XOOR Follow
Making great ideas move forward— Visit us at http://xoor.io and contact for a free tech consultation about your next project!
Aug 3, 2017 · 15 min read

Indexing MongoDB with ElasticSearch

A Simple Autocomplete Index Project

About Full Text Search

Nowadays it's very common to have a search feature in any website or app. This usually happens with platforms that have lots of information to offer to their users. From e-commerce websites which have thousands of products in different categories, to blogs or news sites which have thousands of articles. Whenever a client/user/reader reaches this kind of websites, they automatically tend to find a search box where they can type a query to get to the specific article/product/whatever they're looking for. Having a bad search engine leads to frustrated users which will most probably never come back to our websites again.

Full text search powers all those search boxes you use daily in websites to find the stuff you look for. Whenever you want to find that batman phone case in the Amazon products database, or when you search for cats playing with laser lights videos on Youtube. Of course this huge websites rely on many other things that power up their search engines, but the base of all searches is full text indexes. That said, let's see what this post is about.

MongoDB Limitations

If you quickly do a google search for `MongoDB full text` you'll find in the MongoDB docs that full text search is supported. So why would we bother learning a new complex technology like Elastic Search, and why would we want to introduce a new complexity into our system architecture? Let's have a look at MongoDB text search support to find out the reasons.

I will assume you already have MongoDB installed and that you know the basics of it. If that's the case, then go ahead and open a console and run the `mongo` command to access the MongoDB console and create a database called `fulltext`.

```
$ mongo
$ use fulltext
  switched to db fulltext
```

Our test database will store articles, so let's add a collection which we'll call `articles`.

```
$ db.createCollection('articles')
  '{ "ok" : 1 }'
```

Now let's add a few documents that will be useful to test. We'll insert articles with a title and a paragraph as content. I've taken some paragraphs from two articles in the New York Times Dealbook.

Original article reference: Yahoo's Sale to Verizon Leaves Shareholders With Little Say

```
$ db.articles.insert({
  ... title: 'Yahoo sale to Verizon',
  ... content: 'The sale is being done in two steps. The
first step will be the transfer of any assets related to
Yahoo business to a singular subsidiary. This includes the
stock in the business subsidiaries that make up Yahoo that
are not already in the single subsidiary, as well as the odd
assets like benefit plan rights. This is what is being sold
to Verizon. A license of Yahoo's oldest patents is being
held back in the so-called Excalibur portfolio. This will
```

```
stay with Yahoo, as will Yahoo's stakes in Alibaba Group and
Yahoo Japan.'
  ... })
  WriteResult({ "nInserted" : 1 })
```

Original article reference: Chinese Group to Pay $4.4 Billion for Caesars' Mobile Games

```
$ db.articles.insert({
  ... title: 'Chinese Group to Pay $4.4 Billion for Caesars
Mobile Games',
  ... content: 'In the most recent example in a growing
trend of big deals for smartphone-based games, a consortium
of Chinese investors led by the game company Shanghai Giant
Network Technology said in a statement on Saturday that it
would pay $4.4 billion to Caesars Interactive Entertainment
for Playtika, its social and mobile games unit. Caesars
Interactive is controlled by the owners of Caesars Palace
and other casinos in Las Vegas and elsewhere.'
  ... })
  WriteResult({ "nInserted" : 1 })
```

Now that we have documents, we need to index them using a MongoDB text index. So let's create a text index in both the `title` and `content` fields of the `articles` collection:

```
$ db.articles.createIndex({
... title: 'text',
... content: 'text'
... })
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
```

Index created, now it's time to do some searches to see how that goes, let's see!

```
$ db.articles.find( { $text: { $search: "chinese" } } )
{ "_id" : ObjectId("579e0a35c6d02e54ad6fe556"), "title" :
```

```
"Chinese Group to Pay $4.4 Billion for Caesars Mobile
Games", "content" : "In the most recent example in a growing
trend of big deals for smartphone-based games, a consortium
of Chinese investors led by the game company Shanghai Giant
Network Technology said in a statement on Saturday that it
would pay $4.4 billion to Caesars Interactive Entertainment
for Playtika, its social and mobile games unit. Caesars
Interactive is controlled by the owners of Caesars Palace
and other casinos in Las Vegas and elsewhere." }
```

Good, seems it's working fine, we searched for the word `chinese` and it matched with the article about the Chinese group. Now let's make it a bit harder for MongoDB. Let's say we want to build an autocomplete input (one of those that recommend the user as he/she types on it). For this to work, I will assume that MongoDB will return the same article if I search for the word `chi` :

```
$ db.articles.find( { $text: { $search: "chi" } } )
```

Empty! This is one of the biggest limitations that MongoDB has on the full text search feature. The problem is that it indexes documents on the word level, so it's impossible by using a text index to do what it's called `partial matching` . This is, matching partial parts of a word.

At this point is when a more powerful text indexing platform is useful. In our case I've chosen Elastic Search, mainly because documentation is super helpful, and it provides out of the box a full set of RESTful API endpoints that makes it very easy to test.

. . .

ElasticSearch

What we're trying to do

I just wanted to note that this post is just a super little tiny simple example of what you can achieve with Elastic Search. There are books written on it, so I don't want you to think Elastic Search it's useful just to implement autocomplete inputs. I just find it as an easy to understand example of how Elastic might help doing complex searches that MongoDB can't provide us.

The secondary purpose of the post is to show how you can import your existing MongoDB documents into full text indexed documents in ElasticSearch. Again, the autocomplete example is small enough to be explained in one post for this too. If you find the text indexing world interesting, please go ahead and read more about ElasticSearch (`ES` from now on) and the huge set of features it has.

I'm not going to explain here how to install ES since the process it's quite simple. Since ES is built on Java, just make sure you have Java installed and the `JAVA_HOME` variable set.

Once you have ES installed, this is the overall process we'll follow:

1. Create the index for our documents.
2. Import our MongoDB collection into ES with a tool called `mongo-connector` .
3. Migrate the index created by `mongo-connector` in ES to the index we created in step 1.
4. Try out our new index and see how documents are indexed all the time while we keep the `mongo-connector` running.

Creating the ES index

So… how do we create an index that performs better than the built in MongoDB text index? What do we need to configure in ES? We'll have to define what ES calls the `Analysis Chain` . This is simply put, the pipeline through which each of the documents we insert into the index will go through in order to be indexed.

An analysis chain is formed by analysers. Analysers are filters that take the document, analyse and modify it and pass it to the next one. For example there might be an analyser to remove the so called stop words, which are very common words that do not provide any useful information for indexing, like `the` or `and` .

Analysers are composed by three functions: a `character filter` , a `tokenizer` and a `token filter` . The first one is in charge of cleaning up the string before it's tokenized, for example by striping HTML tags. The second one is the responsible for splitting it into terms, for example by splitting the string by spaces. The last one's job is to modify terms to

optimize the index purpose, for example by removing stop words or lowercasing all the terms.

ES provides different analysers which serve as a starting point for creating custom analysers that suit better to any index needs. One of the alternatives provided by ES is called `edge_ngrams` analyser. To understand what edge n-grams are, we first need to understand what n-grams are. As the n-gram wikipedia page points out:

> **an n-gram is a contiguous sequence of n items from a given sequence of text or speech**

So let's say you have the word `blueberry`, then the `1-grams` or `unigrams` will be:

```
[b, l, u, b, e, r, r, y]
```

Increasing `n` by 1, we get the `bigrams` of blueberry:

```
[bl, lu, ub, be, er, rr, ry]
```

And I guess you know how to build the list of `trigrams` and `4-grams` and so on…

Now we can see what `edge n-grams` are, and according to the ES documentation:

> ***Edge n-grams are anchored to the beginning of the word***

Which means that for `blueberry`, the edge n-grams will be:

```
[b, bl, blu, blue, blueb, bluebe, blueber, blueberr, blueberry]
```

See where are we going with this? If you have the word `blueberry` indexed with it's edge n-grams, you can easily create an autocomplete search module. Because if user types `b`, it will match, if the user types `bl` it will match, if the user types `bla` it won't match anymore and the autocomplete option would disappear.

So this edge n-gram thing should be definitely part of our index, and this is how we'll define it:

```
{
    "filter": {
        "autocomplete_filter": {
            "type":     "edge_ngram",
            "min_gram": 3,
            "max_gram": 20
        }
    }
}
```

So with this json object we're defining a token filter (`filter`) called "autocomplete_filter". And we're saying that it will be an `edge_ngram` filter which will have from 3-grams up to 20-grams. The reason I used 3 as minimum is because for very big databases, having unigrams would slow down the performance a lot, since lots of documents would match the search. That's why many websites that have autocomplete function ask users to type at least three characters until they can suggest alternatives.

Now that we have our token filter defined, we need to define our custom analyser:

```
{
    "analyzer": {
        "autocomplete": {
            "type":      "custom",
            "tokenizer": "standard",
            "filter": [
                "lowercase",
                "autocomplete_filter"
            ]
        }
    }
}
```

Here we define a custom `analyzer` called "autocomplete", we tell ES that it will be a custom analyser, that will use the `standard` tokeniser and we set two filtering steps: `lowercase` (which is self-explanatory) and after that we set our custom `autocomplete_filter` .

Now that we defined the filter and the analyser, let's create the index. Grab a console and execute the following `curl` command:

```
$ curl -H 'Content-Type: application/json' \
       -X PUT http://localhost:9200/fulltext_opt \
       -d \
      "{ \
          \"settings\": { \
              \"number_of_shards\": 1, \
              \"analysis\": { \
                  \"filter\": { \
                      \"autocomplete_filter\": { \
                          \"type\":     \"edge_ngram\", \
                          \"min_gram\": 3, \
                          \"max_gram\": 20 \
                      } \
                  }, \
                  \"analyzer\": { \
                      \"autocomplete\": { \
                          \"type\":      \"custom\", \
                          \"tokenizer\": \"standard\", \
                          \"filter\": [ \
                              \"lowercase\", \
                              \"autocomplete_filter\" \
                          ] \
                      } \
                  } \
              } \
          } \
      }"
{"acknowledged":true}
```

The `fulltext_opt` in the endpoint URL tells ES to create a new index named like that. The reason I chose that name is because our MongoDB collection is named `fulltext`, and when we import it the first time to ES a `fulltext` index will be created automatically. We'll later move all the documents from `fulltext` to the optimized `fulltext_opt` index.

The last thing we have to do in our `fulltext_opt` index is create the mappings. Mappings are just groups of documents. We'll create a mapping called `articles` and we'll define the property `title` and `content` on it:

```
$ curl -H 'Content-Type: application/json' \
        -X PUT
http://localhost:9200/fulltext_opt/_mapping/articles \
        -d \
       "{ \
           \"articles\": { \
               \"properties\": { \
                   \"title\": { \
                       \"type\":     \"string\", \
```

```
                        \"analyzer\": \"autocomplete\" \
                    }, \
                    \"content\": { \
                        \"type\":    \"string\" \
                    } \
                } \
            } \
        }"
{"acknowledged":true}
```

You can see that we used our `autocomplete` analyser for the `title` property only. Since we're supposedly using this for an autocomplete function it makes no sense to index the article content (unless you'd like to suggest article content to the user… which would be weird).

The `acknowledged: true` response means our index was successfully created and the mappings added. Now it's time to import the documents from our MongoDB into it.

Importing from MongoDB into ES

To import our documents I could simply insert them manually into our ES index (I have only two documents in my collection of articles. The problem is that in real life we want to keep both MongoDB and our index synchronized, so that anytime a new document is inserted, the same document will be indexed in ES.

Fortunately for us, there's a tool called `mongo-connector` that does what we need. And even better, it has support for Elastic Search. I'm not going to dive too deep in the mongo-connector. You can find a lot of details about how it works on the previous link. Let's just stick with the idea that it will consume documents from our MongoDB and put them in our ES index.

You can install the mongo-connector using the Python package manager `pip`. You'll need to install the `elastic2-doc-manager` which will provide the support to copy stuff from MongoDB into ElasticSearch 2.X.

```
$ pip install mongo-connector
$ pip install elastic2-doc-manager
```

The next step is to start our MongoDB server as a replica set. I'm not going deep with this as well, if you don't know what replica sets are in MongoDB feed yourself here :). To run MongoDB as a replica set just pass the `--replSet` option when starting it and give the replica set a name (`rs0` in this case):

```
# Set the dbpath to wherever you have your data $ mongod --
dbpath /data/mongodb/db/ --replSet rs0
```

You'll probably see some message like this one:

```
016-07-30T16:17:45.881+0900 [rsStart] replSet can't get
local.system.replset config from self or any seed
(EMPTYCONFIG) 2016-07-30T16:17:45.881+0900 [rsStart] replSet
info you may need to run replSetInitiate -- rs.initiate() in
the shell -- if that is not already done
```

All you have to do is obey and open the mongo shell, and run `rs.initiate()` . It's possible that you might see this error message when trying to initiate the replica set:

```
$ rs.initiate()
 {
     "info2" : "no configuration explicitly specified --
making one",
     "me" : "mbp-mauricio:27017",
     "ok" : 0,
     "errmsg" : "couldn't initiate : can't find self in the
replset config"
 }
```

The problem is that the replica can't find the machine with name `mbp-mauricio` in this case. All you have to do is go to your `/etc/hosts` file and add an entry:

```
127.0.0.1  [your-machine-name]
```

MongoDB is up and running, now let's start ES. Go into your ES installation directory and run:

```
$ ./bin/elastic
```

All set, time to run the mongo-connector.

```
$ mongo-connector -m 127.0.0.1:27017 -t 127.0.0.1:9200 -d
elastic2_doc_manager
```

You can replace the parameters with your custom data, this is just the default localhost implementation of it. So here we basically tell mongo-connector to consume MongoDB data from `localhost:27017` and send it to the ES instance running on `localhost:9200`. All this will be done by using the `elastic2_doc_manager`. After a while (depending on how many MongoDB databases you have and how big they are), you should be able to see the new indexes in your ES instance. In my case it was almost instant, since I had only two documents in my `fulltext` database. So if you call the corresponding ES endpoint to list indices, you should see this:

```
$ curl localhost:9200/_cat/indices?v
health status index                pri rep docs.count
docs.deleted store.size pri.store.size
yellow open   fulltext               5   1          2
0     10.9kb         10.9kb
yellow open   fulltext_opt           1   1          0
0       159b           159b
```

You might have more entries if you had other databases in your MongoDB instance. The good thing of mongo-connector is that it's super configurable, so you can tell it which collections from which databases you want to import. More on that here.

Moving documents between indices

So we have now two indices, one created by mongo-connector which is not optimized and has our two documents, and another one optimized but empty. All we have to do now is copy the documents between

indices. And again, for me it would be simpler to just insert them manually since I have just two documents, but real world applications have thousands to millions of documents.

There is a great tool for this purpose called `elasticdump` which makes this task extremely easy. You can install it via NPM:

```
$ npm install -g elasticdump
```

With elasticdump you can import analysers, mappings and data from one ES index into another (or even into a json file). In our case we don't care about analysers and mappings, we'll just import the data since the analyser and mappings are already defined in our `fulltext_opt` index.

```
$ elasticdump \
   --input=http://localhost:9200/fulltext \
   --output=http://localhost:9200/fulltext_opt
Mon, 01 Aug 2016 01:21:10 GMT | starting dump
Mon, 01 Aug 2016 01:21:10 GMT | got 2 objects from source
elasticsearch (offset: 0)
Mon, 01 Aug 2016 01:21:10 GMT | sent 2 objects to
destination elasticsearch, wrote 2
Mon, 01 Aug 2016 01:21:10 GMT | got 0 objects from source
elasticsearch (offset: 2)
Mon, 01 Aug 2016 01:21:10 GMT | Total Writes: 2
Mon, 01 Aug 2016 01:21:10 GMT | dump complete
```

Now if you run again the indices query in ES you should see that `docs.count` for the fulltext_opt index has been changed to 2 instead of 0:

```
$ curl localhost:9200/_cat/indices?v health status index
pri rep docs.count docs.deleted store.size pri.store.size
yellow open   fulltext               5   1          2
0     10.9kb         10.9kb yellow open   fulltext_opt
1   1          2            0       159b           159b
```

That's it, our documents where copied from one index to the other. Now you can remove the index created by mongo-connector if you want. The last step is to try out our new index and see if it really supports partial matching for our autocomplete function:

```
curl -H 'Content-Type: application/json' \
    localhost:9200/fulltext_opt/articles/_search?pretty \
    -d "{ \"query\": { \"match\": { \"title\": { \"query\":
\"chi\", \"analyzer\": \"standard\" } } } }"


{
  "took" : 12,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.3125,
    "hits" : [ {
      "_index" : "fulltext_opt",
      "_type" : "articles",
      "_id" : "579e0a35c6d02e54ad6fe556",
      "_score" : 0.3125,
      "_source" : {
        "content" : "In the most recent example in a growing
trend of big deals for smartphone-based games, a consortium
of Chinese investors led by the game company Shanghai Giant
Network Technology said in a statement on Saturday that it
would pay $4.4 billion to Caesars Interactive Entertainment
for Playtika, its social and mobile games unit. Caesars
Interactive is controlled by the owners of Caesars Palace
and other casinos in Las Vegas and elsewhere.",
        "title" : "Chinese Group to Pay $4.4 Billion for
Caesars Mobile Games"
      }
    } ]
  }
}
```

Voilà! Got our document back. Note that we defined in our query the specific analyser we wanted to use and set it to the standard one:

```
{
    title: {
        query: "chi",
        analyzer: "standard"
    }
}
```

If we don't do this, since we're querying the index with our custom analyser, it would use the `autocomplete` analyser by default and query

using the edge n-grams of the query text. This would lead to unwanted results, since we want to search for the text `chi` specifically, and not for `c`, or `ch` or `chi`. This is why we have to explicitly set the analyser to the standard one.

Handling new MongoDB inserts

So far we moved all our MongoDB collection contents to the fulltext_opt index by using mongo-connector. The only problem is that, as you probably remember, mongo-connector copies to an index with the same database name from MongoDB. This means that if we keep mongo-connector running as we have it now, all the new documents inserted in the database will be indexed in the `fulltext` index in ES and not the optimized `fulltext_opt`.

The way to solve this is by configuring a bit more the mongo-connector command. There are many configuration options that you can find here. We'll use two of them: `namespaces.include` (`-n` in command line) and `namespaces.mapping` (`-g` in command line). You can see how to configure mongo-connector through a json file, here I'll just use the command line arguments way.

The `-n` option will tell mongo-connector which collections from MongoDB we want to index. The syntax for this is `database_name.collection_name`. In our case we want to index all the articles from the `fulltext` database. So we'll pass a command line argument like this: `-n fulltext.articles`.

The `-g` option will tell mongo-connector into which index it should put all the documents taken from the collections defined with the `-n` option. So in our case we want to put all our articles in the `fulltext_opt` index. We need to provide as well which type within ES we want to use, so the full argument would be: `-g fulltext_opt.articles`, since we want our articles stored with the articles type in the index.

That's it, now we can run the command like this:

```
$ mongo-connector -m 127.0.0.1:27017 -t 127.0.0.1:9200 -d elastic2_doc_manager -n fulltext.articles -g fulltext_opt.articles
```

If you keep mongo-connector running, all new inserts will be indexed in ES as well. Go ahead and insert a new document in the articles collection and then send a query to the ES index, the document should be returned.

Conclusion

With the excuse of creating an autocomplete compatible index, we learnt how to mix both MongoDB with Elastic Search and to keep both of them in sync with the `mongo-connector` module.

I hope your curiosity goes beyond this article contents by trying out alternatives for index tuning and checking other configurations that might suit better for your own systems. As I mentioned around the beginning, the autocomplete example was just
to show how you can import data from MongoDB into Elastic Search. The possibilities that ES brings to text indexing are huge, so go ahead and play with it!

Don’t miss our posts, follow us now on Twitter! 👇