Technology Review of Elasticsearch

By: Ido Tamir itamir2@illinois.edu

As a software engineer, having insight into the health status of my application in production alongside with how customers are using it is of enormous value. This is helpful when I want to debug problems we are seeing in production, get alerted if something is going wrong, see what features customers are using, and even just evaluate standard of performance customers are experiencing. In general, insights into all these metrics are tracked by logs that we are already outputting but the problem arises when we try to search through them. Standard applications that are continuously running and serve many customers (such as the one I work on professionally) may generate terabytes of text data if not more. Combing through and storing all this unstructured data requires a system with massive scaling capabilities alongside with reliability and efficiency. Figuring out how to easily query unstructured documents in a scalable and distributed way is what Shay Banon had initially set out to do when he built Elasticsearch but what he and his team ended up creating ended up being so much more.

Elasticsearch is a search engine/database that is designed specifically for unstructured data (such as logs from an application and many other forms of text data) to operate at large scale. It is built using Java and utilizes Apache Lucene. With respect to the CAP theorem which states that a distributed database can only guarantee 2 of the following attributes: consistency, availability, and partition tolerance, Elasticsearch doesn't fall cleanly in any 2 guarantees because it attempts to achieve all 3 but prioritizes some over others based on how its configured. To interface with Elasticsearch, we use REST API calls optionally with JSON bodies to convey our queries. This interface design was chosen for its flexibility and ubiquity although as I will mention later, there are other applications that hide this layer of communication and instead offer the user a graphical user interface.

From the bottom up the most basic element of an Elasticsearch database is a document. A document is the equivalent of a row in a relational database management system (or RDBMS for short). A document has fields which are the equivalent to columns in a RDBMS. These documents are stored in shards. Shards are all part of a single index which is analogous to a table. Lastly, all the indices together make up the cluster instance which is similar to the RDBMS term for a database.

Elasticsearch uses documents and fields instead of rows and columns because it allows for unstructured data (whereas columns and rows imply a certain uniform structure for every piece of data) which is consistent with a lot of text data in addition to the standard flexibility it provides. As would be expected, there is a pain associated with this because we cannot assume as much about the data that is being stored and must construct our queries and storing mechanism with that in mind. The shards are useful to ensure fault tolerance. When constructing an index the user must determine how many primary and secondary shards there are. As mentioned earlier, shards are where the data is stored. To ensure scalability and fault tolerance we create copies of the data which are referred to as replica shards. These shards copy data from the primary shards. This design implies 2 things immediately, all writes happen strictly on the primary shards and elastic search has eventual consistency but not strong

consistency (since reads are allowed on the replica shards which may not have been updated with the latest writes on the primary shard). This ensures fault tolerance because if the node that the primary shard is on is down, the replica shards can vote for one of the replica shards to become the new primary shard. If a replica shard goes down, things continue as usual and eventually if it comes back up it can read from the primary shard and updates itself. This protocol ensures Elasticsearch is robust and if a single node goes down it will still function properly.

Techniques to further improve performance and robustness, include strategic shard distributions. If you have multiple different indices with multiple respective shards, it is best to distribute them evenly across nodes. This means that a node shouldn't only have shards from a single index but instead each node should have a relatively uniform number of shards from each distribution. This ensures that if one node fails an index doesn't go down and that if one index is more often written/read from, the nodes who have its shards will share the burden equally instead of one node suffering from fielding too many requests.

Now that the anatomy and structure of Elasticsearch with multiple indices and respective shards is clear, it's important to comment on the inner-workings and how it stores the data it will use to execute text-based queries. Each shard is a Lucene shard meaning it has a highly optimized Lucene inverted index like we discussed in class. This index also points to a postings table that it utilizes to execute queries efficiently. To reduce the amount of space the posting table and inverted index take, Elasticsearch uses delta-encoding to encode numbers. This variable-length encoding means that encoding smaller numbers take less space, but it also increases logarithmically with the size of the number meaning large numbers won't require a lot of space either.

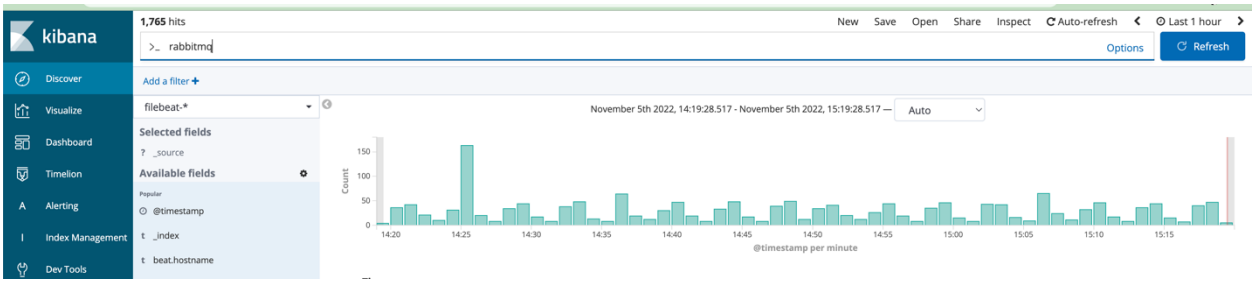
As mentioned above, Elasticsearch's main interface is REST API calls with JSON payloads and responses. Although this is a great interface for a developer to write an application to interface with, it's not as conducive to exploring data. For that reason, among others, there is a particular stack called the ELK stack that is very popular (and I use at my work). ELK stands for Elasticsearch, Logstash, and Kibana. Elasticsearch we have already covered, Logstash is a tool that ingests unstructured data, modifies it, and then sends it somewhere else. Logstash can be thought of as a processor of sorts that looks at your data and applies any transformations you want. Kibana is a front-end application that allows you to interface with Elasticsearch using a graphical user interface. Kibana will automatically create filters for your data, allow you to create dashboards with alerts, view data matching your query in real-time, as well make data exploration easy using their search bar that will construct a query to look across indices to find matches. Together, the ELK stack provides relatively simple solutions for the wide majority of log analysis that most business units require which is why it's such a popular stack.

Elasticsearch is one of the premier solutions available today for storing unstructured data such as text. It can be used for storing application data that users may want to search or data about the health and status of the application itself (this can be conveyed via logs or other means). By utilizing Lucene (which creates a highly optimized inverted-index and posting list) and multiple shards, Elasticsearch delivers a scalable and distributed solution which delivers great performance while also being fault tolerant and robust. In practice, Elasticsearch is often used as part of a stack such as the ELK stack that adds some data processing as well as a graphical user interface to make exploring the data easier. As part of the ELK stack, we get great

performance, robustness, and user interface and hence it is a widely adopted tool supported by many cloud providers and used by many businesses around the world.

Appendix:

I wanted to add some examples of what using the ELK stack is like in real life. I can't show log messages because this is from my work but this is an example of what you may see in Kibana:



As you can see I can search for all the log messages from the past hour with rabbitmq (a service we use for delayed messaging) in them. This can extend to let me search for logs from any given time range and also click on a specific bar to zoom into that time-frame. Next, I will often search for errors that have occurred so I will add an AND clause and then put "error". This way I get a nice visualization of some of the error logs that I can also compare to the previous image to see what proportion of them they are (in this case its 19/1765).