Using the Power of Real-Time Distributed Search with ElasticSearch

by Yuriy Bondaruk

Skelia
Extending Your Enterprise

Introduction

Internet is a place where everyone in the world can find any information they want. But with billions of documents available in the web, how is it possible to find exactly what we want in seconds or less?

For this purpose special programs called ‘search engines’ are developed by using many algorithms for analyzing, stemming, building indexes and searching querying terms. In Java world there is one of the most popular open source libraries called Lucene from Apache. It is a high performance, reliable and widely used full-featured Information Retrieval library written in Java. On top of it are built a few servers such as Solr, ElasticSearch and others.

Nowadays most companies are trying to move all computation into the cloud and Search is not an exception. In this article I would like to consider ElasticSearch, which, besides many other features, is initially designed to work in clouds and is quite successful in accomplishing that mission.

ElasticSearch features

Developing high-loaded systems you encounter a problem of performing fast, up-to-date and comfortable search. ElasticSearch perfectly complies with all those requirements and even more. Here are major pros of the engine:

- High performance
- Open source
- Near-real-time indexing
- Ability to run in any Cloud
- Information exchange in JSON format via HTTP
- Scalability/extensibility
- Multi-tenancy
- Simple installation and configuration procedures
- Provided interfaces (REST, Java and Groovy API)

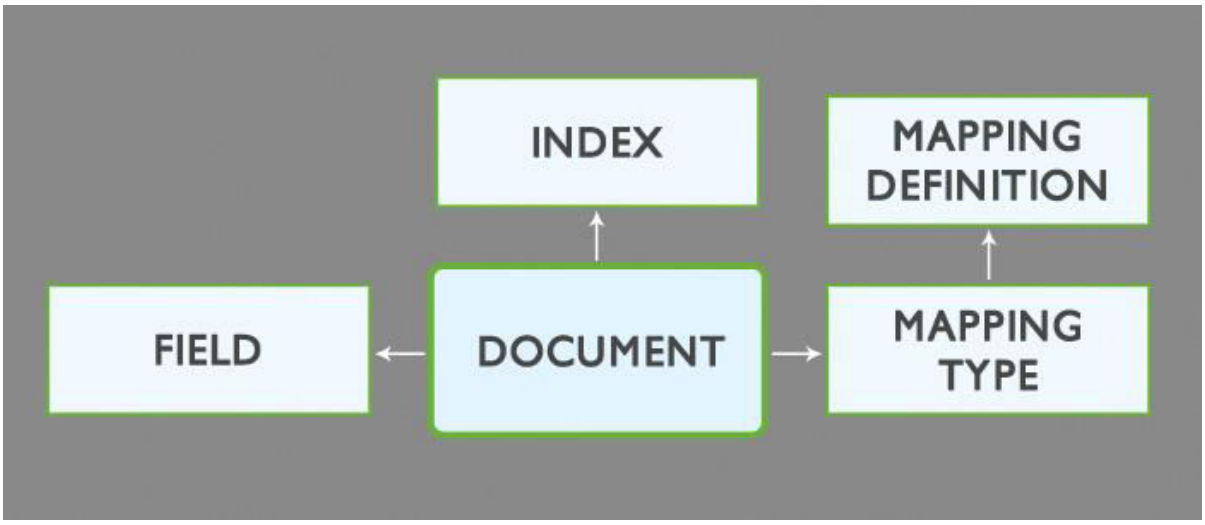
Let’s dig deeper

ElasticSearch is a flexible and powerful open source, distributed real-time search and analytics engine. It uses Apache Lucene as a base and makes it easier to create and implement large search systems. It is schema-free and document-oriented which are very important technical innovations. ElasticSearch has been designed with the cloud in mind. Indexing and searching are performed via simple http requests.

Data structure

ElasticSearch uses document-based data structure. Each document has index, id and set of fields. When new document type or field comes in then ES builds schema for it dynamically. So there is no need to define a strict structure of each type. Of course its also possible to define structure manually in mapping file or via Mapping API. There can be specified the following parameters:

- field names and their types
- whether they are required or not
- the way in which those properties are indexed
- which one should be used as a unique key
- which should be stored
- whether field should be searchable through '_all' or not
- if its value should be "highlightable"



An index in ElasticSearch may store documents of different “mapping types”. It allows one to associate multiple mapping definitions for each mapping type. A mapping type is a way of separating the documents in index into logical groups (like tables in relational databases).

Indexing

Indexing is one of the most important procedures search engines perform. But for it, search would take considerably more time and consume huge amount of resources. In the reality searchers don't perform queries on saved text but on indexes. That's why it's highly important to create efficiently.

It's like in books when we need to find a page with some word. We go to the back of the book and check indexes of words instead of reading all pages. This type of indexing is called inverted indexing (it inverts "page -> words" structure to a keyword-centric data structure "word -> pages").

Before storing a text in index ES analyzes it. Currently there are a few default analyzers, but it's also possible to add your own. One of the most efficient is snowball analyzer. It works very well with stems and roots of words. For example, a document contains words "searchable", "searched" and "searching". All of them will

will be transformed into “search” by analyzer and then added to the index, pointing back to full version of the document. The same happens when a user searches some word – first it analyzes the structure of the word and then uses its root for querying the index to get a list of matching documents.

Searching

Modern applications require not only full-text search by a keyword, but also more complex queries that would allow, for example, to filter out unnecessary data, return results in a certain order or get statistics for each term in the query (e.g., in how many documents occurs a word). ES allows to do that easily and provide results very quickly.

It should be noted that filtered queries could be cached and therefore all the following searches with the same filter would return results immediately.

A very powerful feature of ES is Faceting. It allows to get aggregated data along with standard search query. Here is a list of some facet types:

- Terms (get the most popular terms)
- Filter (number of hits matching the filter)

- Histogram (statistics per interval/bucket)
- Statistics (count, total, sum of squares, mean (average),
- minimum, maximum, variance, and standard deviation)
- Geo-distance (within 500m, 1km)

ES can retrieve a lot of useful information that could be used in software application for solving quite complex tasks. For example it's possible to get locations of hotels close to current location of the user (geo-distance facet), use terms facet for auto-complete functionality or get a histogram of prices per month and so on.

Big data? Let's scale the search!

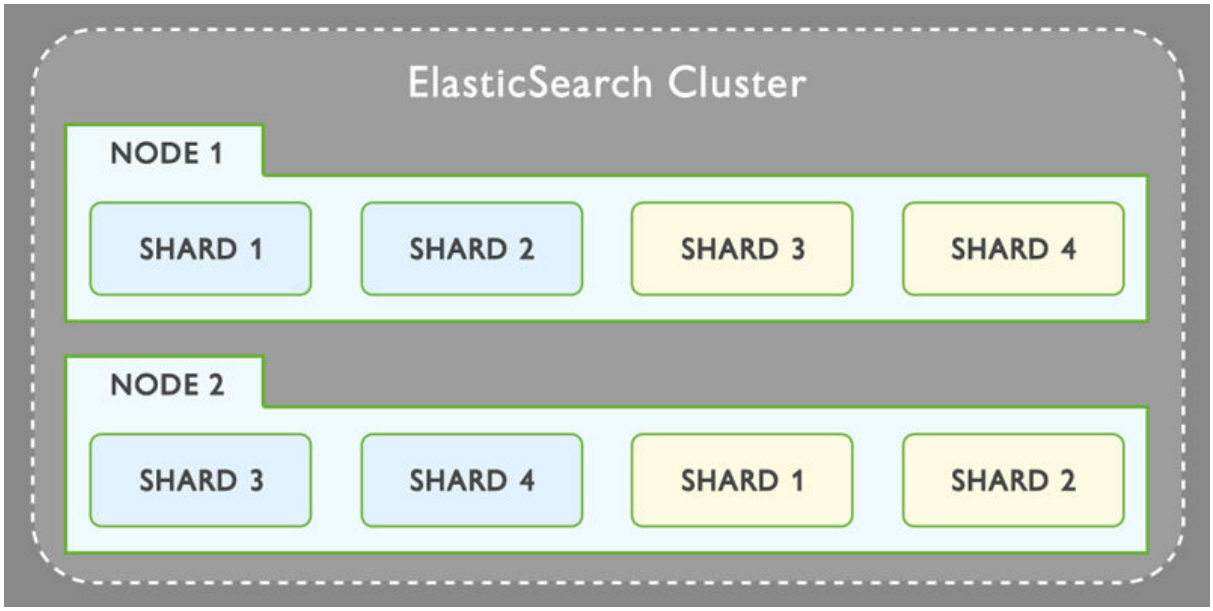
It is very easy to build clusters with ES. If two or more nodes are running on the same or on another server in the same network then by default all of them will automatically discover each other and will form clusters.

Indexes in ElasticSearch are scaling horizontally and scattered on shards (shard is a single Lucene indexes manageable by Elasticsearch). Shards in turn have replicas (backups) and all of them are located in nodes and nodes can be grouped in clusters. ES scans network with so called Zen Discovery mechanism which has IP multicast and unicast methods. Using one of these methods it checks presence of other nodes uniting them together forming a cluster. Unicast discovery is preferable because a new node is not necessary to know about all others in the cluster, it's enough to be connected to only one. Then it can directly ask a master node to get information about other nodes in the cluster.

In case of failure of a shard its role becomes playing appropriate

replica, so that the user does not notice any substitution, since it has the same data as the shard.

By default ES node configured to have 5 shards with 1 replica each. It means that indexes will have 5 primary shards and 5 it's reserved copies (replicas). What does it give? In case if cluster has at least 2 nodes and one of them fails then the cluster will still contain the entire index because the second node has copies of shards from the first one. If shards configured to have 2 replicas then ES guarantees data integrity even if 2 nodes fails (of course there should be more than 2 nodes in a cluster) and so on.



ElasticSearch Cluster with 2 nodes having 2 shards each and 1 replica per shard

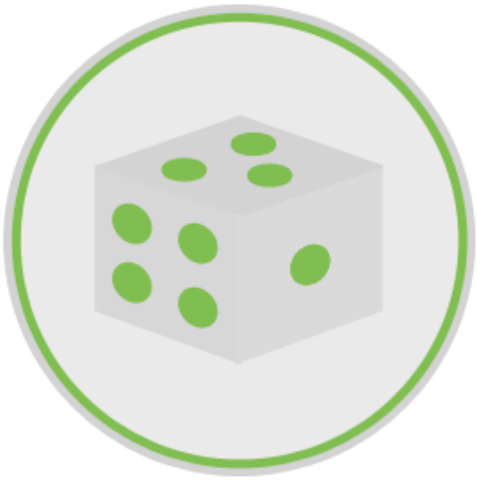
When the amount of data or requests increases, arises the question about extending cluster. In ES it's very simple: it is necessary to start one or more nodes and ES will automatically move a few shards into these new nodes within a cluster, thus unloading the old ones.

Also it should be noted that ES is able to effectively use advantages of multi-core processors.

Each node in ES may play of the following roles:

1. **Workhorse.** The node only holds data in one or more shards which are actually Lucene indexes, never becomes a master node. They are responsible for indexing and executing queries.
2. **Coordinator.** Serves as a master: not to store any data and have free resources. Node marked as a master is a potential candidate to become the Master of the cluster. ES automatically selects one of it. When Master node goes down then ES initiates new elections of the Master between all nodes having master role.
3. **Load balancer.** Node is neither master nor data node, but acts as a “search load balancer” (fetching data from nodes, aggregating results, etc.). It also responsible for ES REST interface.

By default ES node plays all three roles but it can be easily tuned in configuration file. Since ElasticSearch takes care of load balancing then there is no need to use any external tools for managing load of clusters.



What is an optimal number of shards and replicas?

The number of shards and number of replicas can be configured for each node separately. But how to know how many shards and replicas in a node do we need for our application? And how many nodes are needed to form an optimal cluster?

Actually there is no magic formula that always gives 100% correct answer to this question. But there are some general guidelines that can be used when selecting the number of replicas and shards.

1. Prepare the same environment that will be used on production
2. Create an index and configure a node to have only one shard and no replicas
3. Index data into that shard
4. Load the shard with the typical queries and typical load
5. Measure performance

At some moment querying becomes too slow. It means that the max capacity on that hardware is reached. That's the maximum shard size. Using it and knowing the size of index we can calculate the number of shards needed for us by formula:

$$\text{Number of shards} = \frac{\text{Index size}}{\text{Max shard size}}$$

Also ElasticSearch provides general rule of thumb that should be used when configuring of shards and replicas:
Assuming you have enough machines to hold shards and replicas, the rule of thumb is:

1. Having more shards enhances the indexing performance and allows to distribute a big index across machines.
2. Having more replicas enhances the search performance and improves the cluster availability

ElasticSearch in Cloud

ElsaticSearch can be installed on any cloud and extended to hundreds of instances without any changes in client application. A good video tutorial about installation of ElasticSearch with Cloudify is here.

ElasticSearch can be used on Amazon EC2 cloud. Here is a very handy guideline how to set it up.

Summary

ElasticSearch is a very powerful (near)real-time search engine written in Java and based on Apache Lucene. It can be installed on any cloud and easily scaled to hundreds of instances.

For developers it provides APIs to work from Java and Groovy by using libraries. But it doesn’t set a limitation for other languages or technologies, for that there is REST API with full set of possibilities.

Features like prompting a word during input, finding a closest restaurant/hotel/cinema, gathering statistics about appearance query word in different documents and so on enhance usability of modern applications and will attract users to use a software more often. Along with easiness of integration, configuration, scaling, ability to run on a Cloud opens a wide range of ElasticSearch usage opportunities.

Yuriy Bondaruk
Senior Java Developer at Skelia