CE306- INFORMATION RETRIEVAL (REPORT)

Student ID Number: 1403713/ 1402039

CE306 Information Retrieval

Report Contents

Contents

Instructions for running our system

The first step into the 'Indexing' aspect of our complete system was to identify a document collection of our choice to index. Therefore, we decided to go for a more sophisticated approach by indexing a website instead of the sample datasets from Elasticsearch. The URL we decided to index derived from a BBC website. This is because it is from a trusted source and have a range of content within the webpage. The queries devised for the test collection will be more effective for analysing if indexing from an efficient website.

To complete the indexing of a website, a crawler will need to be employed that gathers the data to pass over to Elasticsearch. We have attempted various crawlers and tools to work with Elasticsearch. For instance, we tested on Nutch, GNU wget tool and tapping into an RSS feed using Elasticsearch's on logstash. However, these did not give the results we want and were some issues in the process of scrapping. Therefore, we used another alternative by using 'HTTrack' to scrap our documents. In combination with Elasticsearch, the document was able to be indexed. This can be seen by the structed index format as he results.

The next stage of the process for the system will be to provide 'Searching' options for the user. From the successful indexing of the website and datasets, the user must be able to search it. We have decided to take an enhanced approach and not use the command line for this. Instead, we have used 'Kibana' to provide an interactive system for the user. The choice to use Kibana is because it provides visualisation of the data in Elasticsearch, suggesting better analysis of data.

The following stage is to 'Build a Test Collection', in which ten unique queries are devised with their expected results. This leads to the final stage of our system is 'Evaluation', this includes exploring different search engine settings to discover what effect they have on the evaluation results. To accomplish this phase, we have identified a suitable metric. To provide the best approach for the evaluation results, we tried different retrieval models such as Boolean against TF.IDF. The numerical statistic of TFIDF provides us the reflection of how important a word is to a document in a collection or corpus. The value of tf-idf will increase proportionally to the number of times a word appears in the document. On the other hand, a Bool Query can be used that includes the use of AND/OR/NOT operators. This is used to fine tune the search queries to provide more relevant or specific results.

Description of the document collection, Test Queries & Expected results:

The document collection we have decided to use from Elasticsearch are the complete works of William Shakespeare. This is because the Shakespeare collection is suitably parsed into fields, in which it is organised in the following schema:

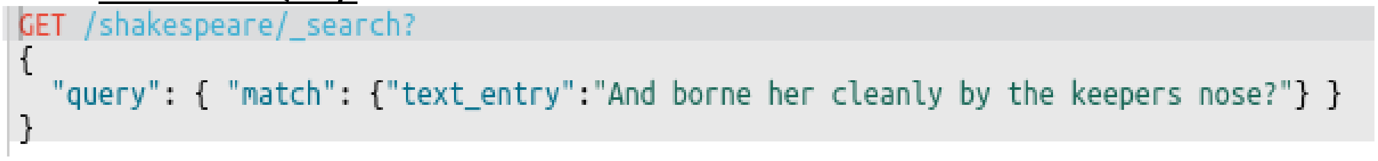
```
{
  "line_id": INT,
  "play_name": "String",
  "speech_number": INT,
  "line_number": "String",
  "speaker": "String",
  "text_entry": "String",
}
```

Mappings for the fields were required before loading the Shakespeare and logs data sets. This allow the documents in the index to be sectioned into logical groups.

We have developed a variety of test queries to be evaluated from the 'Kibana User Interface'. By using a range of test queries, it could fully test the efficiency and effectiveness of varying information retrieval systems.

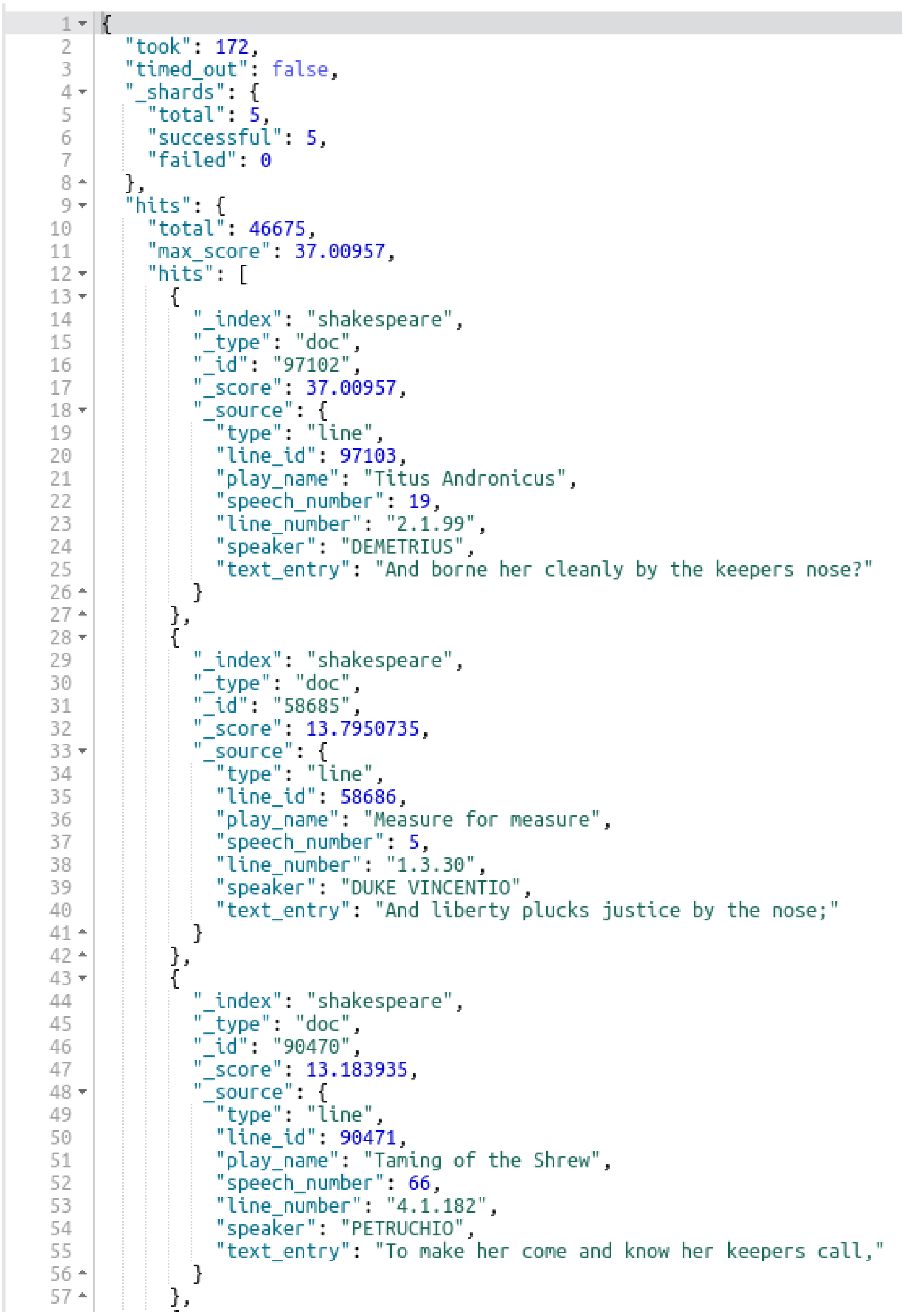
For instance, from the 10-test queries we incorporated:

- **Basic Match Query:**



```
GET /shakespeare/_search?
{
  "query": { "match": {"text_entry":"And borne her cleanly by the keepers nose?"} }
}
```

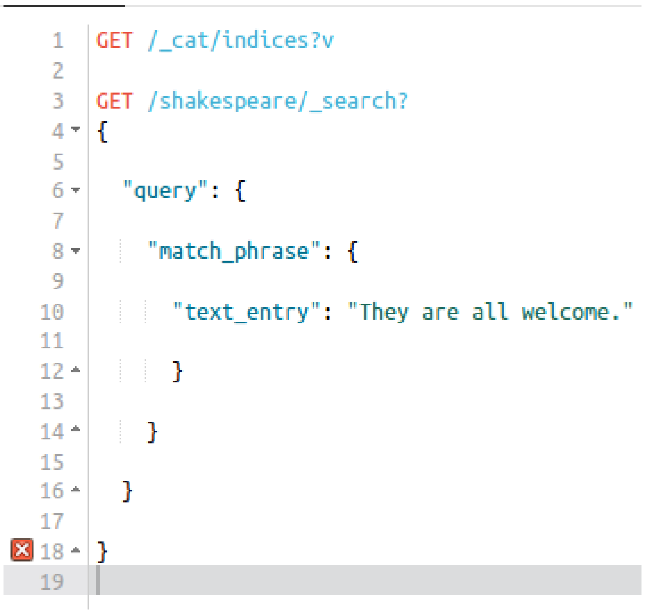
As the name suggests, a basic match query just searches the string that is provided in all the fields. In the test query below, if the text_entry field contains any of the values in ‘And borne her cleanly by the keepers’ then it will be displayed as the search results. The suggests the expected hits will be significantly high going to tens of thousands.



```
{
  "took": 172,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 46675,
    "max_score": 37.00957,
    "hits": [
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "97102",
        "_score": 37.00957,
        "_source": {
          "type": "line",
          "line_id": 97103,
          "play_name": "Titus Andronicus",
          "speech_number": 19,
          "line_number": "2.1.99",
          "speaker": "DEMETRIUS",
          "text_entry": "And borne her cleanly by the keepers nose?"
        }
      },
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "58685",
        "_score": 13.7950735,
        "_source": {
          "type": "line",
          "line_id": 58686,
          "play_name": "Measure for measure",
          "speech_number": 5,
          "line_number": "1.3.30",
          "speaker": "DUKE VINCENTIO",
          "text_entry": "And liberty plucks justice by the nose;"
        }
      },
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "90470",
        "_score": 13.183935,
        "_source": {
          "type": "line",
          "line_id": 90471,
          "play_name": "Taming of the Shrew",
          "speech_number": 66,
          "line_number": "4.1.182",
          "speaker": "PETRUCHIO",
          "text_entry": "To make her come and know her keepers call,"
        }
      },
```

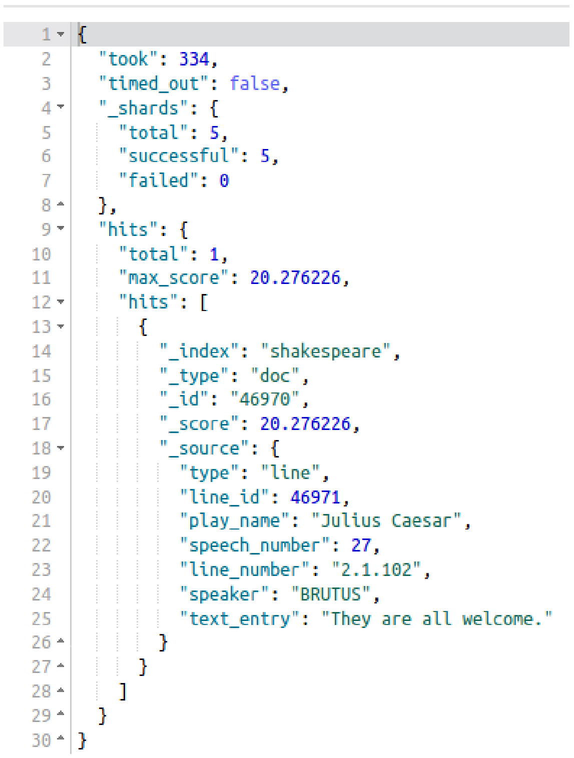
From the above test query, it shows there are 46675 hits which was what we expected to be tens of thousands. This test query is not a useful search as it allows far too much results.

- **Match Phrase Query:**



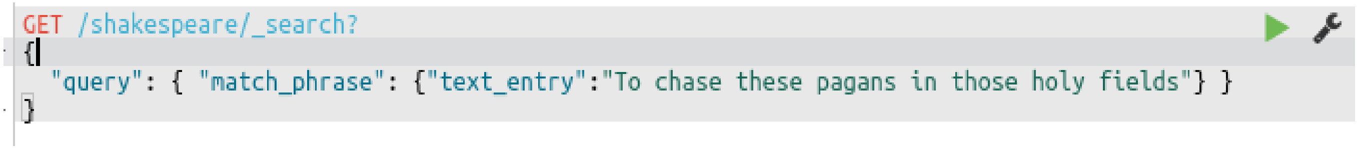
```
GET /_cat/indices?v

GET /shakespeare/_search?
{

  "query": {

    "match_phrase": {

      "text_entry": "They are all welcome."

      }

    }

  }

}

```

By using a 'Match Phrase' query, it will require that all the term in the query string be present in the document. As seen, in the above result the order specified in the query string needs to be exact and as close to each other. For instance, the match query we thought of was to search for text_entry of "They are all welcome". We expect for this query to return 1 hit that has the field 'text_entry' as 'They are all welcome'.



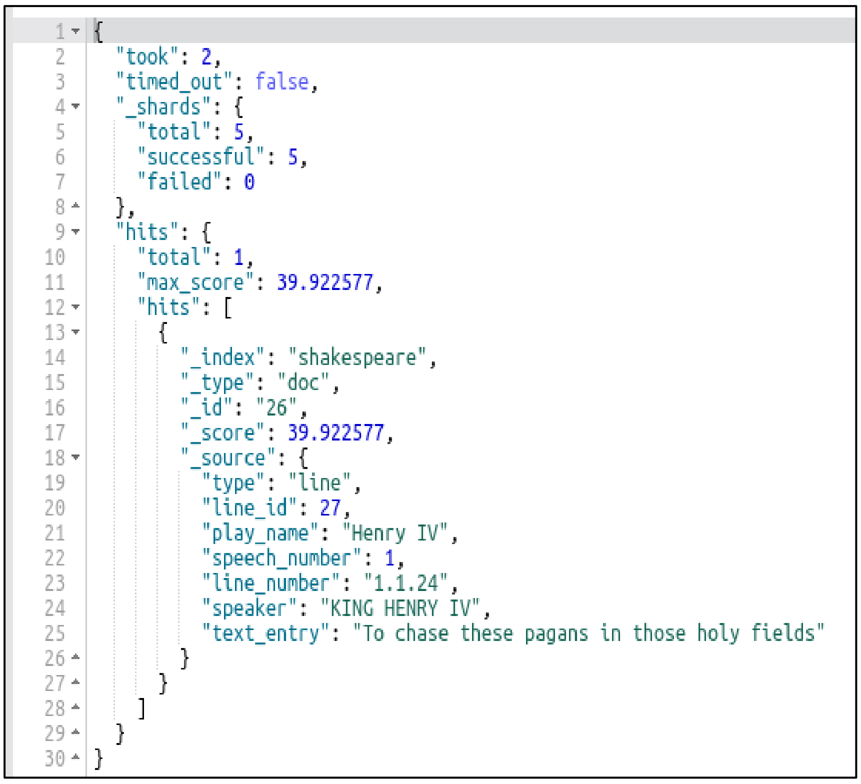
```
{
  "took": 334,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 1,
    "max_score": 20.276226,
    "hits": [
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "46970",
        "_score": 20.276226,
        "_source": {
          "type": "line",
          "line_id": 46971,
          "play_name": "Julius Caesar",
          "speech_number": 27,
          "line_number": "2.1.102",
          "speaker": "BRUTUS",
          "text_entry": "They are all welcome."
        }
      }
    ]
  }
}
```

From the above test query, it shows there are 1 hit which was what we expected to be less than 5 hits. This test query is a useful search as it allows narrowing down of search results.



```
GET /shakespeare/_search?
{
  "query": { "match_phrase": {"text_entry":"To chase these pagans in those holy fields"} }
}
```

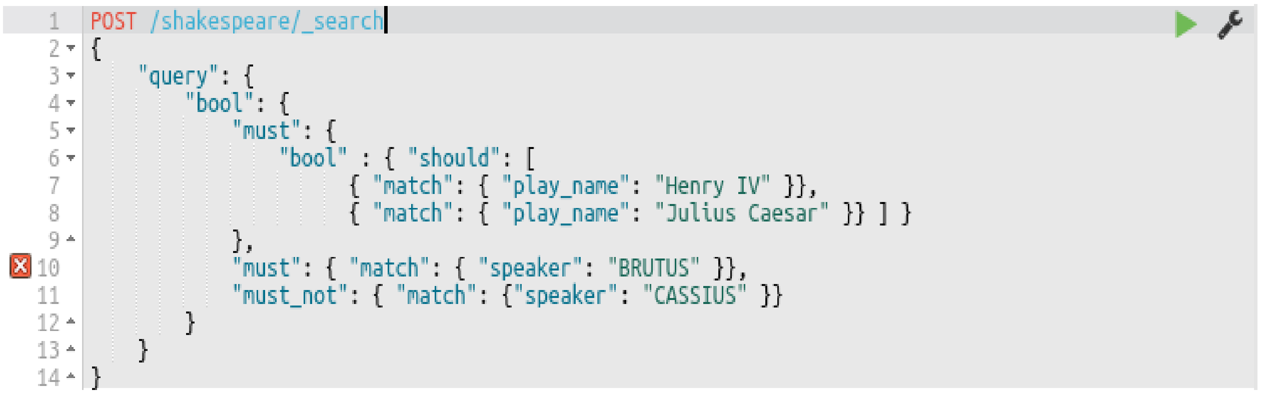
This is another type of test that makes use of 'Match Phrase' query. We expect the results in the text_entry field to contain 'To chase these pagans in those holy fields'. Since it requires to match exactly the full values of the field, the expected results will be limited to a few hits.



```
{
  "took": 2,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 1,
    "max_score": 39.922577,
    "hits": [
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "26",
        "_score": 39.922577,
        "_source": {
          "type": "line",
          "line_id": 27,
          "play_name": "Henry IV",
          "speech_number": 1,
          "line_number": "1.1.24",
          "speaker": "KING HENRY IV",
          "text_entry": "To chase these pagans in those holy fields"
        }
      }
    ]
  }
}
```

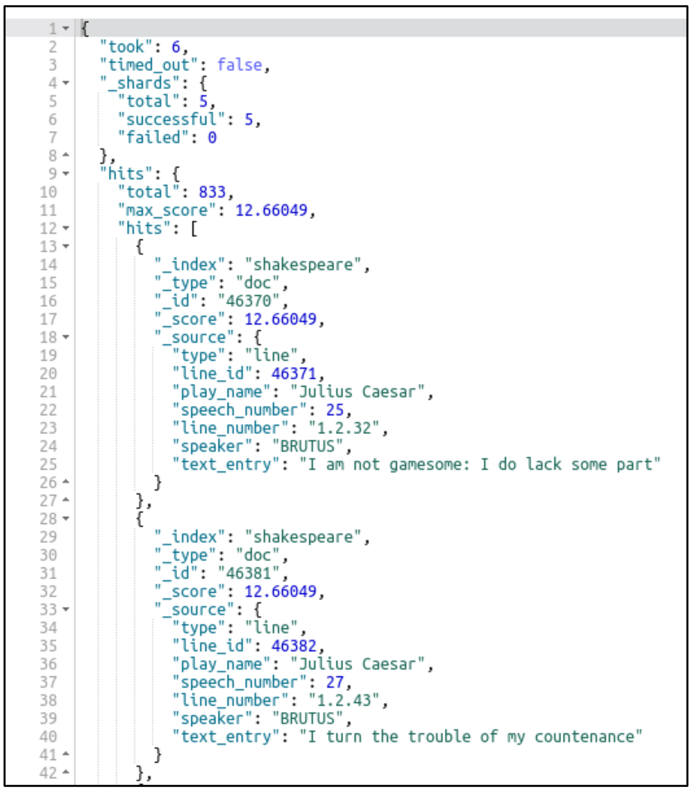
As expected, this test query provides just 1 hit due to it requiring specific values from the 'test_entry' field.

- **Bool Query:**



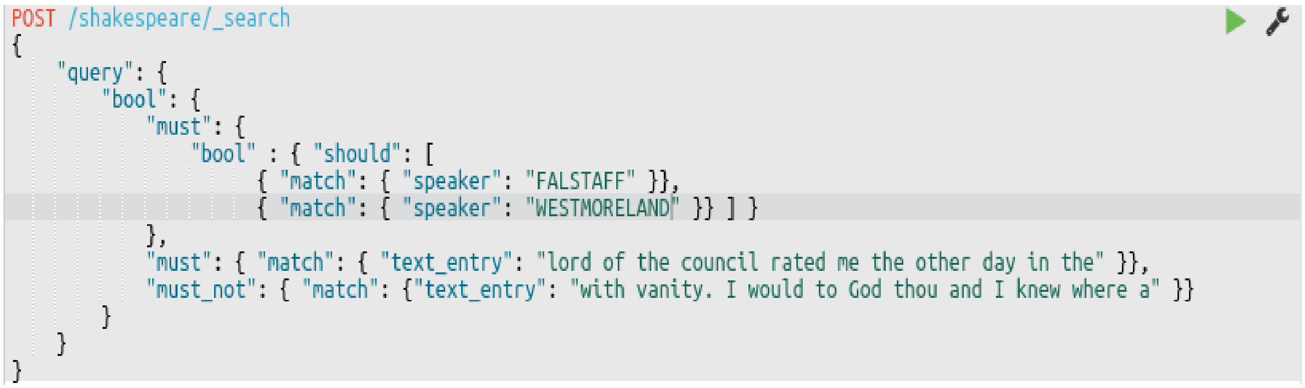
```
POST /shakespeare/_search
{
    "query": {
        "bool": {
            "must": {
                "bool" : { "should": [
                      { "match": { "play_name": "Henry IV" }},
                      { "match": { "play_name": "Julius Caesar" }} ] }
            },
            "must": { "match": { "speaker": "BRUTUS" }},
            "must_not": { "match": {"speaker": "CASSIUS" }}
        }
    }
}
```

The use of Bool Query includes AND/OR/NOT operators and can be used to enhance the efficiency and effectiveness of our search queries. The use of Bool Query provides more relevant or specific results. For example, in the above test query we expected result to contain either Henry IV or Julius Caesar for the 'play_name' field. Furthermore, the speaker field should be 'BRUTUS' and cannot be 'CASSIUS'. The amount of hits will be extensive since many fields contain Henry IV and Julius Caesar as the values.



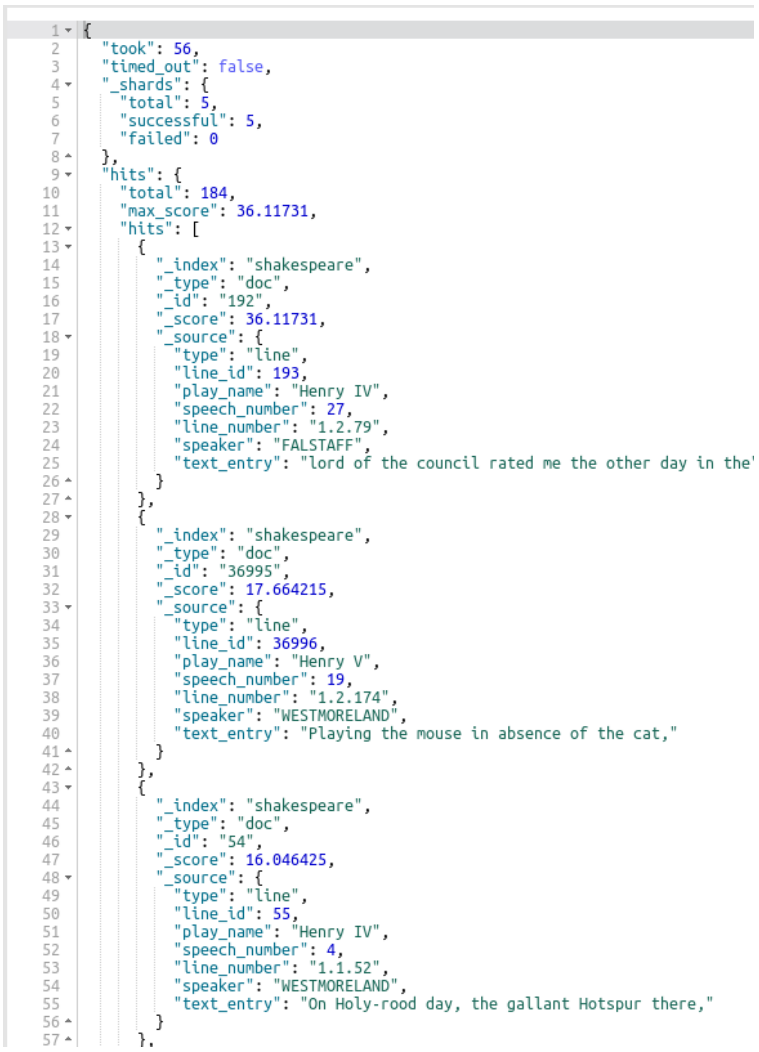
```
{
  "took": 6,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 833,
    "max_score": 12.66049,
    "hits": [
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "46370",
        "_score": 12.66049,
        "_source": {
          "type": "line",
          "line_id": 46371,
          "play_name": "Julius Caesar",
          "speech_number": 25,
          "line_number": "1.2.32",
          "speaker": "BRUTUS",
          "text_entry": "I am not gamesome: I do lack some part"
        }
      },
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "46381",
        "_score": 12.66049,
        "_source": {
          "type": "line",
          "line_id": 46382,
          "play_name": "Julius Caesar",
          "speech_number": 27,
          "line_number": "1.2.43",
          "speaker": "BRUTUS",
          "text_entry": "I turn the trouble of my countenance"
        }
      },
```

As expected, this test query provides many hits at 833 due to the broad range of Boolean values accepted.



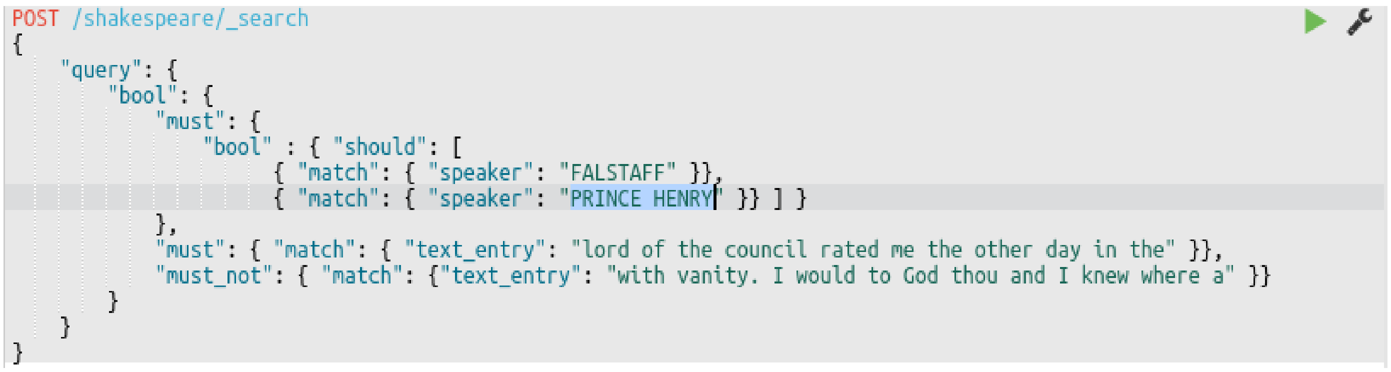
```
POST /shakespeare/_search
{
    "query": {
        "bool": {
            "must": {
                "bool" : { "should": [
                        { "match": { "speaker": "FALSTAFF" }},
                        { "match": { "speaker": "WESTMORELAND" }} ] }
            },
            "must": { "match": { "text_entry": "lord of the council rated me the other day in the" }},
            "must_not": { "match": {"text_entry": "with vanity. I would to God thou and I knew where a" }}
        }
    }
}
```

This is another form of a 'Bool Query', which makes use of the 'OR' operator. From this query we expect the results to show either ‘FALSTAFF’ or ‘WESTMORELAND’ from the speaker field. Since, it also requires the ‘text_entry’ field to match a specific value the total hits would be low at around 200 hits.



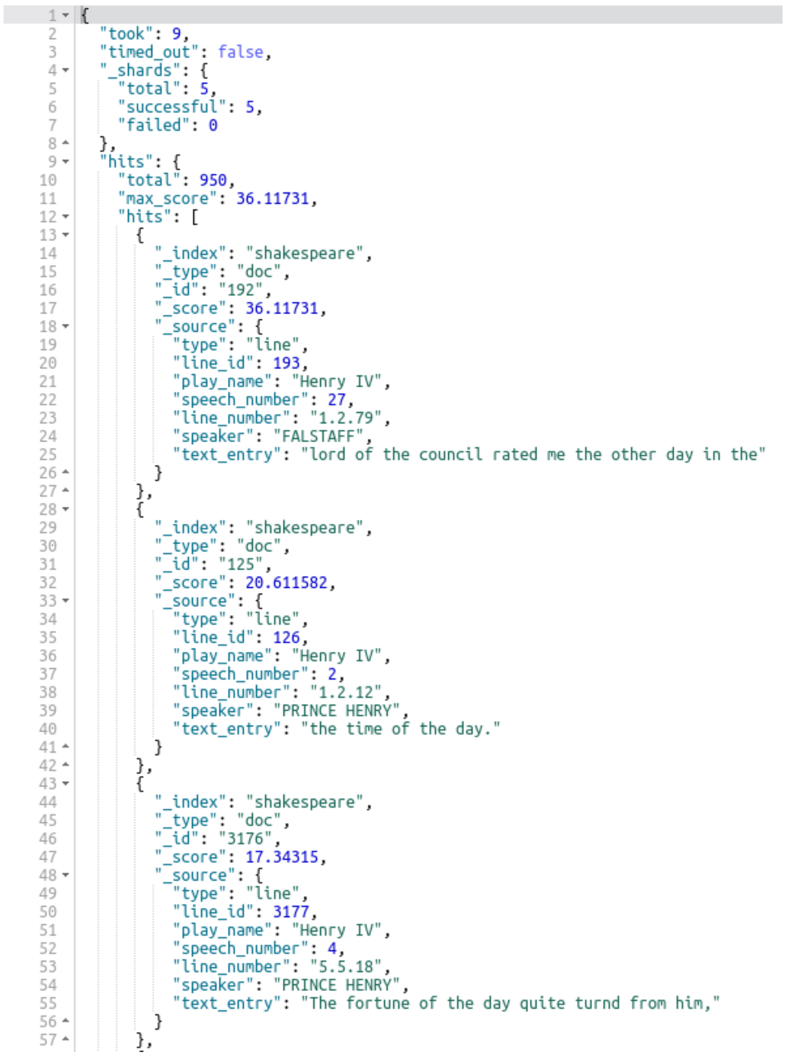
```
{
  "took": 56,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 184,
    "max_score": 36.11731,
    "hits": [
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "192",
        "_score": 36.11731,
        "_source": {
          "type": "line",
          "line_id": 193,
          "play_name": "Henry IV",
          "speech_number": 27,
          "line_number": "1.2.79",
          "speaker": "FALSTAFF",
          "text_entry": "lord of the council rated me the other day in the'
        }
      },
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "36995",
        "_score": 17.664215,
        "_source": {
          "type": "line",
          "line_id": 36996,
          "play_name": "Henry V",
          "speech_number": 19,
          "line_number": "1.2.174",
          "speaker": "WESTMORELAND",
          "text_entry": "Playing the mouse in absence of the cat,"
        }
      },
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "54",
        "_score": 16.046425,
        "_source": {
          "type": "line",
          "line_id": 55,
          "play_name": "Henry IV",
          "speech_number": 4,
          "line_number": "1.1.52",
          "speaker": "WESTMORELAND",
          "text_entry": "On Holy-rood day, the gallant Hotspur there,"
        }
      },
```

As expected, this test query provides not as much hits at 184 due to some specific Boolean values accepted.



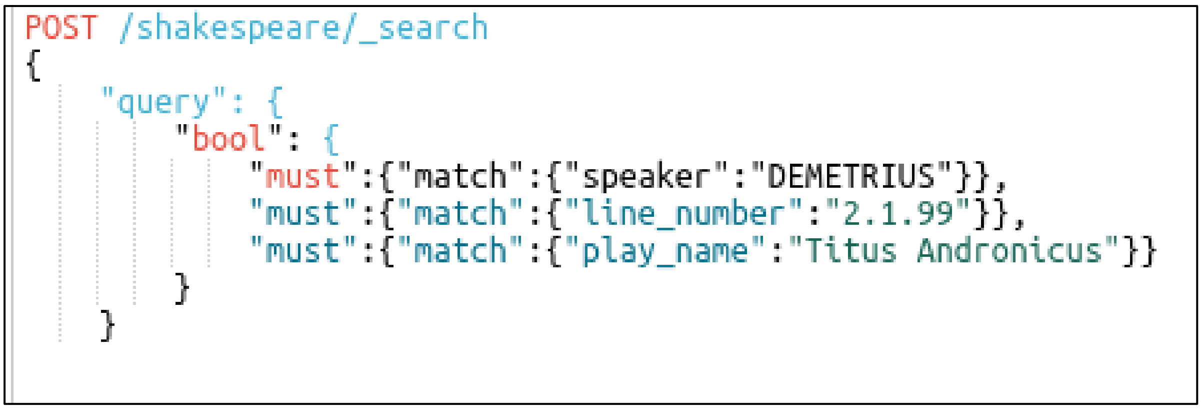
```
POST /shakespeare/_search
{
    "query": {
        "bool": {
            "must": {
                "bool" : { "should": [
                      { "match": { "speaker": "FALSTAFF" }},
                      { "match": { "speaker": "PRINCE HENRY" }} ] }
            },
            "must": { "match": { "text_entry": "lord of the council rated me the other day in the" }},
            "must_not": { "match": {"text_entry": "with vanity. I would to God thou and I knew where a" }}
        }
    }
}
```

The above test query is like the previous one, in which the results will display the field of speaker the values of either ‘FALSTAFF’ or ‘PRINCE HENRY’. The expected hits should be around 1000 as the values for the speaker field is quite common. Plus, the values for text_entry to match can be any characters which narrows down the search by just a little.

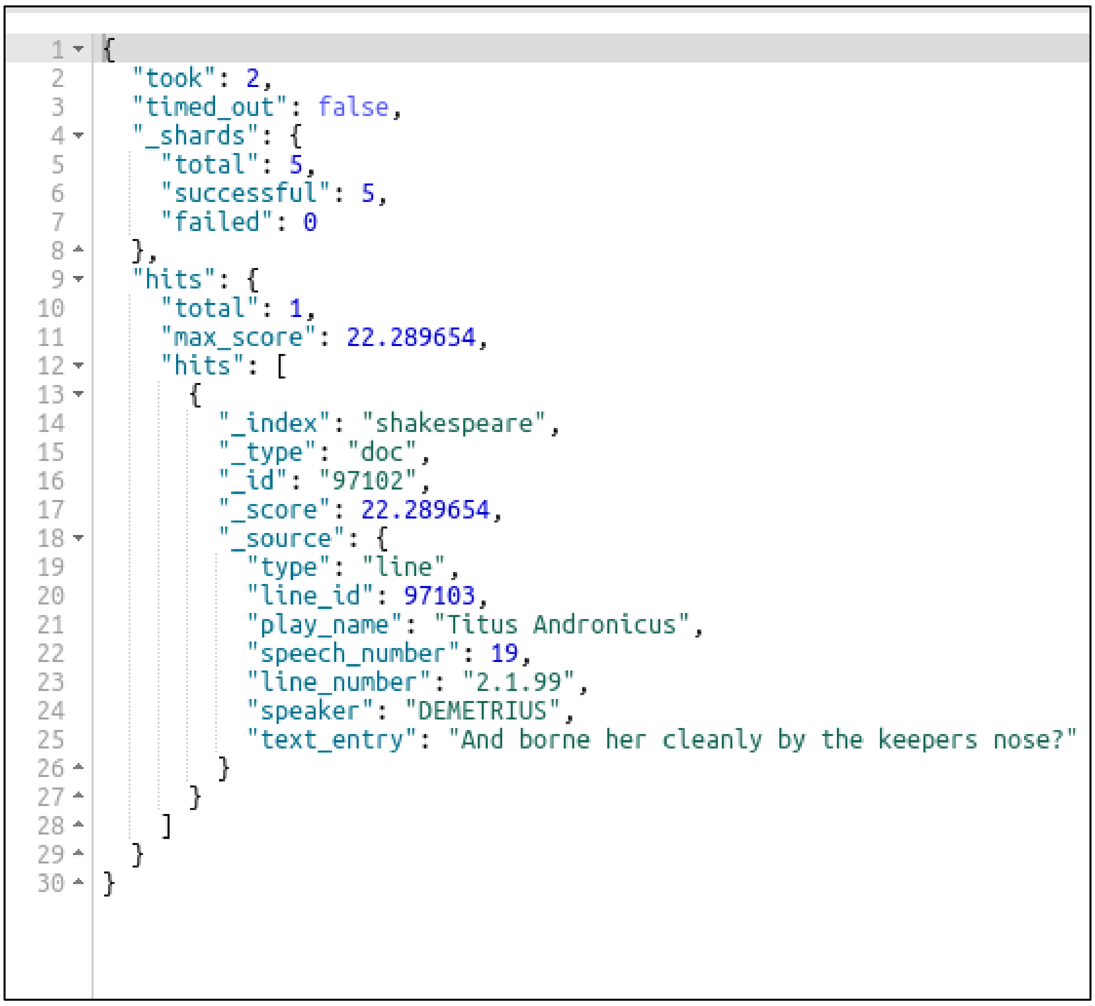


```
{
  "took": 9,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 950,
    "max_score": 36.11731,
    "hits": [
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "192",
        "_score": 36.11731,
        "_source": {
          "type": "line",
          "line_id": 193,
          "play_name": "Henry IV",
          "speech_number": 27,
          "line_number": "1.2.79",
          "speaker": "FALSTAFF",
          "text_entry": "lord of the council rated me the other day in the"
        }
      },
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "125",
        "_score": 20.611582,
        "_source": {
          "type": "line",
          "line_id": 126,
          "play_name": "Henry IV",
          "speech_number": 2,
          "line_number": "1.2.12",
          "speaker": "PRINCE HENRY",
          "text_entry": "the time of the day."
        }
      },
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "3176",
        "_score": 17.34315,
        "_source": {
          "type": "line",
          "line_id": 3177,
          "play_name": "Henry IV",
          "speech_number": 4,
          "line_number": "5.5.18",
          "speaker": "PRINCE HENRY",
          "text_entry": "The fortune of the day quite turnd from him,"
        }
      },
```

As expected, this test query provides around 1000 hits at 950 due to some common values used for searching.

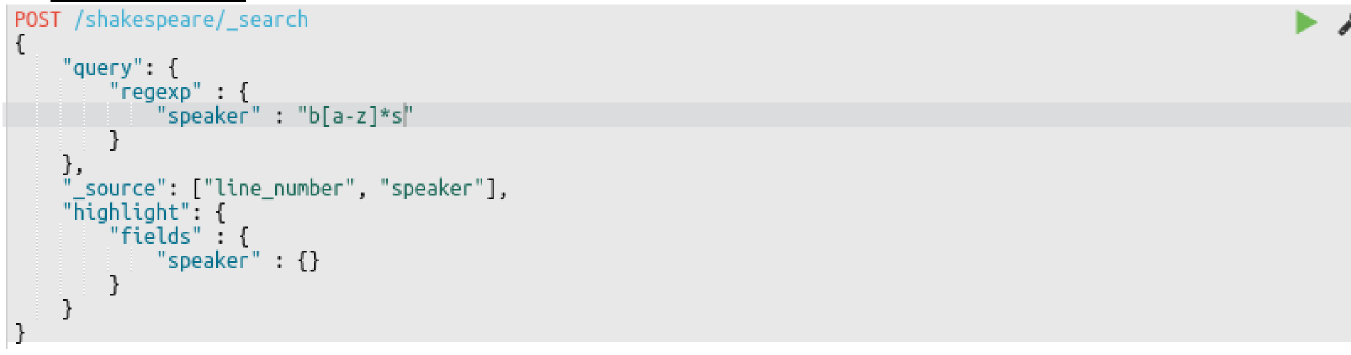

```
POST /shakespeare/_search
{
    "query": {
        "bool": {
            "must":{"match":{"speaker":"DEMETRIUS"}},
            "must":{"match":{"line_number":"2.1.99"}},
            "must":{"match":{"play_name":"Titus Andronicus"}}
        }
    }
```

From this test query, it is very specific in what values it must match. Therefore, it is expected the results to less than 5 hits or just 1. This query requires the fields of ‘speaker’, ‘line_number’ and ‘play_name’ to be exact values.


```
{
  "took": 2,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 1,
    "max_score": 22.289654,
    "hits": [
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "97102",
        "_score": 22.289654,
        "_source": {
          "type": "line",
          "line_id": 97103,
          "play_name": "Titus Andronicus",
          "speech_number": 19,
          "line_number": "2.1.99",
          "speaker": "DEMETRIUS",
          "text_entry": "And borne her cleanly by the keepers nose?"
        }
      }
    ]
  }
}
```

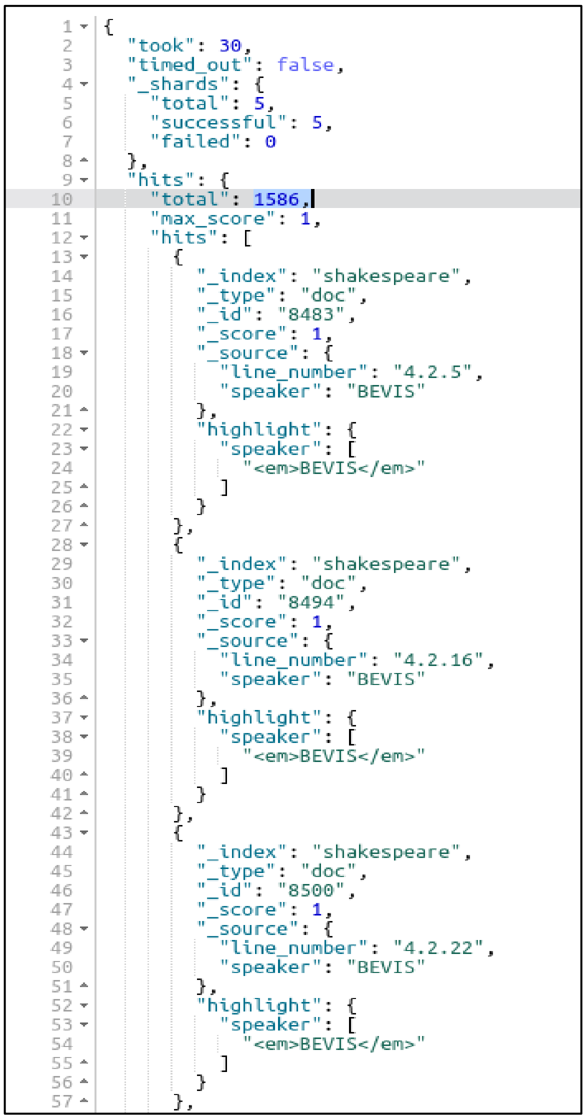
As expected, this test query provides very few hits with 1 only. This is due to it requiring very specific results as it must match certain values for the fields.

- <u>**Regexp Query:**</u>



```
POST /shakespeare/_search
{
    "query": {
        "regexp" : {
            "speaker" : "b[a-z]*s"
        }
    },
    "_source": ["line_number", "speaker"],
    "highlight": {
        "fields" : {
            "speaker" : {}
        }
    }
}
```

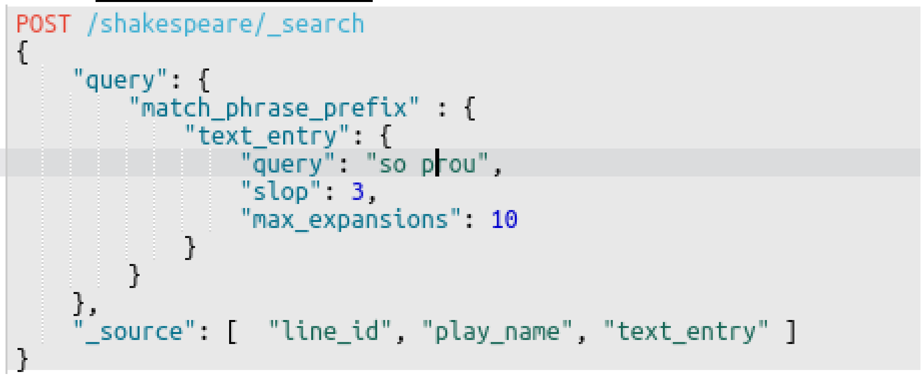
The use of Regexp queries ensures that more complex patterns are specified in comparison to wildcard queries. Instead of using '?' to specify the remaining letters of the query, it uses '[a-z]' to specify any characters that can occur between the starting letter of 'b' and the ending letter of 's'. Therefore, in this test query the expected result would be anything that has in the speaker field beginning with 'b' and ending in 's'. In this case the output should have in the speaker field 'BRUTUS'. This suggests hits will be around 1000 to 2000.



```
{
  "took": 30,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 1586,
    "max_score": 1,
    "hits": [
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "8483",
        "_score": 1,
        "_source": {
          "line_number": "4.2.5",
          "speaker": "BEVIS"
        },
        "highlight": {
          "speaker": [
            "<em>BEVIS</em>"
          ]
        }
      },
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "8494",
        "_score": 1,
        "_source": {
          "line_number": "4.2.16",
          "speaker": "BEVIS"
        },
        "highlight": {
          "speaker": [
            "<em>BEVIS</em>"
          ]
        }
      },
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "8500",
        "_score": 1,
        "_source": {
          "line_number": "4.2.22",
          "speaker": "BEVIS"
        },
        "highlight": {
          "speaker": [
            "<em>BEVIS</em>"
          ]
        }
      },
```

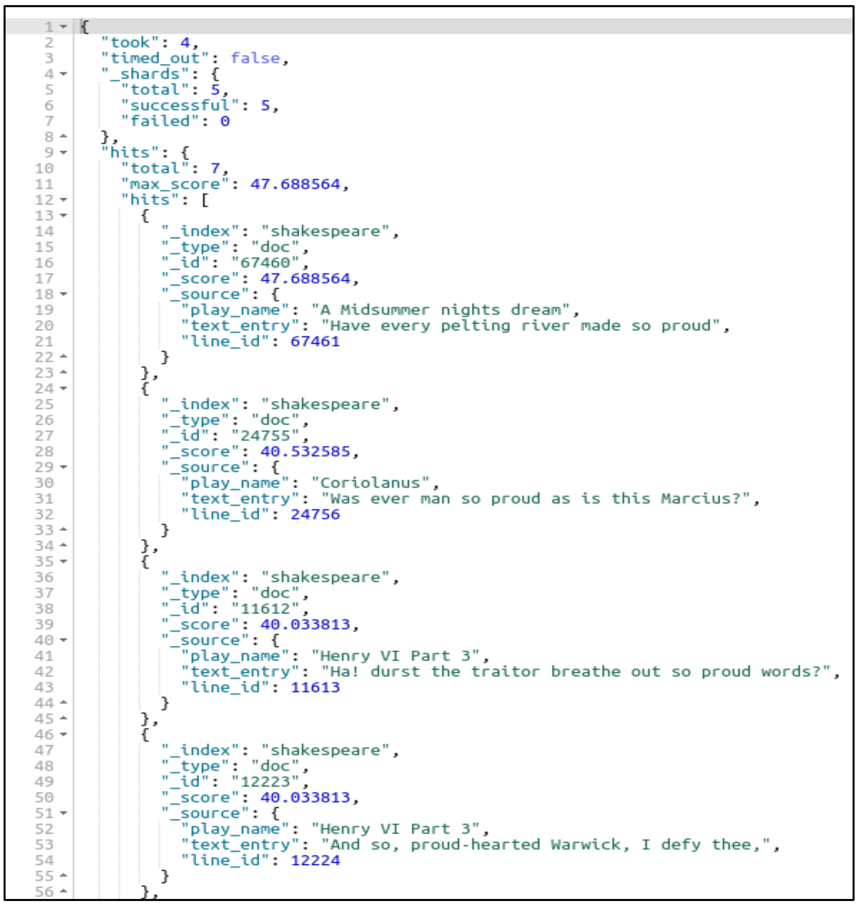
As expected, this test query provides hits into thousands category with 1586. This is due to the flexibility in the values that are allowed in the query.

- **Match Phrase Prefix:**



```
POST /shakespeare/_search
{
    "query": {
        "match_phrase_prefix" : {
            "text_entry": {
                "query": "so prou",
                "slop": 3,
                "max_expansions": 10
            }
        }
    },
    "_source": [  "line_id", "play_name", "text_entry" ]
}
```

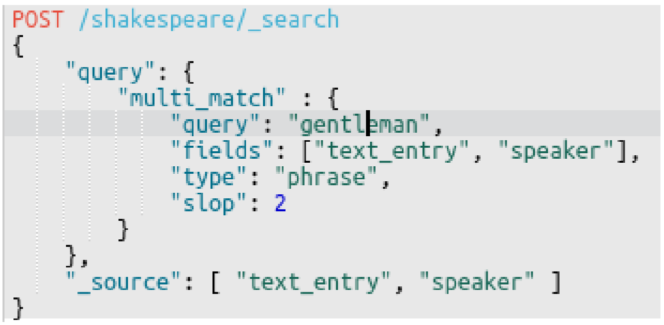
By using 'Match Phrase Prefix', the queries functions like an autocomplete system, in which the user can complete the search without inputting the whole query. In the test query above, it autocompletes 'prou' that is most likely to be 'proud'. It also makes use of 'max_expansions' that is set to 10. The idea of this is that the auto-complete field is limited 10 characters.



```
{
  "took": 4,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 7,
    "max_score": 47.688564,
    "hits": [
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "67460",
        "_score": 47.688564,
        "_source": {
          "play_name": "A Midsummer nights dream",
          "text_entry": "Have every pelting river made so proud",
          "line_id": 67461
        }
      },
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "24755",
        "_score": 40.532585,
        "_source": {
          "play_name": "Coriolanus",
          "text_entry": "Was ever man so proud as is this Marcius?",
          "line_id": 24756
        }
      },
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "11612",
        "_score": 40.033813,
        "_source": {
          "play_name": "Henry VI Part 3",
          "text_entry": "Ha! durst the traitor breathe out so proud words?",
          "line_id": 11613
        }
      },
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "12223",
        "_score": 40.033813,
        "_source": {
          "play_name": "Henry VI Part 3",
          "text_entry": "And so, proud-hearted Warwick, I defy thee,",
          "line_id": 12224
        }
      },
```

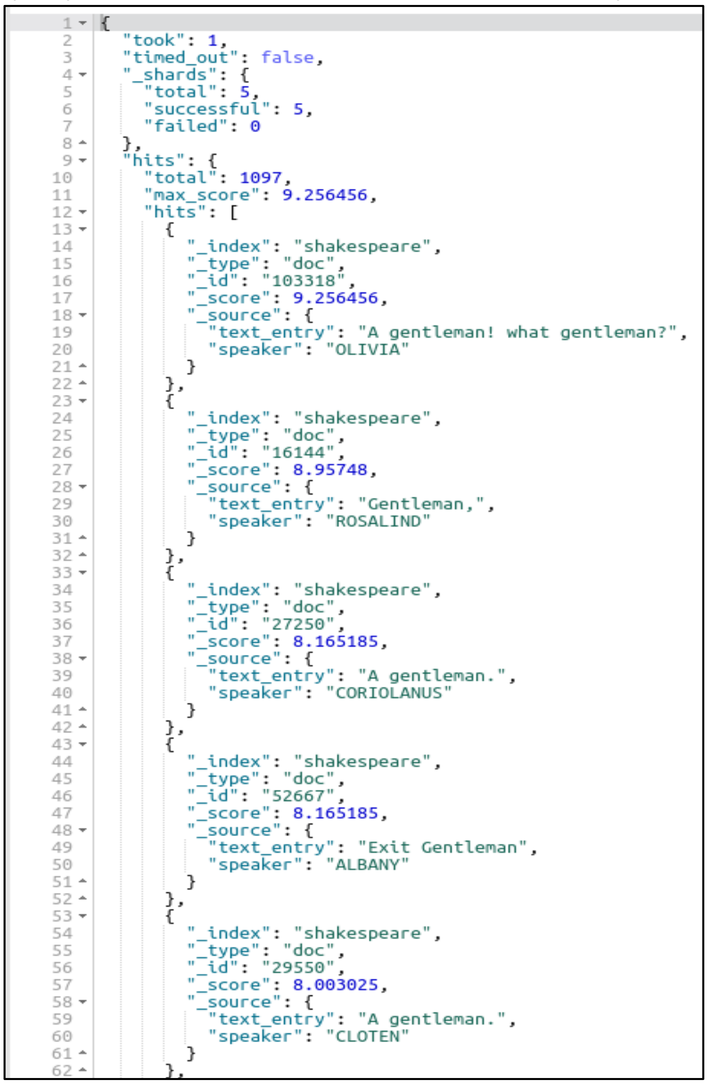
As expected, this test query provides few hits with only 7. This is due it autocompletes specific values of 'proud'.

- **Match Phrase Query:**



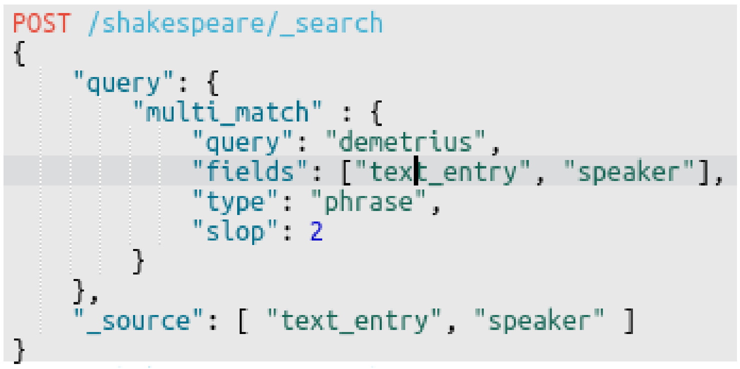
```
POST /shakespeare/_search
{
    "query": {
        "multi_match" : {
            "query": "gentleman",
            "fields": ["text_entry", "speaker"],
            "type": "phrase",
            "slop": 2
        }
    },
    "_source": [ "text_entry", "speaker" ]
}
```

For this test query, it makes use of match phrase query with the requirement of having 'gentleman' in either the fields of 'text_entry' or 'speaker'. It considers the order specified in the query string and needs to be close to each other. The use of the field of 'slop' indicates how far apart terms can be while still considering the match. The expected hits for this test query should be around 1000 hits as it's not a specific search.



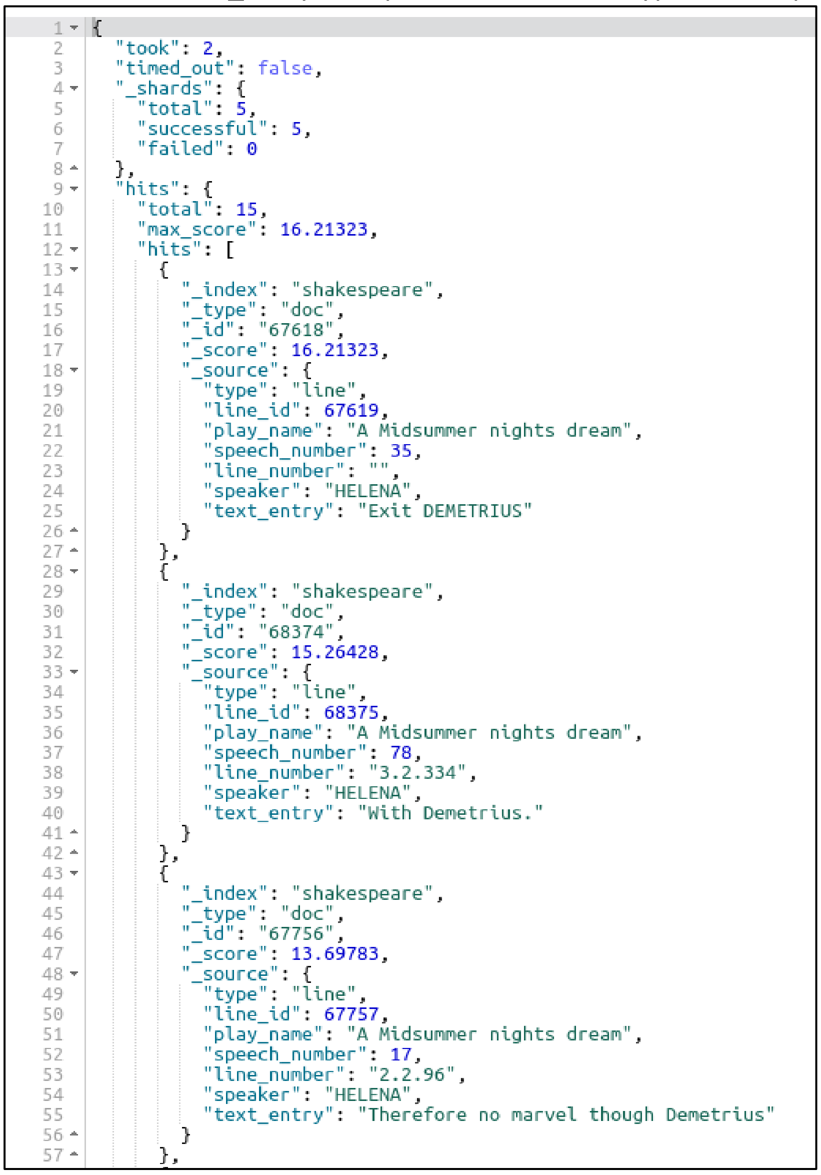
```
{
  "took": 1,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 1097,
    "max_score": 9.256456,
    "hits": [
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "103318",
        "_score": 9.256456,
        "_source": {
          "text_entry": "A gentleman! what gentleman?",
          "speaker": "OLIVIA"
        }
      },
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "16144",
        "_score": 8.95748,
        "_source": {
          "text_entry": "Gentleman,",
          "speaker": "ROSALIND"
        }
      },
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "27250",
        "_score": 8.165185,
        "_source": {
          "text_entry": "A gentleman.",
          "speaker": "CORIOLANUS"
        }
      },
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "52667",
        "_score": 8.165185,
        "_source": {
          "text_entry": "Exit Gentleman",
          "speaker": "ALBANY"
        }
      },
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "29550",
        "_score": 8.003025,
        "_source": {
          "text_entry": "A gentleman.",
          "speaker": "CLOTEN"
        }
      },
```

As expected, this test query provides around a thousand hits and it was 1097. This is due to the effectiveness of using multi match for the search.



```
POST /shakespeare/_search
{
    "query": {
        "multi_match" : {
            "query": "demetrius",
            "fields": ["text_entry", "speaker"],
            "type": "phrase",
            "slop": 2
        }
    },
    "_source": [ "text_entry", "speaker" ]
}
```

For this test query, it is like the previous one but this time it allows 'demetrius' to appear either in the 'text_entry' or 'speaker' fields. The type of this query is also set to 'phrase'.



```
{
  "took": 2,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 15,
    "max_score": 16.21323,
    "hits": [
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "67618",
        "_score": 16.21323,
        "_source": {
          "type": "line",
          "line_id": 67619,
          "play_name": "A Midsummer nights dream",
          "speech_number": 35,
          "line_number": "",
          "speaker": "HELENA",
          "text_entry": "Exit DEMETRIUS"
        }
      },
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "68374",
        "_score": 15.26428,
        "_source": {
          "type": "line",
          "line_id": 68375,
          "play_name": "A Midsummer nights dream",
          "speech_number": 78,
          "line_number": "3.2.334",
          "speaker": "HELENA",
          "text_entry": "With Demetrius."
        }
      },
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "67756",
        "_score": 13.69783,
        "_source": {
          "type": "line",
          "line_id": 67757,
          "play_name": "A Midsummer nights dream",
          "speech_number": 17,
          "line_number": "2.2.96",
          "speaker": "HELENA",
          "text_entry": "Therefore no marvel though Demetrius"
        }
      },
```

As expected, this test query provides few hits at 15. This is due to it requiring the specific value of 'demetrius' in the 'text_entry' field.

Design decisions of overall architecture

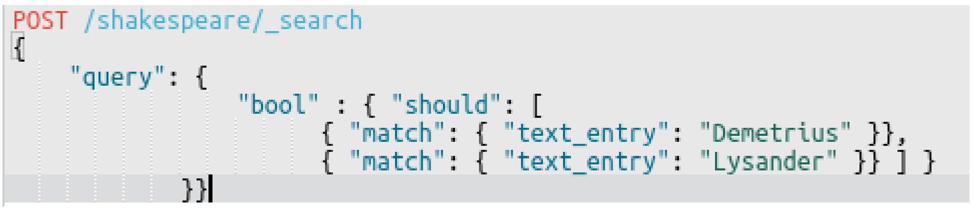
We used Kibana as our user interface because it provides a user-friendly interface and has great compatibility. Kibana provides advantages such as allowing visualisation of our Elasticsearch data and is easy efficient to use. It also has a dashboard that includes essential data such as the different requests running.

We also made the decision of using Elasticsearch because it is one of the most popular and available open source search engine.

How we attempted to Scrape and Crawl a website

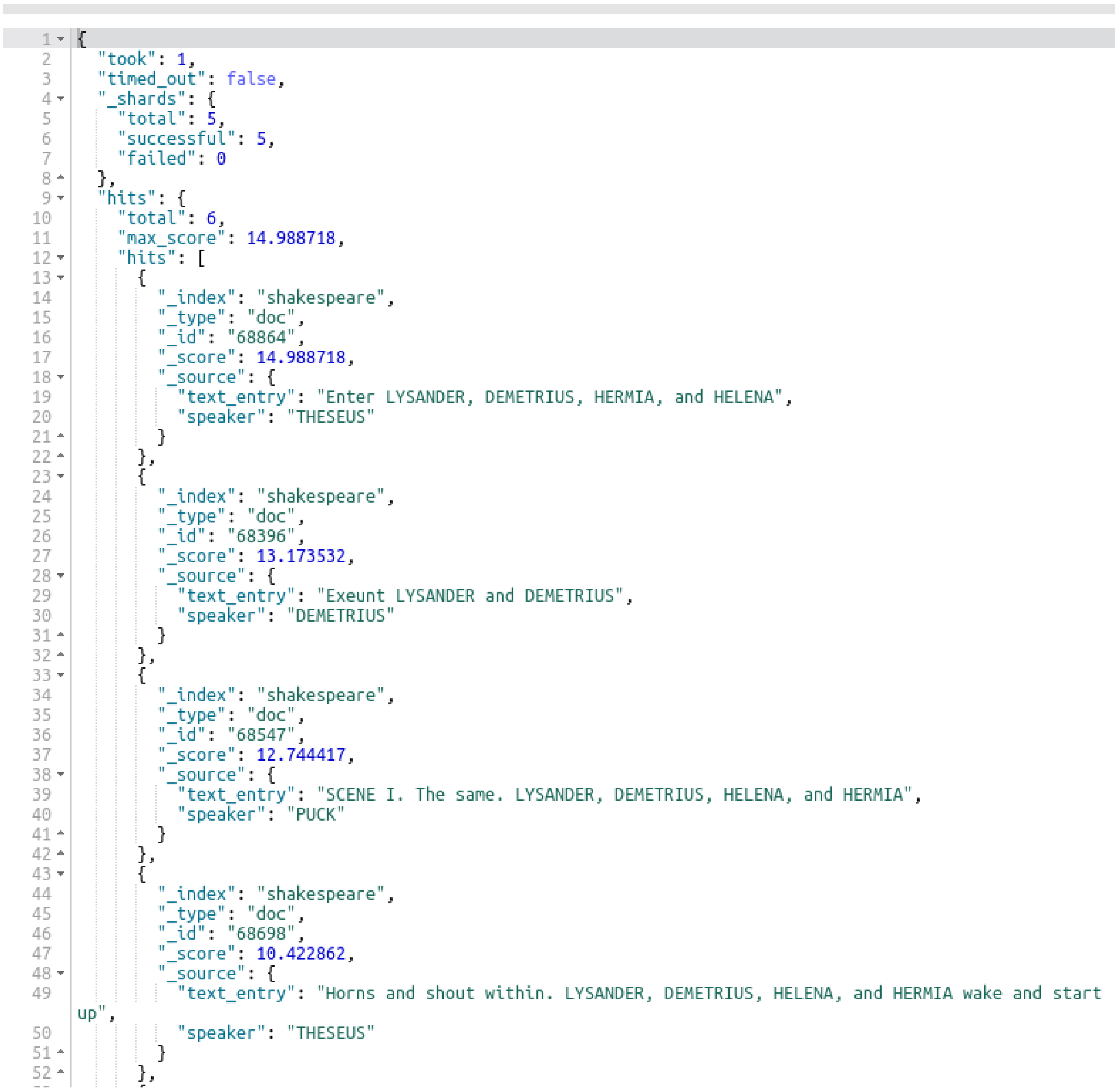
Scrapper: HTTrack
http://www.httrack.com/
Install current machine version from download page.
Visit: http://www.httrack.com/html/shelldoc.html for Windows or Linux system set up guide.
In our version we used Linux ubuntu distribution and were running from terminal window.
$ webttrack
Once that done open localhost:8080 in your web browser to see actual HTTrack web plugin and follow previously mentioned guide for set up.
After you extracted your website locate HTML file.
Visit GitHub at https://github.com/scraperwiki/elasticsearch-tool and download tar.gz
Extract it where you like, elasticsearch-tool has more functions, but as right now we only going to use httrack_html_to_json.py file.
Once you have HTML file from HTTrack and elasticsearch-tool folder from github, move or copy HTML file to elasticsearch-tool folder and run httrack_html_to_json.py in terminal window.
$ python httrack_html_to_json.py website.html
This will give JSON file format with Text only from website
JSON has few attributes like "title", "url", "scrape_date", and "body". "Body" is where all the text is held.
Having only these few attributes we decided that it would be insufficient to have such low number of attributes to have complex queries.

Evaluation results



```
POST /shakespeare/_search
{
    "query": {
            "bool" : { "should": [
                  { "match": { "text_entry": "Demetrius" }},
                  { "match": { "text_entry": "Lysander" }} ] }
        }}
```

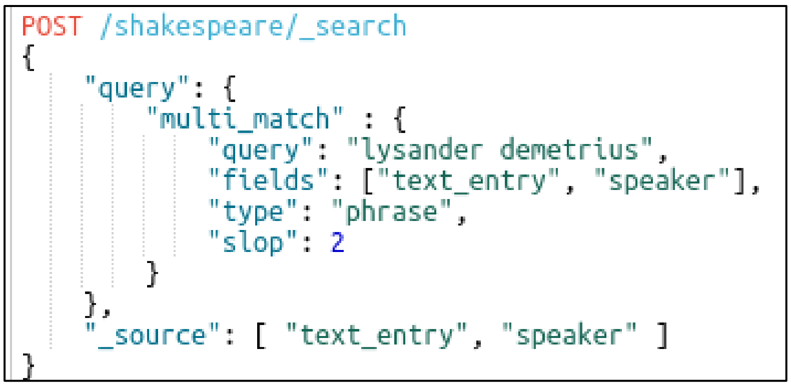
This is an example of a Boolean Query that searches 'Demetrius' and 'Lysander' as the values for the field of 'text_entry'. Both the values could appear as single text_entry or together. This suggests if they both appear in the same query then the score would be higher.



```
{
  "took": 1,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 6,
    "max_score": 14.988718,
    "hits": [
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "68864",
        "_score": 14.988718,
        "_source": {
          "text_entry": "Enter LYSANDER, DEMETRIUS, HERMIA, and HELENA",
          "speaker": "THESEUS"
        }
      },
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "68396",
        "_score": 13.173532,
        "_source": {
          "text_entry": "Exeunt LYSANDER and DEMETRIUS",
          "speaker": "DEMETRIUS"
        }
      },
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "68547",
        "_score": 12.744417,
        "_source": {
          "text_entry": "SCENE I. The same. LYSANDER, DEMETRIUS, HELENA, and HERMIA",
          "speaker": "PUCK"
        }
      },
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "68698",
        "_score": 10.422862,
        "_source": {
          "text_entry": "Horns and shout within. LYSANDER, DEMETRIUS, HELENA, and HERMIA wake and start up",
          "speaker": "THESEUS"
        }
      },
```

The above shows the results of Boolean query that have 6 hits. Based on this query there were relevant documents at position 1, 3 and 6 with our average precision at 0.722, with the other documents being irrelevant.

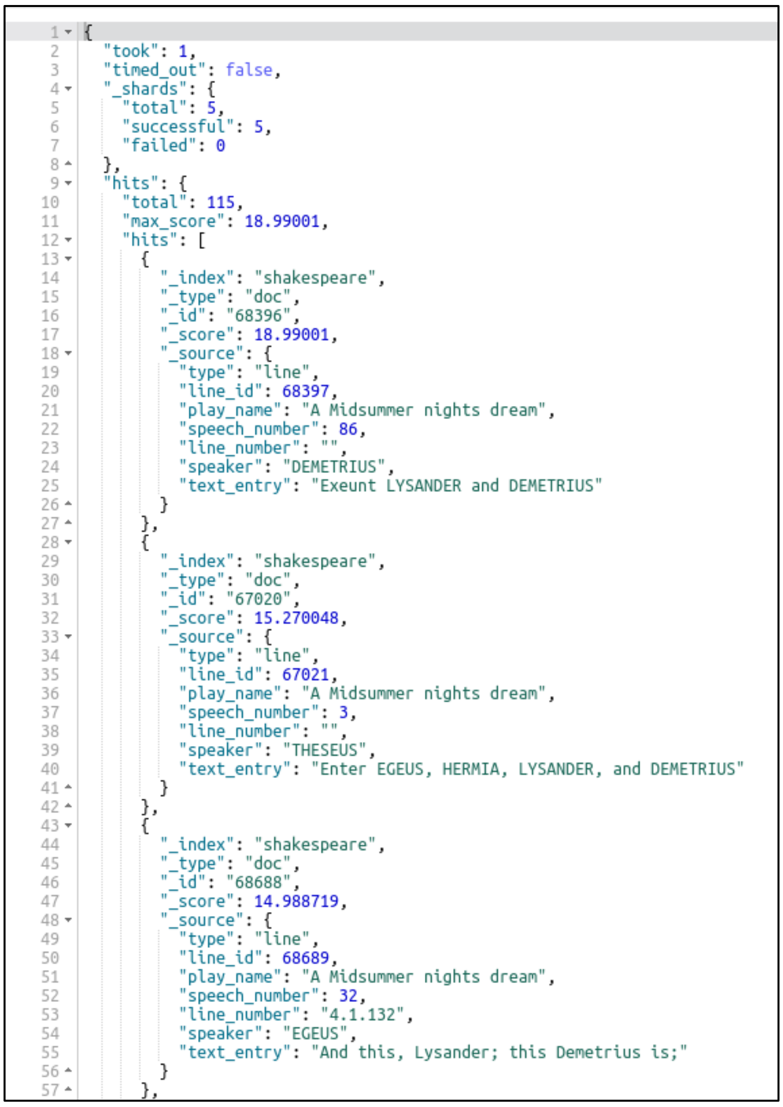
<u>TF.IDF search query</u>



```
POST /shakespeare/_search
{
    "query": {
        "multi_match" : {
            "query": "lysander demetrius",
            "fields": ["text_entry", "speaker"],
            "type": "phrase",
            "slop": 2
        }
    },
    "_source": [ "text_entry", "speaker" ]
}
```

This query searches if ‘lysander demetrius’ appears in the fields of ‘text_entry’ and ‘speaker’.

<u>The results for this specific query are as follows:</u>



```
{
  "took": 1,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 115,
    "max_score": 18.99001,
    "hits": [
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "68396",
        "_score": 18.99001,
        "_source": {
          "type": "line",
          "line_id": 68397,
          "play_name": "A Midsummer nights dream",
          "speech_number": 86,
          "line_number": "",
          "speaker": "DEMETRIUS",
          "text_entry": "Exeunt LYSANDER and DEMETRIUS"
        }
      },
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "67020",
        "_score": 15.270048,
        "_source": {
          "type": "line",
          "line_id": 67021,
          "play_name": "A Midsummer nights dream",
          "speech_number": 3,
          "line_number": "",
          "speaker": "THESEUS",
          "text_entry": "Enter EGEUS, HERMIA, LYSANDER, and DEMETRIUS"
        }
      },
      {
        "_index": "shakespeare",
        "_type": "doc",
        "_id": "68688",
        "_score": 14.988719,
        "_source": {
          "type": "line",
          "line_id": 68689,
          "play_name": "A Midsummer nights dream",
          "speech_number": 32,
          "line_number": "4.1.132",
          "speaker": "EGEUS",
          "text_entry": "And this, Lysander; this Demetrius is;"
        }
      },
```

The above shows the results of TF.IDF query that have 115 hits. Based on this query there were relevant documents at position 2 and 6 with our average precision at 0.416, with the other documents being irrelevant.

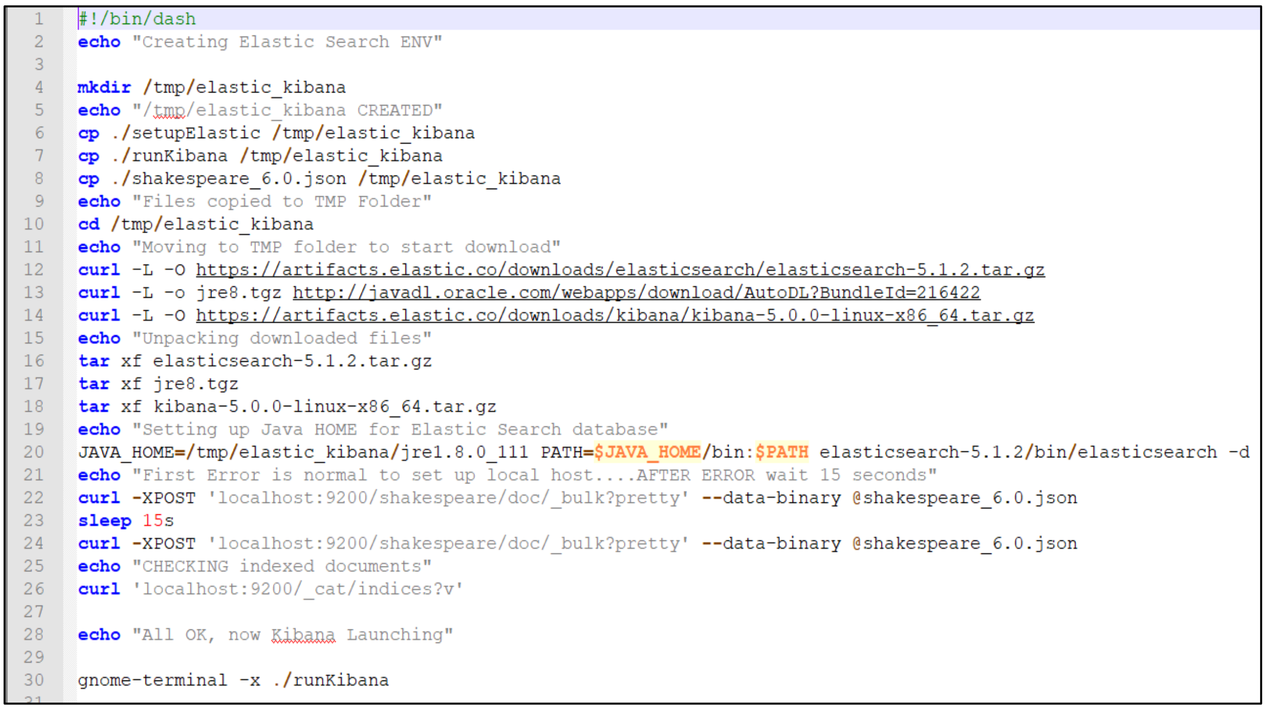
We decided to use Mean Average Precision (MAP), because it will allow us to compare between different information retrieval models. The chosen information retrieval models were TF.IDF and Boolean. Above search queries resulted in MAP of 0.569, which suggests it is quite accurate. The ideal result would be 1, but there will be no information retrieval model that are perfect.

Engineering a Complete System

Complete system is created in tmp folder on Ubuntu operating system for simplicity.

To run script open terminal window inside file existing folder and type $ ./setupElastic

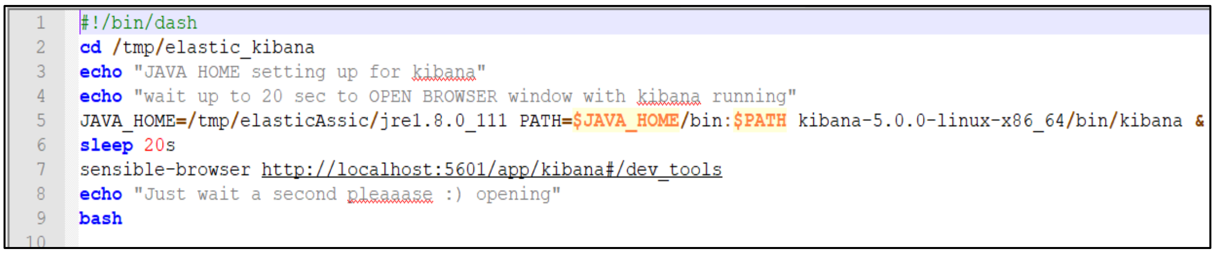
- Creating directory in tmp folder
- Copies all needed script and json dataset files
- Moves terminal to created folder at step 1
- Starts downloading all necessary files like, elasticsearch, kibana and java environment
- Next is unpacks everything in folder
- Java home is set for elastic search to use
- Posting indexed Json file into elastic search indices
- Checks if dataset is uploaded
- New terminal window is opened for kibana launch



```
#!/bin/dash
echo "Creating Elastic Search ENV"

mkdir /tmp/elastic_kibana
echo "/tmp/elastic_kibana CREATED"
cp ./setupElastic /tmp/elastic_kibana
cp ./runKibana /tmp/elastic_kibana
cp ./shakespeare_6.0.json /tmp/elastic_kibana
echo "Files copied to TMP Folder"
cd /tmp/elastic_kibana
echo "Moving to TMP folder to start download"
curl -L -O https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-5.1.2.tar.gz
curl -L -o jre8.tgz http://javadl.oracle.com/webapps/download/AutoDL?BundleId=216422
curl -L -O https://artifacts.elastic.co/downloads/kibana/kibana-5.0.0-linux-x86_64.tar.gz
echo "Unpacking downloaded files"
tar xf elasticsearch-5.1.2.tar.gz
tar xf jre8.tgz
tar xf kibana-5.0.0-linux-x86_64.tar.gz
echo "Setting up Java HOME for Elastic Search database"
JAVA_HOME=/tmp/elastic_kibana/jre1.8.0_111 PATH=$JAVA_HOME/bin:$PATH elasticsearch-5.1.2/bin/elasticsearch -d
echo "First Error is normal to set up local host....AFTER ERROR wait 15 seconds"
curl -XPOST 'localhost:9200/shakespeare/doc/_bulk?pretty' --data-binary @shakespeare_6.0.json
sleep 15s
curl -XPOST 'localhost:9200/shakespeare/doc/_bulk?pretty' --data-binary @shakespeare_6.0.json
echo "CHECKING indexed documents"
curl 'localhost:9200/_cat/indices?v'

echo "All OK, now Kibana Launching"

gnome-terminal -x ./runKibana
```

The screen capture below shows how Kibana is set up.

- Java Home environment is set for Kibana
- Wait 20 sec for default browser to open local host
- Start writing queries



```
#!/bin/dash
cd /tmp/elastic_kibana
echo "JAVA HOME setting up for kibana"
echo "wait up to 20 sec to OPEN BROWSER window with kibana running"
JAVA_HOME=/tmp/elasticAssic/jre1.8.0_111 PATH=$JAVA_HOME/bin:$PATH kibana-5.0.0-linux-x86_64/bin/kibana &
sleep 20s
sensible-browser http://localhost:5601/app/kibana#/dev_tools
echo "Just wait a second pleaaaase :) opening"
bash
```

Possible systems improvements & Extensions

The system can be improved with Logstash extensions for easy web scrapping integration into Elasticsearch. Enhancements can be made for ranked evaluation using the latest version of Elasticsearch 6.2. in addition, we could incorporate more search queries with different parameters to produce more accurate results score.

Possible improvements for the search queries could be having more alternative metrics to produce average precision that would be used to gather the MAP for overall system performance.