Script for the lecture: HLD Case Study 4 Elastic Search

Agenda

1. Full-text search
2. Elastic Search
3. Sharding keys

Imagine you are working for LinkedIn. And you are supposed to build a search feature which helps you search through all matching posts on LinkedIn.

Why is SQL not a good choice for designing such a system?

In SQL, we would be constructing a B+ tree of all posts. But this approach works fine when the posts are small in size (single word or 2 words). For large text, you will have to go to every tree node, which is time-consuming.

Someone might search for “job posting” which means that you are looking for all posts that contain “job posting” or “job post” in their body. This would require you to write a query like the following:

```
SELECT * FROM posts WHERE content LIKE “%job post%”.
```

The above query will end up doing the entire table scan. It will go to every row and do a string search in every post content. If there are N rows and M characters, the time complexity is O(N*M^2). Not good. Will bring LinkedIn down.

NoSQL DB?

Using a NoSQL DB would have a similar issue. In key value store or column family store, you would have to go through every row and search for matching values/columns which would take forever. Same for document store.

Correct Data Structure that can be used to solve this problem:

The data structure that we can use to solve this problem is:
Hashmap Or Trie

If we use a hashmap, the key-value pair can be as follows. The key will be the word to be searched, and the value will be the list of IDs of the document (or the review) where the queried word is present.

This is called as **INVERSE INDEX.**

APACHE LUCENE performs a similar thing**.** Every entry (the entire post for example) is called as a document. The following are the steps performed in this:

1. **Character elimination**
   In this phase, we remove characters such as “a”, “an”, “the”, etc. Although the name is character elimination, but it also includes word elimination.
2. **Tokenization**
   The entire post is broken down into words.
3. **Token Index**
   All of the tokens are broken down into their root word.
   Consider the following sentences for example,
   → When I ran the app, the app crashed.
   → While running the app, the app crashes.
   Here the pair of words
   → "ran" and "running"
   → "crashed" and "crashes"
   Carry the same meaning but in a different form of sentence. So this is what reduction to root word means. This process is also called **stemming.**
   So the words "running" and "ran" are converted to the root word "run"
   The words "crashes" and "crashed" are converted to the root word "crash"
4. **Reverse Indexing**
   In this phase, we store the (document id, position) pair for each word.
   For example,
   If for document 5, the indexed words after 3rd phase look as follows
   – decent - 1
   – product - 2
   – wrote - 3
   – money - 4

Then in the reverse indexing phase, the word "decent" will be mapped to a list that will look as
[(5,1)]
Where each element of the list is a pair of the (document id, position id)

Full-Text Search

Use cases of full-text search:

1. Log processing
2. Index text input from the user
3. Index text files / documents (for example, resume indexing to search using resume text).
4. Site indexing

Elastic Search

Apache Lucene is great. But it's just a software built to run on one single machine.
Single machine could however:

- Become single point of failure
- Might run out of space to store all documents.
- Might not be able to handle a lot of traffic.

So, ElasticSearch was built on top of Lucene to help it scale.

Should ES be more available vs more consistent?
Most search systems like LinkedIn post search is not supposed to be strongly consistent. Hence, a system like ElasticSearch should prioritize high availability.

Terminologies:

- **Document:** An entity which has text to be indexed. For example, an entire LinkedIn post is a document.
- **Index:** An index is a collection of documents indexed. For example, LinkedIN posts could be one index. Whereas Resumes would be a different index.
- **Node:** A node refers to a physical / virtual machine.

Sharding:
How would you shard if there are so many documents that the entire thing does not fit in a single machine?

1. Elastic search shards by document id.
2. Given a lot of document_ids, a document is never split between shards, but it belongs to exactly one shard.
3. **Sharding algorithm:** ElasticSearch requires you to specify the number of shards desired at the time of setup. If number of shards is fixed or does not change often, then we can use something much simpler than consistent hashing:
    a. A document with document_id will be assigned to shard no. `hash(document_id)%number of shards.`

Replication in ElasticSearch:

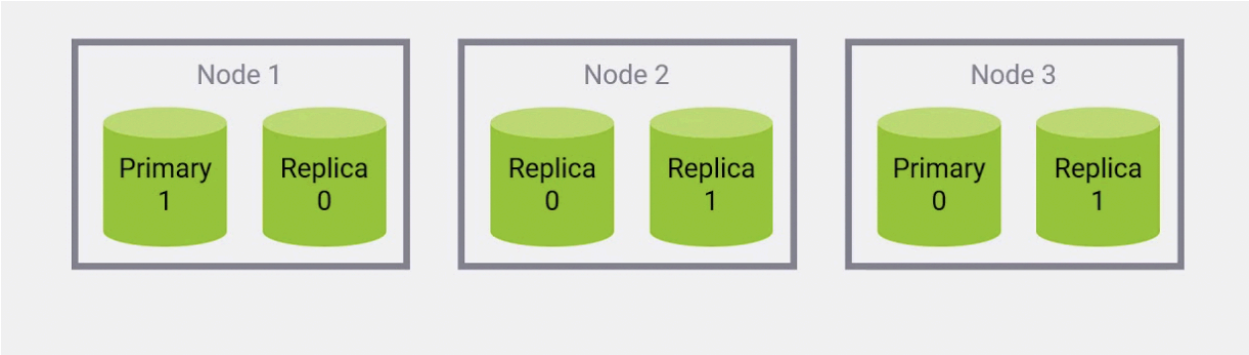
Just like number of shards, you can also configure number of replicas at the time of setup.
You need replicas because:

- Machines die. Replicas ensure that even if machines die, the shard is still alive and data is not lost.
- More replicas help in sharing the load of reads. A read can go to any of the replicas.

Just like in master-slave model, one of the replicas in the shard is marked as primary/master and the remaining replicas are followers/slaves.

So, imagine if num_nodes = 3, num_shards = 2(0 and 1), num_replicas = 3, then it could look like the following:

Given there are less number of nodes, hence, multiple shards reside on the same node. You can reduce that by adding more nodes into the cluster. With more nodes in the cluster, you can also configure and control the number of shards per node. Further reading at https://sematext.com/blog/elasticsearch-shard-placement-control/



Read / write flow:
Write (Index a new document): This finds the right shard for the document_id and the node containing primary replica. Request to index the document (just as writes happen in Lucene as detailed earlier) is sent to that node (primary replica). Updates from primary replica are propagated to slaves async.

Read (Given a phrase, find matching documents along with matching positions): Given documents are spread across shards, and any document can match, read in ElasticSearch is read in every shard.

When a read request is received by a node, that node is responsible for forwarding it to the nodes that hold the relevant shards, collating the responses, and responding to the client. We call that node the coordinating node for that request. The basic flow is as follows:

- Resolve the read requests to the relevant shards.
- Select an active copy of each relevant shard, from the shard replication group. This can be either the primary or a replica.
- Send shard level read requests to the selected copies.
- Combine the results and respond.

When a shard fails to respond to a read request, the coordinating node sends the request to another shard copy in the same replication group. Repeated failures can result in no available shard copies.

To ensure fast responses, some ElasticSearch APIs respond with partial results if one or more shards fail.