Search Indexes

Background

In most websites, there is some functionality involving open ended search queries - see Google’s search engine, searching for products on Amazon, searching Twitter for posts matching keywords, etc.

Generally speaking, in order to optimize for these types of use cases, which can be a very intense query (since we may have to scan through millions or more documents), it can be helpful to use a search index!

Lucene

- The most popular open source search index
- Created in 1999
- The index behind popular search services such as ElasticSearch and Solr

Lucene Architecture

Lucene is in part highly performant because it uses an LSM tree + SSTable for its format:

- Writes first sent to an in memory buffer
  - Cannot be read just yet, need to be on disk first
- Eventually the buffer is written to an immutable SSTable index file on disk
- As SSTable files get larger they are eventually merged and compacted with other SSTable files
- Reads are sent to multiple SSTable files, and the results from them eventually need to be merged

Lucene Architecture Continued

When a document is first added to a Lucene index, it must be split into terms (this process is known as tokenizing):

- The way that a document is tokenized has serious implications for how we may query it later
  - Handling punctuation
  - Handling case
  - Handling contractions
  - Handling common words like “the” or “and”
  - These are all problems that occur in natural language processing as well!

Lucene Architecture Continued

After being tokenized, the document is given an ID, and added to something called the inverted index, which maps terms to document IDs that contain them. Note that since these are on SSTables, they are sorted by term.

Term	IDs
lewinsky	2, 3
malarkey	3, 18, 21
sheesh	1, 4, 12

Lucene Architecture Continued

What if we want to be able to go beyond finding just the term itself, and say also be able to get documents based on a similar query string?

Term	IDs
pee	2, 3
pen	14, 20
poop	3, 18, 21
purr	1, 4, 12

Lucene Architecture Continued

What if we want to be able to go beyond finding just the term itself, and say also be able to get documents based on a similar query string?

Term	IDs
pee	2, 3
pen	14, 20
poop	3, 18, 21
purr	1, 4, 12

1) Want to find all documents with terms starting with “pe”

Lucene Architecture Continued

What if we want to be able to go beyond finding just the term itself, and say also be able to get documents based on a similar query string?

Term	IDs
pee	2, 3
pen	14, 20
poop	3, 18, 21
purr	1, 4, 12

1) Want to find all documents with terms starting with “pe”
2) Binary search the SSTable to find the terms starting with “pe”, we can see those are “pee” and “pen”

Lucene Architecture Continued

If we are trying to do searches more complicated than just by the prefix of a term, we have to use additional logic.

- For example, if we wanted to search by term suffix, we could create a second copy of the index based on the terms with all of their characters reversed, and by keeping those sorted we could binary search on suffix
- We could do something similar for numeric terms by splitting numbers into their individual digits and seeing how many of the digits are the same to start with

Ultimately, Lucene has a lot of cool search capabilities not just on text, but on numbers, similar words (using Levenshtein distance), and geolocation!

ElasticSearch

ElasticSearch is a service that takes the capability of an individual Lucene index and allows it to run over a distributed cluster.

ElasticSearch is able to ensure availability through replication, but the major point here is to be able to hold index shards on different machines which are mapped based on the ID of the document. In this sense, ElasticSearch basically creates a bunch of local inverted indexes for the documents on a given node.

Ideally, you keep documents that are frequently searched together on the same shard to avoid cross shard queries.

ElasticSearch Caching

ElasticSearch is able to provide extremely fast performance on reads thanks to caching!

- Caching of index pages in memory by the operating system
- ElasticSearch caches queries on a shard level in memory
  - Not just the index itself but the actual result of the computation done
  - Assuming that the data on the shard has not changed
- Query caches also cache only parts of certain queries to be used again by different queries in the future if they require some data in common (use the same filters)

Conclusion

Search indexes are an incredibly important part of many large applications, and are capable of finding strings of text in a manner that is much faster than a typical database query. Lucene, using SSTables holding inverted indexes is able to achieve very good performance, and additional functionality has been built in to work just beyond exact term matching.

Ultimately, while Lucene at scale (in ElasticSearch for example) is very powerful, you still have to be very careful of how you organize/partition your data in order to avoid hotspots, while also limiting cross partition queries.