Elastic Search

Till now we have looked at:

1. a system that needs sharding vs a system that doesn't need sharding
2. different types of schema
3. Search typeahead : requires hashing
4. Messaging Systems: Column based

How to do a text search?

1. MySQL: indexes keywords
    a. Resume: Name, id, text
    b. Find people who have skills: Java / React JS
    c. like query will work but will not be very optimal
2. NoSQL
    a. **Key-Value**: Search in the value will be not optimal
    b. **Document**: Search will not be optimal if we want to query on a part document
    c. **Column Family**: Same as above, Search will not be optimal if we want to query on a part document
    d. **Graph**: How will we structure the node and edges?

Google Search

There are three parts to the architecture:

1. Crawler
    a. Goes to all web pages
    b. Collects lots of documents
    c. Billions of web pages
2. Index
    a. The problem is to store them in such a way, that it is quick to find words matching the search query.
    b. Search Query: Virat Kohli,
    c. find all documents where these words appear and give higher priority to pages where they appear together,
    d. run proprietary algorithm to rank these search result
    e. For the session we'll work with the subset of the problem:
        - given a phrase find all documents that completely or partially match the query
        - given a single word, how do we find matching pages?
        - given multiple words, how do we find matching pages?

f. If I search for “quick'' and the web page has “quickly”, should it match?
    - Yes, but low priority
    - Neglect stopwords: “A” quick fox
    - Number of times the search phrase appear
g. substring might not always yield correct results
    - think, thought
h. We need some form of mapping
    - word -> document, freq, positions
    - steming: every words has a root word
        1. run: running, run, ran, bhagna (hindi), odu (Kannada), correr (Spanish)
        2. for gibberish we can’t have a root word
    - get rid of insignificant words: articles, prepositions
        1. Ram is running
        2. Ram ran
i. Elastic Search is build on Apache lucene, it uses stemming
j. Alexa: Voice to text -> text -> semantic -> action (search, buy, volume)
k. How should we index?
    - Word -> documents -> frequency per doc + positions of occurrence
        1. Abhishek ate maggi -> Abhishek eat maggi
        2. Higher preference if they occur together
        3. Lower preference if they are scattered in the doc
l. Word (Key) -> [{doc_id, position}}
    - run -> [(1,10), (1,20), (2,3)]
    - The array can be quite large, hence, we’ll talk about sharding as well
    - This is also called inverted index

3. Query
    a. Search term: Abdul Kalam
    b. Ranking takes diff params: domain score, number of backlinks Words in H1, H2 etc
    c. Based on the query we can have three scenarios:
    d. Docs that contain both Abdul and Kalam: R1
    e. Docs that contain only Abdul: R2
    f. Docs that contain only Kalam: R2
    g. R1 can be split into
        - words appear consecutively: R11
        - words don’t appear consecutively: R12
    h. Sorting: R11 (higher count) > R11 (lower count) > R12 > R2
    i. Ranking
        - AND (all keywords)

        - OR
    j. Tech Design?
        - Abdul -> [(1,10), (1,25), (1,30), (2,15),(2,38), (3,50)]
        - Kalam -> [(1,15) ..... (3,51)]
        - Now we need to find documents that have all keywords
        - how to find documents with occurrences in the consecutive order? // DSA Problem
            1. A1: [(1,10), (1,25), (1,30), (2,15),(2,38), (3,50)]
            2. A2: [(1,15), (3,51)]
            3. variables: i, j , consecutive_cnt[doc_id], non_consecutive_cnt[doc_id]
            4. increment counters and update relevant counters
    k. In real word we have lots of documents
        - array corresponding to a value can be quite large
        - if the size is quite large, how would we shard?
        - word / document_id
        - Two types of queries
            1. Write*: (word, doc)
            2. Read*: word -> fetch matching docs // trying to optimise for this to keep latency low
    l. Availability vs Consistency: Availability

Sharding

Sharding on words:

- Abdul will be on S1
- Kalam will be on S2
- Array can become quite large // 1 billion entries
- Copy all entries to a central place to figure out the common docs // time consuming
- Go to S1, S2 and fetch A1 and A2 // can be done in parallel
- 2 pointer to find intersection // for billion entries, this will also take time
- Update will be slow, since we'll have to update shards corresponding to each word

Sharding on document

- Let's say we have 1000 shards, then every shard has 10 million document
- Every shard has its own independent inverted index
- Every single word from a page will be present on a single shard
- Shard1 can have max 10M entries
- In every shard (in parallel) use 2 pointer approach to find intersection

- Shards reply back with document_id, score
- with whatever response I have in 0.5 seconds, we sort & return the response
- Downside
    - every query will run on every shard
    - cost intensive
    - TPS on each shard will be very high

Extras
Lucene: does stemming of inverted index
ES: Takes care of sharding, coordinating with shards
Google Search*: uses lots of signals

- Keyword match
- Personalisation
- Page ranking algorithms (Domain authority/ Page Score, Backlinks)
- Removing Typo
- At Scaler we use ES for finding keywords in resume

Image search

- search via tags: ML models to apply tags to image, show images in captcha to train these ML models
- Not good if we need consistent data

Further reading:

1. https://blog.insightdatascience.com/anatomy-of-an-elasticsearch-cluster-part-i-7ac9a13b05db
2. https://buildingvts.com/elasticsearch-architectural-overview-a35d3910e515
3. Official Docs: https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html