Enhancement of Searching and analyzing the document using Elastic Search

Subhani shaik[1], Nallamothu Naga Malleswara Rao[2]

[1]*Asst.professor, Dept. of CSE, St. Mary's Group of Institutions Guntur, Chebrolu, Guntur, A.P, India.*
[2]*Professor, Department of IT, RVR & JC College of Engineering, Chowdavaram, Guntur, A.P, India.*

---***---

Abstract - *Elasticsearch is an open source distributed document store and search engine that stores and retrieves data structures in near real-time. Elasticsearch represents data in the form of structured JSON documents, and makes full-text search accessible via RESTful API and web clients for languages like PHP, Python, and Ruby. It's also elastic in the sense that it's easy to scale horizontally. Elasticsearch provide lots of advanced features for adding search to your application. In this paper, Searching and Analyzing Data with Elasticsearch will be covered by introducing the basic building blocks of search algorithms, how inverted index will be created for the documents you want to search, perform a variety of search queries on these documents, how to explore the TF/IDF for search ranking and relevance, and the important factors which determine how a document is scored for every search term.*

***Key Words*: Inverted Index, stopwords, Tokenizer, Filter, TF/IDF.**

1. INTRODUCTION

In the world of big data, it is unimaginable to use traditional techniques such as RDBMS to analyze the data, as volume of the data is growing very quickly. Big data offers the solution for analyzing huge amount of data. Using Elastic search, access to data can be made even quicker. Elastic search is a search engine based on Lucene. Elastic search uses the concept of indexing to make the search quicker. This paper elaborates the search technique of Elastic search.

Elasticsearch is a schema less big data technology that uses the indexing concept. It is a document oriented tool. That means once the document is added, it can be searched within a second. Elasticsearch can be used for many use cases like analytics store, auto completer, spell checker, alerting engine, and as a general purpose document store; Full text search is one of it. It is a robust search engine that provides a quick full text search over various documents. It searches within full text fields to find the document and return the most relevant result first. The relevancy of documents is good as Elasticsearch uses boolean model to find document. As soon as a document matches a query, Lucene calculates its score for that query, combining the scores of each matching term. The relevance of the document can be calculated using practical scoring function.

2. SEARCHING WITH ELASTIC SEARCH

The search feature is a central part of every product today. Elastic search is one of the most popular open source technologies which allow you to build and deploy efficient and robust search quickly. Elastic search database is document oriented. By default, the full document is returned as part of all searches. This is referred to as the source. If the entire source document is not to be returned, then only a few fields from within source can be returned. The Elastic search uses the inverted index to search the term. The terms are sorted in ascending order.

Elastic search provides the ability to subdivide the index into multiple pieces called shards. When a new document is stored and indexed, Elastic search server defines the shard responsible for that document. When an index is created, user can simply define the number of shards. Each shard is in itself a fully-functional and independent "index" that can be hosted on any node in the cluster. Any number of documents can be uploaded irrespective of its type. Indexes are used to group documents and each document is stored using a certain type. Shards are used to distribute parts of an index across several nodes and replicas are copies of shards that are used for distributing load as well as for fault tolerance.

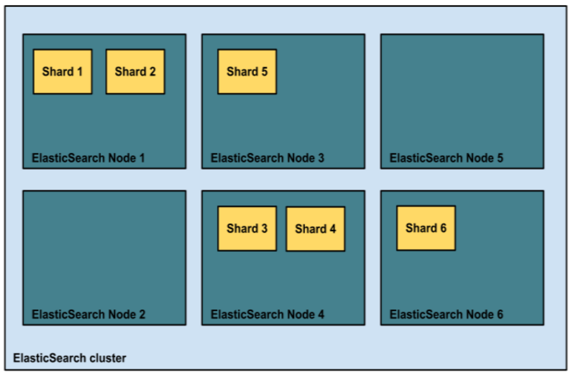


Fig.1: Creating Shards in Nodes

2.1 Creation Of Inverted Index Using Elasticsearch

Elasticsearch uses the concept of inverted index for searching. When the query is fired, Elasticsearch looks into inverted index table to find the required data. It will show the relevant document in which the term is contained. The inverted index searches the relevant document by mapping

the term to its containing document. In the dictionary the terms are sorted which results in quick search.
The act of storing data in Elastic search is called *indexing*. Before indexing a document, we need to decide *where* to store it. An Elastic search cluster can contain multiple *indices*, which in turn contain multiple *types*. These types hold multiple *documents*, and each document has multiple *fields*.

2.1.1 Creating an employee directory

Index a *document* per employee to include all the details of an employee. Each document will be of *type* employee. That type will live in the stmarys *index* that resides within the Elasticsearch cluster.

PUT /stmarys/employee/501

```
{
        "first_name" : "Subhani",
        "last_name" :  "Shaik",
        "age" :       25,
        "about" : “I like to play cricket",
        "interests": [ "sports", "music" ]
}
```

PUT /stmarys/employee/502

```
        {
        "first_name" :  "Firoze",
        "last_name" :   "Shaik",
        "age" :        32,
        "about" :      "I like to play football ",
        "interests":  [ "music" ]
        }
        PUT / stmarys/employee/503
        {
        "first_name" :  "Lakshman",
        "last_name" :   "Pinapati",
        "age" :        35,
        "about":"I like to read books",
         "interests":  [ "study" ]
        }
```

Every path contains three pieces of information: The index name, the type name and The ID of this particular employee. The request body contains all the information about this employee. His name is Subhani Shaik, he’s 25 years old, and enjoys playing cricket. Now we have some data stored in Elastic search.

Retrieving a Document

If we want to retrieve the data of a particular employee execute an HTTP GET request and specify the *address* of the document—the index, type, and ID.

GET /stmarys/employee/501

The response contains some metadata about the document, and Subhani Shaik’s original JSON document as the _source field:

```
{
    "_index" :  "stmarys",  "_type" : "employee",
    "_id" :     ‘501",  "_version" : 1,
    "found" :   true,  "_source" : {
    "first_name" : "Subhani",  "last_name": "Shaik",
       "age" : 25, "about" : "I like to play cricket",
       "interests": [ "sports", "music" ]
     }
}
```

Searching a document:

To retrieve three documents in the result array use the _search endpoint.

GET /stmarys/employee/_search

```
{
                "took":6,"timed_out": false,
                "shards": { ... },”result ": {
                "total": 3,  "max_score":  1,
                "result": [
                {
                   "_index": "stmarys",
                   "_type" : "employee",
                   "_id"   : "503", "_score":  1,
                   "_source": {
                   "first_name":  "Lakshman",
                   "last_name" :  "Pinapati",
                   "age": 35,
                   "about":"I like to read books",
                   "interests":  [ "study" ]
                            }
              },
  {
      "_index": "stmarys"
      "_type": "employee",
      "_id": "501","_score":1,
      "_source": {
      "first_name" : "Subhani",
      "last_name" : "Shaik",
      "age" : 25,
      "about": "I like to play cricket",
      "interests" : [ "sports", "music" ]
      }
   },
  {
      "_index": "stmarys","_type": "employee",
      "_id": "2","_score":1,
      "_source": {
      "first_name" :  "Firoze","last_name" :"Shaik",
      "age" : 32, "about" : "I like to play football ",
      "interests":  [ "music" ]          }
     }     ]
  }
}
```

To Search for employees who have “Shaik” in their last name use a *lightweight* search method that is easy to use from the command line. This method is often referred to as a *query-string* search, since we pass the search as a URL query-string parameter:

GET/stmarys/employee/_search? q=last_name:Shaik

We use the same _search endpoint in the path, and we add the query itself in the q= parameter. The results that come back show all Shaiks:

```
{
  ...
  "result": {
    "total":     2,
    "max_score":  0.30685282,
    "result": [
      {
        ...
        "_source": {
"first_name"              : "Subhani",
"last_name"               : "Shaik",
"age"                     : 25,
"about" : "I like to play cricket",
"interests":[ "sports", "music" ]
                  }
      },
{
        ...
"_source": {
"first_name" :  "Firoze",
"last_name" :  "Shaik",
"age"        :  32,
"about"      :  "I like to play football ",
"interests":  [ "music" ]
        }      }
    ]  }
}
```

Search with a query DSL

Query-string search is handy for ad hoc searches from the command line, but it has its limitations. Elastic search provides a rich, flexible, query language called the *query DSL*, which allows us to build much more complicated, robust queries.

The *domain-specificlanguage* (DSL) is specified using a JSON request body. We can represent the previous search for all Smiths like so:

```
GET /stmarys/employee/_search
{
  "query" : {
    "match" : {
      "last_name" : "Shaik"
         }   }
}
```

Full Text Search:

to search for all employees who enjoy rock climbing:

```
GET /megacorp/employee/_search
{
  "query" : {
    "match" : {
    "about" : " Play Cricket"
    }
  }
}
```

You can see that we use the same match query as before to search the about field for "Play Cricket". We get back two matching documents:

```
{
  ...
  "result": {
    "total":     2, "max_score":  0.16273327,
    "result": [
      {
        ...

 "_score":       0.16273327,
 "_source": {
     "first_name": "Subhani", "last_name": "Shaik",
     "age"  : 25,"about" : "I like to play cricket",
     "interests": [ "sports", "music" ]
       }
     },
     {
       ...

          "_score":  0.016878016,
          "_source": {
          "first_name" :  "Firoze","last_name" :   "Shaik",
          "age"  : 32, "about" : "I like to play football",
           "interests"  :  [ "music" ]
       }
     }    ]  }}
```

Elasticsearch sorts matching results by their relevance score, that is, by how well each document matches the query. The first and highest-scoring result is obvious: Subhani Shaik's about field clearly says "Play Cricket" in it.

But why did Firoze Shaik come back as a result? The reason his document was returned is because the word "Play" was mentioned in about field. Because only "Play" was mentioned, and not "Cricket," his _score is lower than Subhani's.

Phrase Search

To match exact sequences of words or *phrases* we can use a slight variation of the match query called the match_phrasequery:

```
GET /stmarys/employee/_search
{
  "query" : { "match_phrase" : {
    "about" : " Play Cricket"
                                 }
                 }
}
```

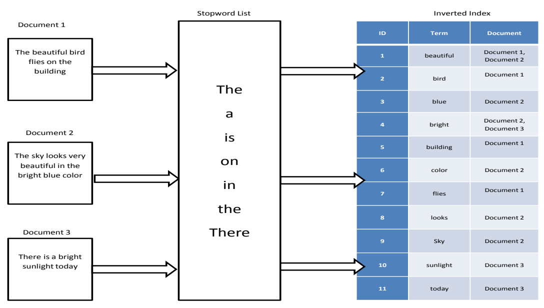
This returns only Subhani Shaik's document:

```
{
  ...
 "result": {  "total":1, "max_score":  0.23013961,
 "result": [
  {    ...
    "_score": 0.23013961,  "_source": {
     "first_name"       : "Subhani",
     "last_name"        : "Shaik",
      "age"  : 25,"about" : "I like to play cricket",
      "interests": [ "sports", "music" ]
         }
       }
     ]
   }
}
```

Above query gives an output that will match only employee records that contain both "Play" *and* "cricket" *and* that display the words next to each other in the phrase "Play Cricket".

2.2 Using stopword list to fasten the search process.

Elastic search uses the concept of Stopword list to fasten the search process. Elasticsearch has its own list of predefined stopwords. Stopwords can usually be filtered out before indexing. The inverted index is then created over the terms of document to make search faster.


Document 1
The beautiful bird flies on the building
Document 2
The sky looks very beautiful in the bright blue color
Document 3
There is a bright sunlight today
Stopword List
The
a
is
on
in
the
There
Inverted Index

ID	Term	Document
1	beautiful	Document 1, Document 2
2	bird	Document 1
3	blue	Document 2
4	bright	Document 2, Document 3
5	building	Document 1
6	color	Document 2
7	flies	Document 1
8	looks	Document 2
9	Sky	Document 2
10	sunlight	Document 3
11	today	Document 3

Fig.2: Stopword list of Elastic search

2.2.1 Stopwords

To use custom stopwords in conjunction with the standard analyzer, all we need to do is to create a configured version of the analyzer and pass in the list of stopwords that we require:

```
PUT /my_index
{
 "settings": {  "analysis": {  "analyzer": {


    "my_analyzer": {
     "type": "standard",
     "stopwords": [ "and", "the" ]
    }   }
    }   }
}
```

2.2.2 Specifying Stopwords

Stopwords can be passed inline, by specifying an array:"stopwords": ["and", "the"]. The default stopword list for a particular language can be specified using the _lang_ notation: "stopwords": "_english_". Stopwords can be disabled by specifying the special list: _none_. To use the english analyzer without stopwords, you can do the following:

```
PUT /my_index
{
 "settings": {  "analysis": {  "analyzer":  {  "my_english":
           {  " type"      :    "english",
                "stopwords"  :    "_none_"
                 } }
           } }
}
```

Stopwords can also be listed in a file with one word per line. The file must be present on all nodes in the cluster, and the path can be specified with the stopwords_path parameter:

```
PUT /my_index
{
 "settings": {  "analysis": {  "analyzer":
          {  "my_english": {
             "type":        "english",
             "stopwords_path":      "stopwords/english.txt"


                          }    }
          } }
}
```

2.2.3 Updating Stopwords

Updating stopwords is easier if you specify them in a file with the stopwords_path parameter. You can just update the file on every node in the cluster and then force the analyzers to be re-created by either Closing and reopening the index or restarting each node in the cluster, one by one.

2.2.4 Stop words and performance

The major drawback of keeping stopwords is that of performance. When Elasticsearch performs a full-text search, it has to calculate the relevance _score on all matching documents in order to return the top ten matches. What we need is a way of reducing the number of documents that need to be scored. The easiest way to reduce the number of documents is simply to use and operator with the match query, in order to make all words required.

A match query like this:

```
{
   "match": {
     "text": {
       "query":   "the quick brown fox",
       "operator": "and"
     }   }
}
```

is rewritten as a bool query like this:

```
{
   "bool": {
     "must": [
       { "term": { "text": "the" }},
       { "term": { "text": "quick" }},
       { "term": { "text": "brown" }},
        { "term": { "text": "fox" }}
            ]
    }
}
```

The bool query is intelligent enough to execute each term query in the optimal order—it starts with the least frequent term. Because all terms are required, only documents that contain the least frequent term can possibly match. Using the and operator greatly speeds up multi term queries.

3. ANALYZING DATA USING ELASTIC SEARC

The analyzing process is one of the main factors for determining the quality of search. In Elastic search, how fields are analyzed is determined by the mapping of the type. The analyzing process for a certain field is determined once and cannot be changed easily. The process of tokenization and normalization which is called analysis can be used to fasten the search process.

3.1 TOKENIZATION

A *tokenizer* receives a stream of characters, breaks it up into individual *tokens*, and outputs a stream of *tokens*. The tokenizer is also responsible for recording the order or *position* of each term and the start and end *character offsets* of the original word which the term represents. Elasticsearch has a number of built in tokenizers to build custom analyzers.

3.1.1 Word Oriented Tokenizers

The following tokenizers are used for tokenizing full text into individual words:

- **Standard Tokenizer :** Divides text into terms on word boundaries, as defined by the Unicode Text Segmentation algorithm. It removes most punctuation symbols.
- Letter Tokenizer : Divides text into terms whenever it encounters a character which is not a letter.
- Lowercase Tokenizer : Divides text into terms whenever it encounters a character which is not a letter, but it also lowercases all terms.
- Whitespace Tokenizer : Divides text into terms whenever it encounters any whitespace character.
- UAX URL Email Tokenizer : Similar to the standard tokenizer except that it recognizes URLs and email addresses as single tokens.
- Classic Tokenizer : A grammar based tokenizer for the English Language.
- Thai Tokenizer : Segments Thai text into words.

3.1.2 Partial Word Tokenizers

These tokenizers break up text or words into small fragments, for partial word matching:

- N-Gram Tokenizer : Breaks up text into words when it encounters any of a list of specified characters (e.g. whitespace or punctuation), then it returns n-grams of each word: a sliding window of continuous letters, e.g. quick → [qu, ui, ic, ck].
- Edge N-Gram Tokenizer : Breaks up text into words and returns n-grams of each word which are anchored to the start of the word, e.g. quick → [q, qu, qui, quic, quick].

3.1.3 Structured Text Tokenizers

Structured text like identifiers, email addresses, zip codes, and paths, rather than with full text will use the following tokenizers

- Keyword Tokenizer : The keyword tokenizer is a "noop" tokenizer that accepts whatever text it is given and outputs the exact same text as a single term. It can be combined with token filters like lowercase to normalize the analyzed terms.
- Pattern Tokenizer : The pattern tokenizer uses a regular expression to either split text into terms whenever it matches a word separator, or to capture matching text as terms.
- Path Tokenizer : The path_hierarchy tokenizer takes a hierarchical value like a filesystem path, splits on the path separator, and emits a term for each component in the tree, e.g. /foo/bar/baz → [/foo, /foo/bar, /foo/bar/baz].

3.1.4 Email-link tokenizer

In circumstances where we have URLs, emails, or links to be indexed, a problem comes up when we use the standard tokenizer.

Consider we index the following two documents in an index:

```
curl -XPOST '<a ref="http://localhost:9200/analyzers-
blog-03/emails/1">http://localhost:9200/analyzers-
blog-03-01/emails/1</a>' -d '{<br />
"email": "stevenson@gmail.com"<br />}'
curl -XPOST '<a ref="http://localhost:9200/analyzers-
blog-03/emails/2">http://localhost:9200/analyzers-
blog-03-01/emails/2</a>' -d '{<br />
"email": "jennifer@gmail.com"<br />}'
```

Here you can see that we only have email ids in each document.

Run the following query

```
curl -XPOST  '<a
href="http://localhost:9200/analyzers-blog-
03/emails/_search?&pretty=true&size=5">http://local
host:9200/analyzers-blog-03-
01/emails/_search?&pretty=true&size=5</a>' -d '{
 "query": {
 "match": {
 "email": "stevenson@gmail.com"
          }
 }}'
```

Here, the standard tokenizer, split the values in the field "email" at the "@" character. i.e., "stevenson@gmail.com" is split into "stevenson" and "gmail.com." This happens for all the documents and this is why all the documents have "gmail.com" as a common term. Here the query must return only the first document but the response consists of all the other documents.

We can solve this issue by using the "UAX_Email_URL" tokenizer instead of the default tokenizer. The UAX_Email_URL tokenizer works the same as the standard tokenizer, but it can recognize URLs and emails and will output them as single tokens.

3.2 FILTERS

3.2.1 Edge-n-gram token filter

Most of our searches are single word queries, but not in all circumstances. For example, if we are implementing an autocomplete feature, we might want to have the feature of substring matching too. So if we are searching for "prestige," the words "pres", "prest" etc should match against it. In Elasticsearch, this is possible with the "Edge-Ngram" filter. So let's create the analyzer with "Edge-Ngram" filter as below:

```
curl -X PUT
 "<a href="http://localhost:9200/analyzers-blog-03">
http://localhost:9200/analyzers-blog-03</a>-02" -d '
{ "index": {   "number_of_shards": 1,
              "number_of_replicas": 1  },
 "analysis": {    "filter": {   "ngram": {
 "type":"edgeNGram",min_gram":2,"max_gram": 50
    }  },
  "analyzer": {   "NGramAnalyzer": {
     "type": "custom", "tokenizer": "standard",
     "filter": "ngram"    }  }
 } }'
```

The analyzer has been named "NGramAnalyzer," this analyzer would create substrings of all the tokens from one end of the token with the minimum length of two (min_gram) and the maximum length of fifty (max_gram). This kind of filtering can be used to implement the autocomplete feature or the instant search feature in our application.

3.2.2 Phonetic token filter

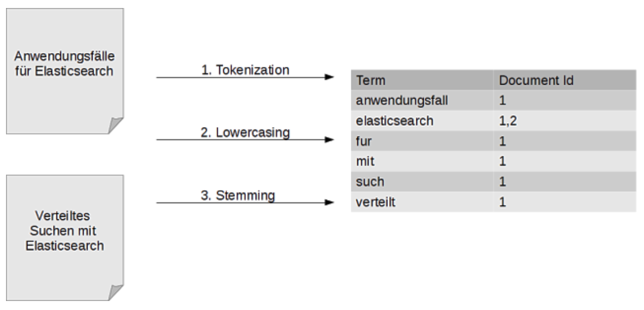
Sometimes we make small mistakes while searching, i.e., we may type "grammer" instead of "grammar." These words are phonetically same but in a dictionary search environment if "grammer" is searched there will not be returning results. Elasticsearch makes use of the Phonetic token filter to search for "Kanada" and still see the results for "Canada."

3.2.3 Stop Token Filter

The stop token filter can be combined with a tokenizer and other token filters to create a custom analyzer

3.3 NORMALIZATION

As search is used everywhere users also have some expectations of how it should work. Instead of issuing exact keyword matches they might use terms that are only similar to the ones that are in the document. We can normalize the text during indexing so that both keywords point to the same term in the document. Lucene, the library search and storage in Elasticsearch is implemented with provides the underlying data structure for search, the inverted index. Terms are mapped to the documents they are contained in. A process called analyzing is used to split the incoming text and add, remove or modify terms.


Anwendungsfälle für Elasticsearch
Verteiltes Suchen mit Elasticsearch
1. Tokenization
2. Lowercasing
3. Stemming

Term	Document Id
anwendungsfall	1
elasticsearch	1,2
fur	1
mit	1
such	1
verteilt	1

Fig.3: Process of Normalization

On the left we can see two documents that are indexed, on the right we can see the inverted index that maps terms to the documents they are contained in. During the analyzing process the content of the documents is split and transformed in an application specific way so it can be put in the index. Here the text is first split on whitespace or punctuation. Then all the characters are lowercased. In a final step the language dependent stemming is employed that tries to find the base form of terms. This is what transforms our Anwendungsfälle to Anwendungsfall.

Elasticsearch does not know what language our string is in so it doesn't do any stemming which is a good default. To tell Elasticsearch to use the German Analyzer instead we need to add a custom mapping. We first delete the index and create it again:

```
curl -XDELETE "http://localhost:9200/conferences/"
curl -XPUT "http://localhost:9200/conferences/“
```

We can then use the PUT mapping API to pass in the mapping for our type.

```
curl -XPUT
"http://localhost:9200/conferences/talk/_mapping" -
d'
{   "properties": { "tags": {
       "type": "string","index":"not_analyzed"
     },
     "title": { "type": "string","analyzer": "german"
     }   }
}'
```

4. TERM FREQUENCY/INVERSE DOCUMENT FREQUENCY (TF/IDF) ALGORITHM

The standard *similarity algorithm* used in Elasticsearch is known as *term frequency/inverse document frequency*, or *TF/IDF.* The list of matching documents need to be ranked by relevance. The relevance score of the document depends on the *weight* of each query term that appears in that document. The weight of a term is determined by Term frequency, Inverse document frequency and Field-length norm

Term frequency

The more often the term appears in the document, the *higher* the weight. A field containing five mentions of the same term is more likely to be relevant than a field containing just one mention. The term frequency (tf) for term t in document d is the square root of the number of times the term appears in the document

$$\mathbf{tf(t\ in\ d) = \sqrt{frequency}}$$

Setting index_options to docs will disable term frequencies and term positions. A field with this mapping will not count how many times a term appears, and will not be usable for phrase or proximity queries. Exact-value not_analyzed string fields use this setting by default.

Inverse document frequency

The more often the term appears in all documents in the collection, the *lower* the weight. The inverse document frequency is calculated as follows:

$$\mathbf{idf(t) = 1 + \log(\ numDocs\ /\ (docFreq + 1))}$$

The inverse document frequency (idf) of term t is the logarithm of the number of documents in the index, divided by the number of documents that contain the term,

Field-length norm

The shorter the field, the *higher* the weight. If a term appears in a short field, such as a title field, it is more likely that the content of that field is *about* the term than if the same term appears in a much bigger body field. The field length norm is calculated as follows:

$$\mathbf{norm\ (d) = 1\ /\ \sqrt{numTerms}}$$

The field-length norm (norm) is the inverse square root of the number of terms in the field.

5. THE PRACTICAL SCORING FUNCTION

The relevance score of each document is represented by a positive floating-point number called the _score. The higher the _score, the more relevant the document. Before Elasticsearch starts scoring documents, it first reduces the candidate documents down by applying a boolean test by checking whether the document match the query or not. Once the results that match are retrieved, the score they receive will determine how they are rank ordered for relevancy.

The scoring of a document is determined based on the field matches from the query specified. Elasticsearch uses the *Boolean model* to find matching documents, and a formula called the *practical scoring function* to calculate relevance. This formula borrows concepts from *term frequency/inverse document frequency* and the *vector space model* but adds more-modern features like a coordination factor, field length normalization, and term or query clause boosting.

score(q,d) = queryNorm(q)* coord(q,d)* SUM (tf(t in d), idf(t)^2, t.getBoost(),norm(t,d)) (t in q)

- score(q,d) - relevance score of document d for query q.
- queryNorm(q) - query normalization factor.
- coord(q,d) - coordination factor.
- tf(t in d) - term frequency for term t in document d.
- idf(t) - inverse document frequency for term t.
- t.getBoost() - boost that has been applied to the query.
- norm(t,d) - field-length norm, combined with the index-time field-level boost, if any.

The practical scoring function can also be improved for better relevancy of documents. The boost parameter is used to increase the relative weight of a clause with a boost greater than 1 or decrease the relative weight with a boost between 0 and 1, but the increase or decrease is not linear. Instead, the new _score is *normalized* after the boost is applied. Each type of query has its own normalization algorithm. A higher boost value results in a higher _score.

6. CONCLUSION

Elastic search is both a simple and complex product. In this paper we have covered how to perform searching and analyzing the data with elastic search. We have discussed how to explore the tf/idf for search ranking and relevance, and the important factors which determine how a document is scored for every search term. We have discussed how elastic search handles a variety of searches, such as full-text queries, term queries, compound queries, and filters. We have discussed the creation of inverted index using elasticsearch,using stopword list to fasten the search process. And finally we have studied how term frequency/inverse document frequency (tf/idf) algorithm explores for calculating practical scoring function.

7. REFERENCES

[1]Survey Paper on Elastic Search Pragya Gupta, Sreeja Nair International Journal of Science and Research (IJSR)

[2]http://www.elasticsearchtutorial.com/basic-elasticsearch-concepts.html

[3]https://www.elastic.co/guide/en/elasticsearch/guide/current/inverted-index.html

[4]https://www.tutorialspoint.com/elasticsearch/elasticsearch_basic_concepts.html

[5]Full-Text Search on Data with Access Control Ahmad Zaky, Rinaldi Munir, S.T., M.T.

School of Electrical Engineering and Informatics Institut Teknologi Bandung.

[6]Use Cases for Elasticsearch: Full Text Search:http://blog.florian-hopf.de/2014/07/use-cases-for-elasticsearch-full-text.html

[7]https://www.elastic.co/guide/en/elasticsearch/guide/current/stopwords.html#stopwords

[8]https://qbox.io/blog/optimizing-elasticsearch-how-many-shards-per-index

BIOGRAPHIES:

Mr. Subhani shaik is working as Assistant professor in Department of computer science and Engineering at St. Mary's group of institutions Guntur, he has 12 years of Teaching Experience in the academics.

Dr. Nallamothu Naga Malleswara Rao is working as Professor in the Department of Information Technology at RVR & JC College of Engineering with 25 years of Teaching Experience in the academics.