6 not so obvious things about Elasticsearch

Dariusz Mydlarz
Feb 4, 2019 · 8 min read

Photo by Ben White on Unsplash

Elasticsearch is a widely adopted search engine. It is used by big names like Netflix, Microsoft, eBay, Facebook and others. It is easy to start working with, but hard to master in the long run. In this article we share six not so obvious things about Elasticsearch worth knowing before using it in your systems.

1. Elastic Stack

Elasticsearch was initially developed as an independent product. Its sole role was to provide a scalable search engine, that can be used from any language. Thus it was created with a distributed model at the very core with a REST API to communicate with it.

After an early adoption phase new tools were invented to work with Elasticsearch. It started with Kibana — for visualisation and data analysis, and Logstash — for logs collection. Currently there is a number of tools which are all developed under care of Elastic company:

1. Elasticsearch — you know, for search,
2. Kibana — data analytics and visualisation,
3. Logstash — server-side data processing pipeline,
4. Beats — single-purpose data shippers,
5. Elastic Cloud — hosting Elasticsearch clusters,
6. Machine Learning — for discovering data patterns,
7. APM — Application Performance Monitoring,
8. Swiftype — one-click site search.

Number of tools is growing every year, that enables companies to meet new goals, and create new opportunities.

2. Two kinds of data sets

Basically you can index (ie. store) any data you want in Elasticsearch. But actually there are two classes of them, which heavily impacts how the cluster should be configured and managed: **static data** and **time series data**.

Static data are datasets that may grow or change slowly. Like a catalog or an inventory of items. You can think of them as of data you store in your regular databases. Blog posts, library books, orders, etc. You may want to index such data in Elasticsearch to enable blazing fast searches, that outrages the regular SQL databases.

On the other hand, you can store time series datasets. Those can be events associated with a moment in time that typically grows rapidly, like log files or metrics. You basically index them in Elasticsearch for data analysis, pattern discovery and systems monitoring.

Depending on the type of data you store you should model your cluster in a different way. For static data you should choose a fixed number of indices and shards. They are not going to grow very fast, and you always want to search across all the documents in the dataset.

For time-series data you should pick time-bound rolling indices. You will more often query recent data, and eventually will even like to drop, or at least archive the obsolete documents in order to save money on machines.

3. Search score

The main purpose of Elasticsearch is to provide a search engine. The goal is to serve the best matching documents. But how does actually Elasticsearch know what are they?

For every search query Elasticsearch computes a relevance score. The score is based on the tf-idf algorithm, which stands for Term Frequency — Inverse Document Frequency.

Basically two values are calculated in this algorithm . The first one — term frequency — says how frequent a given *term* is being used in a document. The second one — inverse document frequency — says how unique a given term is across all documents.

For instance if we have two documents:

1. *To be or not to be, that is the question.*
2. *To be. I am. You are. He, she is.*

The TF for the term *question* is

1. for document 1: 1/10 (1 occurrence out of 10 terms)
2. for document 2: 0/9 (0 occurrences out of 9 terms).

On the other hand the IDF is calculated as a single value for a whole dataset. It is a ratio of all documents to documents containing the searched term.

In our case it is:

```
log(2/1) = 0.301
```

(2 — number of all documents, 1 — number of documents containing *question* term).

Finally the tf-idf score for both document is calculated as product of both values:

- document 1: 1/10 x 0.301 = 0.1 * 0.301 = 0.03
- document 2: 0/9 x 0.301 = 0 * 0.301 = 0.00

Now we see that document 1 got relevancy of value **0.03**, while document 2 got **0.00**. Thus document 1 will be served higher on a results list.

4. Data model

Elasticsearch has two benefits in terms of performance. It is horizontally scalable and very fast. Where does the latter come from? It is based on the fact how data is stored.

When you index a document it is being passed through three steps: character filters, a tokenizer and token filters. They are used to normalize the document. For instance a document:

```
To be or not to be, that is the question.
```

may be actually stored as:

```
to be or not to be that is the question
```

if punctuation marks are removed and all terms are lowercased.

That is not the end. It can be as well stored as

```
question
```

if the stop word filter is applied which removes all the common language terms like: *to, be, or, not, that, is, the*.

So this is the indexing part. But the same steps are applied when searching for documents. The query is being as well filtered for chars, tokenized and filtered for tokens. Then Elasticsearch is searching for documents with the normalized terms. Fields in Elasticsearch are stored in an inverted index structure, and it makes picking up matching documents really fast.

Specific filters can be defined per field. Definitions are grouped into structures called **analyzers**. A field can be analyzed with multiple analyzers to achieve different goals. For instance it can be analyzed with a English analyzer, German Analyzer, etc. Then in a search phase you can define which flavour of field you want to scan and you will get your results.

By applying this behaviour, ElasticSearch can serve results times faster than regular databases.

5. Shards planning

Now comes the most often asked questions by newbies to Elasticsearch. How many shards and indices should I have? Why does this question arise? The number of shards can be set only at the very beginning of index creation.

So the answer really depends on the dataset you have. **The rule of thumb is that shards should consists of 20–40 GB of data**. Shards comes from Apache Lucene (which is the search engine that is used under the hood). Having in mind all the structures, and overheads that Apache Lucene uses for inverted indices and fast searches, there is no sense in having small shards, like 100 MB, or 1 GB.

20–40 GB is the recommended size by Elastic consultants. Remember, that a shard cannot be divided further, and resides always on a single node. Such sized shard can be as well easily moved to other nodes or replicated, if needed, within a cluster. Having this capacity of shard gives you recommended tradeoff between speed and memory consumption.

Of course in your particular case, the performance metrics can show something different, so keep in mind that this is just a recommendation, and you may want to achieve other performance goals.

In order to know how many shards per index you should have, you can simply estimate that, by indexing a number of documents into a temporary index and see how much memory they are consuming and how many of them you expect to have in a period of time (in a time-series datasets), or at all (in a static datasets).

Do not forget that even if you misconfigure the number of shards or indices, you can always reindex data to a new index that has a different number of shards set up.

Last but not least. You can always query for multiple indices at once. For instance you can have rolling indices for log-based data with daily retention and simply ask for all days from last month in one query. **Querying 30 indices with 1 shard has the same performance impact as querying 1 index with 30 shards**.

6. Node Types

Elasticsearch nodes can fulfil multiple roles. By default — which is good for small clusters — they can serve all of them. The roles I am writing about are:

1. master node,
2. data node,
3. ingest node,
4. coordinating-only node.

Each role has its consequences. **Master nodes** are in charge of cluster-wide settings and changes, like creating or deleting indices, adding or removing nodes and allocating shards to nodes.

Every cluster should consist of at least 3 master-eligible nodes, and actually do not need to have more of them. From all of the master-eligible nodes, the one is being picked as a master node, and its role is to perform cluster-wide actions. The other two nodes are required purely for high availability. Master nodes have low requirements on CPU, RAM and disk storage.

Data nodes are used for storing and searching data. Thus they have high requirements on all of the resources: CPU, RAM and disk. The more data you have, the higher the expectations are.

Ingest nodes are used for documents pre-processing before the actual indexing happens. They intercept bulk and index queries, apply transformations and then pass documents back to the index or bulk APIs. They require low disk, medium RAM and high CPU.

Coordination-only node is used as load balancer for client requests. They know where specific documents can reside and serve search requests only to those nodes. Then they perform scatter & gatter actions on the received results. The requirements for them are low disk, medium or high RAM and medium or high CPU.

Each node can serve one or many of the roles listed above. The coordination role is fulfilled by any type of node. In order to have a coordination-only node you have to disable all other roles on it.

Now comes the popular question. **What is the preferred way of configuring a big cluster?** Here are the recommendations:

1. three master nodes — which are not exposed to the world, and maintain cluster state and cluster settings,
2. a couple of coordination-only nodes — they listen to external requests, and act as smart load balancers to the whole cluster,
3. a number of data nodes — depending on dataset needs,
4. couple of ingest nodes (optionally) — if you are performing Ingest Pipelines and want to relieve other nodes of the impact of pre-processing documents.

The specific numbers depends on your particular use-case and must be sized based on performance tests.

. . .

Summary

These were couple of insights into Elasticsearch which we wanted to share with you. We hope this knowledge will help you delivering your own solutions.

PS. If you have further questions do not hesitate to ask. We have a Certified Elastic Engineer on board. Yes, it’s me ;)

• • •

Looking for Scala and Java Experts?

Contact us!

We will make technology work for your business. See the projects we have successfully delivered.

Elasticsearch Search Database Data Programming