CS 410 Tech Review: Lucene and ElasticSearch

Chaaru Dingankar (NetID: chaarud2)

Introduction

Lucene is an open-source search engine library, distributed under the high-profile Apache software umbrella of open-source projects. It is a core component of a large number of widely used software packages and libraries, which combine Lucene with other features (for example, web crawling, or database-like features) to provide a particular service. One of the most widely used of these is Elasticsearch, which is designed to be a large-scale, distributed search engine that is designed to be deployed in the cloud to support queries at high throughput. It is designed to be used to power modern web applications and to be scalable.

In this paper, I will dive into specific features of Elasticsearch, namely the features/design considerations that allow Elasticsearch to be scaled up and down to power high-demand (either high-throughput, high-multitenancy, or really large data volume) applications.

Body

Elasticsearch is designed to scale basically linearly to the size of data and the demanded throughput, if configured correctly. There's a few knobs that are available to someone using Elasticsearch that they can turn in order to get the most performance out.

One of the primary design considerations of elasticsearch is that it is meant to be able to scale horizontally. This essentially means that given more resources (more nodes in the elasticsearch cluster), an operator can grow the size of the data under management and/or the maximum load on the cluster.

Elasticsearch is also designed to be vertically scalable - this is a simpler consideration than horizontal scalability which was discussed above, and simply means that Elasticsearch will be able to leverage more and more powerful hardware on any given node. This, however, quickly reaches a limit due to the limitations of a single server's capabilities - horizontal scaling is a much more long-term solution with a higher ceiling for keeping Elasticsearch performant.

Another major design consideration is sharding of indexes, which is what enables the horizontal scalability Elasticsearch provides. A single elasticsearch data collection (called a "document index") can be sharded into multiple partitions, so that the data is in smaller chunks than the entire collection as a whole. These shards can then be either all stored in a single node, or a single node could also not have all the shards of an index - this means that node won't be able to serve all possible requests, since it can't see all the data, but it does allow data sets to be managed that are larger than the storage available to a single node.

A single shard can be replicated across multiple nodes - this is a mechanism designed to reduce fault tolerance, but also, more useful for our problem of scalability, having multiple replicas reduces the potential load on a single node that would otherwise be the only place that a potential high-usage shard would be housed.

The final knob/design consideration that an Elasticsearch user can toggle is at the cluster level itself. While a single index can't be split across multiple clusters, data can often be split across multiple clusters (if absolutely necessary) by rethinking index design/how data is grouped into multiple collections. This should only be done after a single cluster runs into technical limitations, which under most use cases is an absolutely massive amount of data and concurrent requests (and therefore, a single cluster is generally sufficient to support most even high-demand use cases).

With this information about how Elasticsearch is designed, there is a small set of building blocks that can be changed (number of nodes, number of shards, number of replicas) to fit a user's needs. A user could want to tweak these for multiple goals. One is to manage how large an index the user can fit into the cluster (for example, when the single index is multiple hundreds of terabytes, or larger). A user could also want to optimize the number of nodes in the system, usually to reduce the cluster's maintenance costs (either in terms of dollars, and/or in terms of the maintenance effort and time associated with keeping many different machines online). And finally, a user could want to optimize the multi-tenancy of the application (ie, how many concurrent requests the cluster can serve while keeping the response within a certain latency).

These goals that a user can have interact with each other in complex ways - the most important conflict is probably the tension between latency and throughput, and the tension/tradeoff between the number of nodes in the cluster vs performance (this can be thought of as a cost/performance tradeoff). So it is possible (and, indeed, common) to tune for the optimization of one of these goals, and lose performance in another dimension.

One important insight for managing a large index (supposing without loss of generality, for simplicity's sake, that there is a single node in the cluster) is that adding shards doesn't change how many documents that node manages. The only thing that changes is that the “groupings” of documents inside that index are now in smaller chunks (since there's now more shards but the same total number of docs in the index). The number of documents in a single shard is what controls how long a single request takes - so increasing the number of shards will make a single search faster. However, having more shards on a single node means that the node won't be able to serve as many concurrent requests (since each shard runs a search request on a single CPU thread, which is a fixed number on that node).

Another important toggle is the number of replicas. This leads to a tradeoff between the resilience of the cluster as a whole, and the amount of memory a single node has. This is because the more shards a cluster has, the more redundancy it has - for example, if a single node fails, there is no chance that a cluster with replication factor of 2 will lose any data. However, the more shards a cluster has, the more documents a single node has to manage (since we're assuming here that the cluster is of a fixed size - of course, the cluster can be horizontally scaled to mitigate this problem, but that's a separate tuning “knob” to turn). This means that more shards requires more memory on a node, since there's more documents that need to be stored.

Perhaps the most important, and most complex, type of tuning to be done is to optimize the time it takes to execute a single search while keeping concurrency high (this is the fundamental latency/throughput tradeoff). There are a lot of inputs and interactions, but one fundamental idea is that shard size will impact this (as discussed above) but so will the type of request being executed (as the shard size increases, the time it takes to execute a query will increase in different ways - some kinds of query response times may increase linearly with shard size, some may be sub- or super-linear).

Conclusion

In conclusion, there’s several fundamental concepts in configuring an Elasticsearch cluster that will dictate performance. These are: the cluster itself, the nodes in the cluster, the shards in an index, and the replicas of every shard.

Performance itself can be measured in various ways, and is application-specific (ie, specific to the needs/constraints/desires of the particular use cases that the cluster is going to support). For example, an application could be extremely concerned with data integrity, and would therefore desire more replicas above all else. Another application could be concerned with cost, and therefore desire to reduce replicas and be indifferent to shard sizes, in order to save cost and possibly degrade performance.

In general, the important tradeoffs to be aware of are cost/performance and latency/throughput. Cost/performance is the simpler one, and often simply comes down to increasing the number of nodes in a cluster (horizontal scaling), or, alternatively, increasing the quality of the existing nodes (vertical scaling). The latency/throughput tradeoff is more complex, but can often be tackled initially by changing the shard size (by basically adding/removing shards from a fixed-size index), since more shards can mean better latency, but worse throughput.

References

https://medium.com/hipages-engineering/scaling-elasticsearch-b63fa400ee9e

https://www.elastic.co/guide/en/elasticsearch/reference/current/scalability.html

https://www.elastic.co/guide/en/elasticsearch/reference/current/size-your-shards.html

https://lucene.apache.org/