUERY** : - Can Use it for :- NUMBER, DATE ,BOOLEAN

| GET /products/\_search  {  "query": {  "match": {  "name": {  "query": "pasta chicken",  "operator": "and"  } } }} | Default – **OR** applied (i.e pasta OR chicken)  Can Change to **AND** (i.e Pasta AND chicken) |
| --- | --- |

## **RELEVANCE SCORING**

*Most match word* → *Highest score* → *Sort by highest Score*

| GET /products/\_search  {  "query": {  "multi\_match": {  "query": "vegetable",  "fields": ["name^2", "tags"]  } }} | Search Multiple fields  If matched in Name field the Relevance Score is multiplied by 2 |
| --- | --- |

**Phrase Searches →** *Match phrases*, *Order Matters*

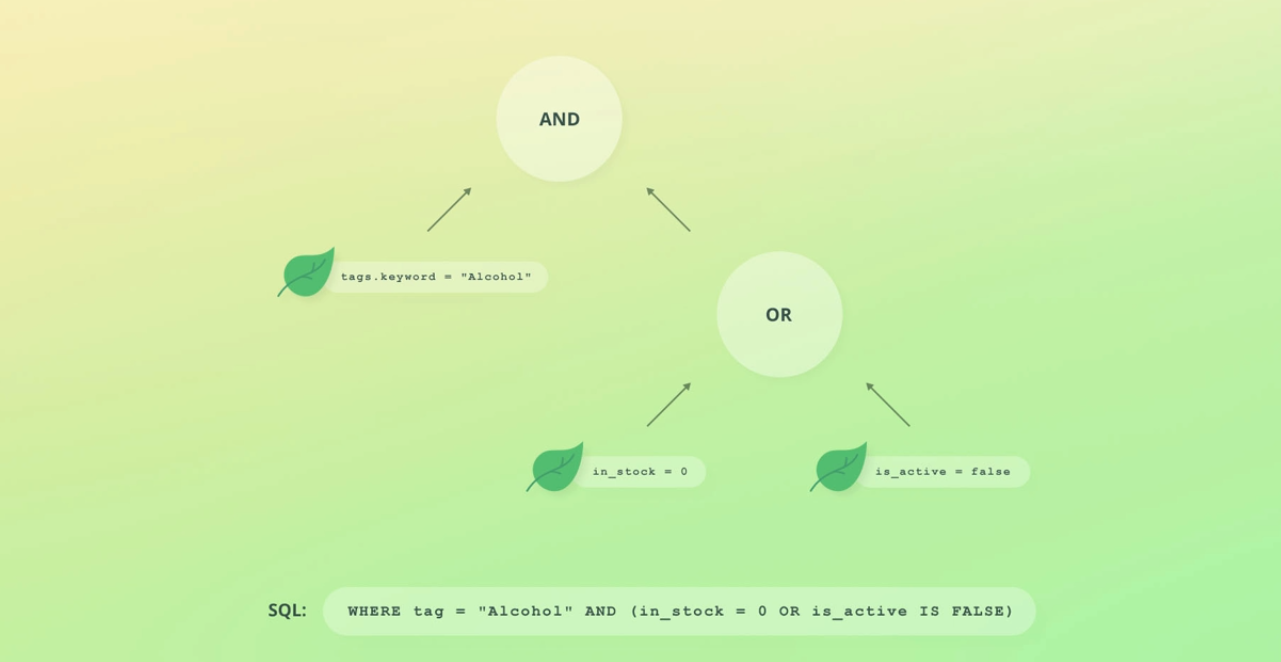
| GET /products/\_search  {  "query": {  "match\_phrase": {  "description": "browse the internet"  } }} | Search in description field ,  Should be in in the same order to match |
| --- | --- |

A query in elastic search is made up of Two clauses:

1. **Leaf Query Clauses**: These clauses are match, term or range, which look for a specific value in specific field
2. **Compound Query Clauses**: These queries are a combination of leaf query clauses and other compound queries to extract the desired information

Every document field can be classified either as an exact value or analysed text (also called full text). Exact values are fields like user\_id, date, email\_address, etc. Analysed Text is text data like product\_description or email\_body.

Full text searching is a highly complex operation and involves different analyzer packages depending on the type of text data. For example, some analyzer packages are language specific which are used to analyse text in certain languages. The default analyzer package is the standard analyzer which splits text by word boundaries, lowercases and removes punctuation.



Multiple Leaf queries are combined to make the Compound Query

| GET /products/\_search  {  "query": {  "bool": {  "filter": [ {  "range": {"in\_stock": {"lte": 100 } } },  {  "term": {  "tags.keyword": "Beer"  } }  ],  "should": [  { "match": { "name": "Beer" } },  { "match": { "description": "Beer" } }  ],  "minimum\_should\_match": 1  } }} | Filter, term, should,match -> are leaf queries , They combined together to make the Compound Query,  **Bool Query :**- allow multiple queries combining, (must, should, must\_not, filter) |
| --- | --- |

**QUERY EXECUTION CONTEXTS**

| **QUERY EXECUTION CONTEXT** | **FILTER EXECUTION CONTEXT** |
| --- | --- |
| How well does Document Match? (\_Score) | ***Relevance Score*** is ***Not calculated*** |
| Most Relevant → TOP at Search | For Structured data(Num,Date,Keyword) |
| ***Relevance Score*** is ***calculated***  EG :- “Query”:{ match:{}} | Improved Performance (Query is cached)  EG :- filter:[{},{}]  Must\_not: [{},{}] |

**CHANGE QUERY EXECUTION CONTEXT**

{

"query": {

"bool": {

"must": [{ "match": { "title": "search term" } }], → **This is Query Execution Context**

"filter": [{ "range": { "price": { "gte": 100, "lte": 500 } } }] → **Filter Execution Context**

}

}

}

**BOOSTING QUERY**

**Boost** **Relevance** **Score** that **meets** **certain criteria** and decrease of some

{

"size": 20,

"query": {

"boosting": {

"positive": { "match": { "name": "juice" } }, → **Boost the score if name field**==Juice

"negative": { "match": { "name": "apple" } }, →***Decrease*** *the* ***score*** *if* ***doc******name*** ==Apple

"negative\_boost": 0.5

}

}

}

**How to Query Nested Data Types?**

**Problem** :- When you query it will return the root document ?

**Solution** :- Use **Nested Inner Hits**

| {  "mappings": {  "properties": {  "ingredients": {  "type": "nested",  "properties": {  "name": { "type": "text" },  "amount": { "type": "integer" }  }  },  "email": { "type": "keyword" }  }  }  } | {  "query": {  "nested": {  "path": "ingredients",  **"inner\_hits": { "name": "my\_hits", "size": 10 },**  "query": {  "bool": {  "must": [  { "match": { "ingredients.name": "parmesan" } },  { "range": { "ingredients.amount": { "gte": 100 } } }  ]  } } }}  } |
| --- | --- |

This Query only **gets** the **nested data** that **match** the **condition** if (**inner\_hits**) was **not there** then it will **return** the **root document** only(**email**).

**PARENT -CHILD RELATIONSHIP**

1. **Child - parent** are **stored** in **same SHARD** but differently
2. **Updating child** - **does not affect parent** (**saves** **INDEXING**)
3. **Search/Sort** – **Difficult**
4. **One to many** → If **updating** is required **frequently…**

| MAPPING PARENT -CHILD  PUT /department/\_mapping  **{**  "properties": {  **"join\_field": {**  **"type": "join",**  **"relations": {**  **"department": "employee"**  } } }} | Add Department  PUT /department/\_doc/1  **{**  **"name": "Development",**  **"join\_field": "department"**  **}** |
| --- | --- |

| Add Employee  PUT /department/\_doc/3?**routing=1**  {  "name": "Bo Andersen",  "age": 28,  "gender": "M",  **"join\_field": {**  **"name": "employee",**  **"parent": 1**  **}**  }  **Routing == 1**  Must be defined (Default routing == parent\_id)  To store parent-child in same shard | **Query By Parent\_Id**  GET /department/\_search  **{**  **"query": {**  **"parent\_id": {**  **"type": "employee", "id": 1**  **}**  **}**  **}** |
| --- | --- |

| Matching child documents by parent criteria  GET /department/\_search  **{**  **"query": {**  **"has\_parent": {**  **"parent\_type": "department",**  **"query": {**  **"term": {**  **"name.keyword": "Development"**  **}}}}**  **}** |  |
| --- | --- |

**MULTI LEVEL RELATION**

| PUT /company  **{**  **"mappings": {**  **"properties": {**  **"join\_field": {**  **"type": "join",**  **"relations": {**  **"company": ["department",supplier"],**  **"department": "employee"**  **}**  **}}}**  **}** |  |
| --- | --- |

**TERM LOOKUP MECHANISM**

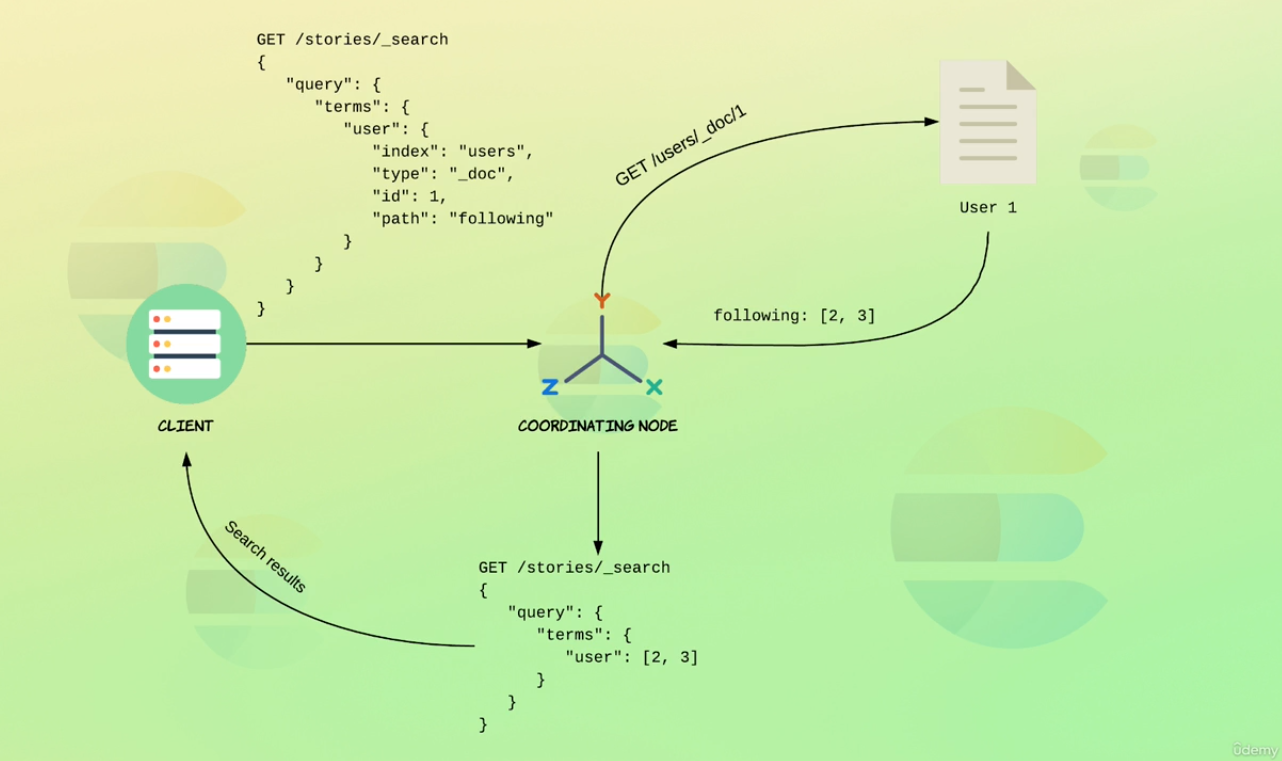
* **Data Enrichment**: Terms lookup allows enriching search queries by retrieving terms from different indices or documents, enabling more sophisticated and context-aware search operations.
* **Flexible Querying**: It offers a means to use terms from one location to filter or query data in another location, enhancing the flexibility and power of Elasticsearch search capabilities.

| PUT /users/\_doc/1  {  "name": "John Roberts",  "following" : [2, 3]  } | PUT /users/\_doc/2  {  "name": "Elizabeth Ross",  "following" : []  } | PUT /users/\_doc/3  {  "name": "Jeremy Brooks",  "following" : [1, 2]  } |
| --- | --- | --- |

| PUT /stories/\_doc/1  {  "user": 3,  "content": "Wow look, a penguin!"  } | PUT /stories/\_doc/2  {  "user": 1,  "content": "Just another day at the office... #coffee"  } | PUT /stories/\_doc/3  {  "user": 1,  "content": "Making search great again! #elasticsearch #elk"  } |
| --- | --- | --- |

| GET /**stories**/\_search  {  "query": {  "terms": {  **"user": {**  **"index": "users",**  **"id": "1",**  **"path": "following"**  } }}  } | This Elasticsearch query performs a search in the "stories" index. It utilizes the "terms" query, which functions based on a list of terms provided.  In this case, it is searching for stories where the field "**user**" matches any value present in the specified list of users that the user with ID "1" follows.  The "**path**" attribute refers to the nested field within the "users" index where these users being followed by user ID "1" are stored. |
| --- | --- |

INTERNALLY



Saves us from querying 1 times (network latency increases) Es does by itself

**LIMITATION OF JOIN**

* Document must stored within same Index
* Parent -child must be on same shard
* Performance Expensive so avoid

**CONTROL QUERY RESULTS →**Fetch only required fields

Can do this by specifying **\_Source** field

Good for performance

| Only return Created field  GET /recipes/\_search  {  **"\_source": "created",**  "query": {  "match": { "title": "pasta" }  }  } | Including all of the ingredients object's keys, except the name key GET /recipes/\_search  **{**  **"\_source": {**  **"includes": "ingredients.\*",**  **"excludes": "ingredients.name"**  **},**  "query": {  "match": { "title": "pasta" }  }  } |
| --- | --- |

**PAGINATION**

| GET /recipes/\_search  {  "\_source": false,  **"size": 2,**  **"from": 2,**  "query": {  "match": {"title": "pasta"}  }} | **SIZE** :- No of document to fetch  **FROM** :- **OFFSET** (SQL) no of doc to skip |
| --- | --- |

**AGGREGATIONS**

* **Purpose**: Aggregations in Elasticsearch are used to gather and process data, summarising and providing insights into large sets of data.(Same AS SQL)

**Types of Aggregations**:

* **Metric Aggregations**: Calculate metrics like average, sum, minimum, maximum, etc.
* **Bucket Aggregations**: Group documents into "buckets" based on certain criteria.
* **Pipeline Aggregations**: Perform operations on the output of other aggregations.

**Metric Aggregation (** SUM, AVG, MAX,MIN,STATS ,EXTENDED\_STATS)

| Retrieving the number of **distinct** values GET /orders/\_search  {  "size": 0,  "aggs": {  "total\_salesmen": {  **"cardinality": {**  **"field": "salesman.id"**  } } }  } | Retrieve (**count,sum,avg,min,max**) in One  STAT  GET /orders/\_search  {  "size": 0,  "aggs": {  "amount\_stats": {  "stats": {  "field": "total\_amount"  }}}  } |
| --- | --- |

**BUCKET AGGREGATION (**Group documents into buckets acc. To some criteria)

****

| GET /orders/\_search  {  "size": 0,  "aggs": {  "status\_terms": {  "terms": {  "field": "status"  }  }  }  }  It Creates different buckets for each status | OUTPUT : -  "aggregations": {  "status\_terms": {  "doc\_count\_error\_upper\_bound": 0,  "sum\_other\_doc\_count": 0,  **"buckets": [**  **{ "key": "processed", "doc\_count": 209 },**  **{ "key": "completed", "doc\_count": 204 },**  **{ "key": "pending", "doc\_count": 199 }**  **]**  }  } |
| --- | --- |

| GET /orders/\_search  {  "size": 0,  **"aggs": {**  **"status\_terms": {**  **"terms": {**  **"field": "status", →** Makes **Bucket** for each **distinct** **value** of **status**  **"size": 20, →** Include **20** instead of 10(Default)  **"missing": "N/A", →** Put NULL value in N/A Bucket  **"min\_doc\_count": 0, →***Min no. of doc* to **present** to **create** **bucket**  **"order": {**  **"\_key": "asc" →** Order them in asc order  **}**  **}**  **}**  **}**  } |
| --- |



**First Aggregation** runs in the **parent query result** ,**Second Aggregation** runs

in the first **aggregation result.**

**DEFINE RULES FOR BUCKET → FILTERS**

**{**

"aggs": {

"my\_filter": {

"filters": {

"filters": {

"pasta": { "match": { "title": "pasta" } },

"spaghetti": { "match": { "title": "spaghetti" } }

}

}

}

}

}

| **RANGE AGGREGATION**  {  "size": 0,  "aggs": {  "amount\_distribution": {  "range": {  "field": "total\_amount",  "ranges": [  { "to": 50 },  { "from": 50, "to": 100}  { "from": 100 }  ]  }  }  }  } | **DATE AGGREGATION**  GET /orders/\_search  {  "aggs": {  "purchased\_ranges": {  "date\_range": {  "field": "purchased\_at",  "ranges": [  {  "from": "2016-01-01",  "to": "2016-01-01||+6M"  },  {  "from": "2016-01-01||+6M",  "to": "2016-01-01||+1y"  }  ]  }  }  }  } |
| --- | --- |

**HISTOGRAMS(**Create bucket of fixed variables)

GET /orders/\_search

{

"size": 0,

"aggs": {

"amount\_distribution": {

"histogram": {

"field": "total\_amount",

"interval": 25, → divide the bucket by 25 [0-25,25-50…]

"min\_doc\_count": 5 → min no of doc req. To form bucket

"extended\_bounds": { → Custom range ( even if price is till 100 , it will still go upto 500 even there are 0 documents

"min": 0,

"max": 500

}

}

}

}

}

**DATE HISTOGRAM**

| **GET /orders/\_search**  {  "size": 0,  "aggs": {  "orders\_over\_time": {  "date\_histogram": {  "field": "purchased\_at",  "calendar\_interval": "month" **→Bucket** of **each month**  }  }  }  } |
| --- |

| **Aggregating nested objects** GET /department/\_search  {  "size": 0,  "aggs": {  "employees": {  "nested": {  "path": "employees"  }  }  }  } |
| --- |

**PROXIMITY SEARCHES**

* Find word , phrase that occur near (jova → java )
* No. of order Not required to specify ( How many word can be wrong) EDIT DISTANCE
* NOTE :- I**nverted index** also **puts the position** of **character** it is **placed** in that **field**

| GET /proximity/\_search  {  "query": {  "match\_phrase": {  "title": {  "query": "spicy sauce",  **"slop": 2**  }  }  }  } |  |
| --- | --- |

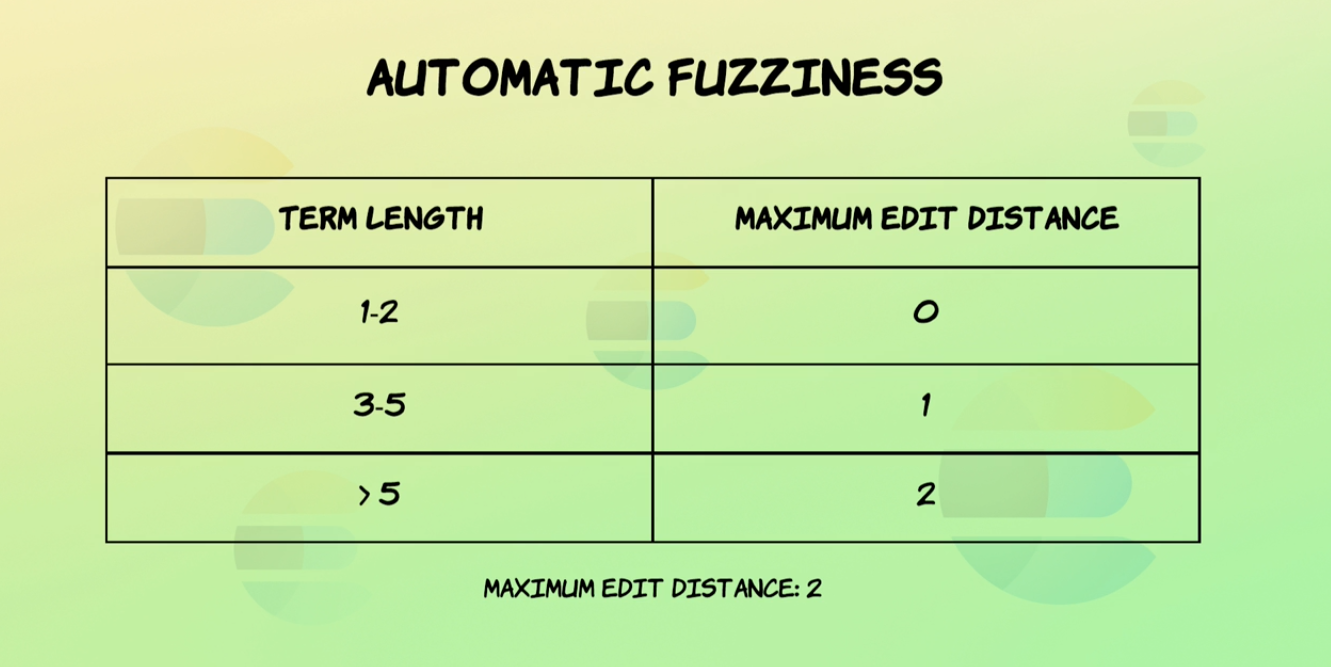
It need to move two steps or places to match the query (Slop Distance)

**FUZZY QUERIES(** Result similar Not Exact)

* Search Similar terms, allow Variation of spellings, Transposition of Letters

Internal working

* Edit distance, [ No or char to transform one term to Another]
* Lavenstein distance Algorithm
* **Fuzziness** :- Define allowed distance

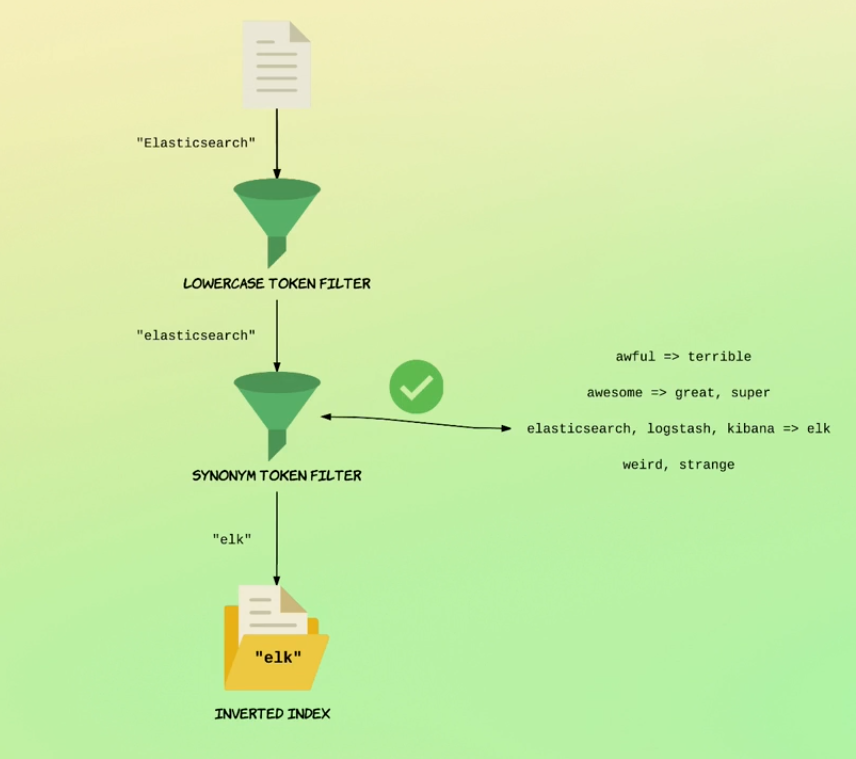


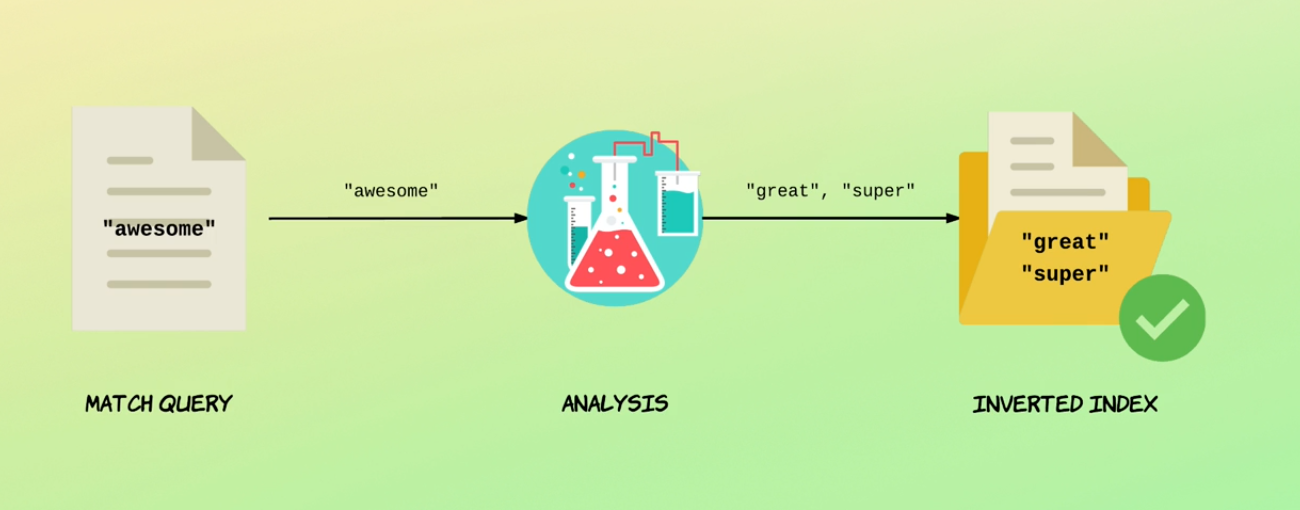
| **fuzzy query** GET /products/\_search  {  "query": {  "fuzzy": {  "name": {  "value": "LOBSTER",  "fuzziness": "auto"  }  }  }  } | **Searching with fuzziness set to auto** GET /products/\_search  {  "query": {  "match": {  "name": {  "query": "jOva",  "fuzziness": "auto"  }  }  }  } |
| --- | --- |

**ADDING SYNONYMS**

* It add the synonyms to the inverted index and next time if we search the word it will run through the same analyzer and finds it for you

| {  "settings": {  "analysis": {  "filter": {  "synonym\_test": {  "type": "synonym",  **"synonyms": [**  **"awful => terrible",**  **"awesome => great, super",**  **"elasticsearch, logstash, kibana => elk",**  **"weird, strange"**  **]**  }  },  **"analyzer": {**  **"my\_analyzer": {**  **"tokenizer": "standard",**  **"filter": ["lowercase", "synonym\_test"]**  **}**  **}**  }  },  "mappings": {  "properties": {  **"description": {**  **"type": "text",**  **"analyzer": "my\_analyzer"**  **}**  }  }  } |
| --- |

****

****

Even if there is no great in awesome in inverted index it will run through analyzer and finds it

Adding synonyms analyzer in the middle can affect. As previous terms are not synonymized

**ADD SYNONYMS FROM THE FILE**

| PUT /synonyms  {  "settings": {  "analysis": {  "filter": {  "synonym\_test": {  "type": "synonym",  "synonyms\_path": "analysis/synonyms.txt"  }  },  "analyzer": {  "my\_analyzer": {  "tokenizer": "standard",  "filter": [  "lowercase",  "synonym\_test"  ]  }  }  }  },  "mappings": {  "properties": {  "description": {  "type": "text",  "analyzer": "my\_analyzer"  }  }  }  } |
| --- |

**HILIGHLIGHT MATCH WORDS**

If the terms match it will highlight the match words for you . You can also do it at Application Level.

| **Highlighting matches within the description field** GET /highlighting/\_search  {  "\_source": false,  "query": {  "match": { "description": "Elasticsearch story" }  },  "highlight": {  "fields": {  "description" : {}  }}  } |
| --- |