Back To Module Home

Basic Building Blocks for Modern System Design

0% completed

Search Module

Introduction to Building Blocks

Domain Name System

Load balancers

Cache

Databases

Key-value Store

Content Delivery Network (CDN)

Sequencer

Distributed Monitoring

Distributed Cache

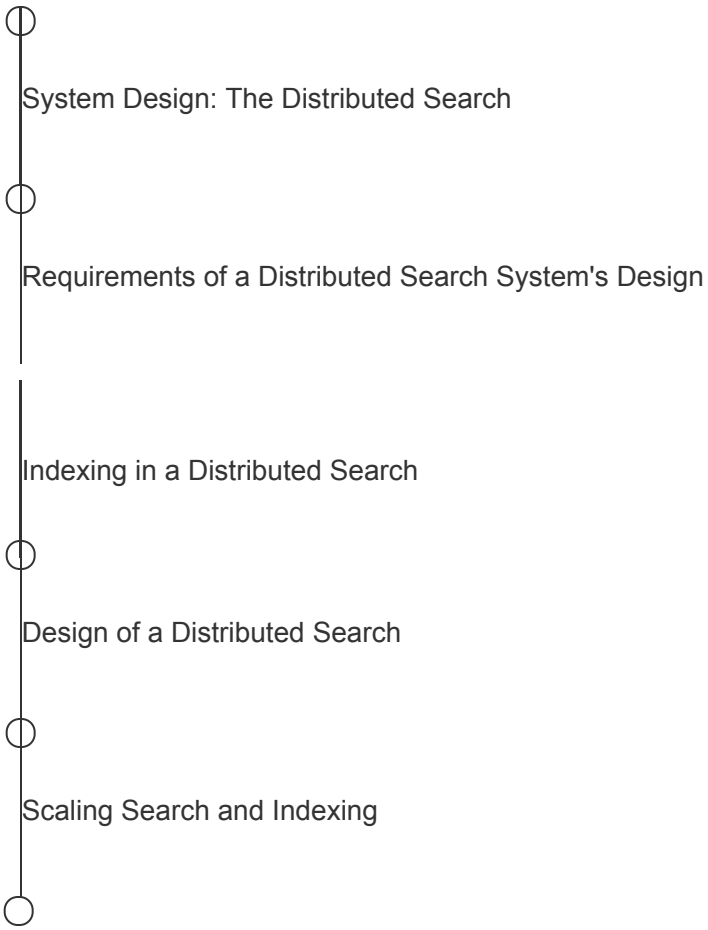
Distributed Messaging Queue

Pub-sub

Rate Limiter

Blob Store

Distributed Search



Indexing in a Distributed Search

Learn about indexing and its use in a distributed search.

We'll cover the following

We'll first describe what indexing is, and then we'll make our way toward distributing indexes over many nodes.

Indexing#

Indexing is the organization and manipulation of data that's done to facilitate fast and accurate information retrieval.

Build a searchable index#

The simplest way to build a searchable index is to assign a unique ID to each document and store it in a database table, as shown in the following table. The first column in the table is the ID of the text and the second column contains the text from each document.

Simple Document Index

ID	Document Content
1	Elasticsearch is the distributed and analytics engine that is based on REST APIs.
2	Elasticsearch is a Lucene library-based search engine.
3	Elasticsearch is a distributed search and analytics engine built on Apache Lucene.

The size of the table given above would vary, depending on the number of documents we have and the size of those documents. The table above is just an example, and the content from each document only consists of one or two sentences. With an actual, real-world example, the content of every document in the table could be pages long. This would make our table quite large. Running a search query on the document-level index given above isn't a fast process. On each search request, we have to traverse all the documents and count the occurrence of the search string in each document.

> **Note:** For a **fuzzy search**, we also have to perform different pattern-matching queries. Many strings in the documents would somehow match the searched string. First, we must find the unique candidate strings by traversing all of the documents. Then, we must single out the most approximate matched string out of these strings. We also have to find the occurrence of the most matched string in each document. This means that each search query takes a long time.

The response time to a search query depends on a few factors:

- The data organization strategy in the database.
- The size of the data.
- The processing speed and the RAM of the machine that's used to build the index and process the search query.

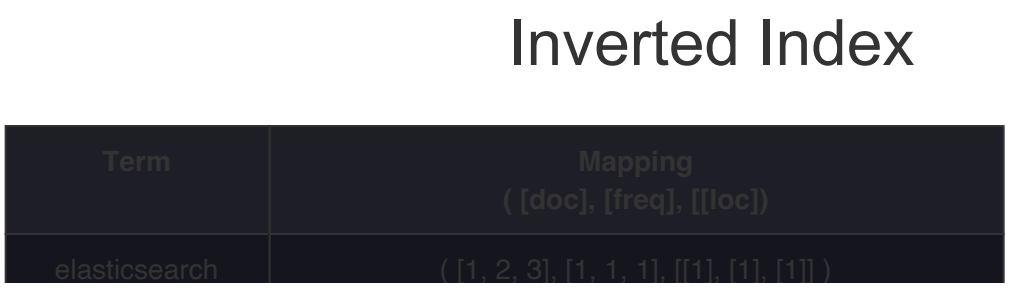
Running search queries on billions of documents that are document-level indexed will be a slowprocess, which may take many minutes, or even hours. Let's look at another data organization and processing technique that helps reduce the search time.

Inverted index#

An **inverted index** is a HashMap-like data structure that employs a document-term matrix. Instead of storing the complete document as it is, it splits the documents into individual words. After this, the **document-term matrix** identifies unique words and discards frequently occurring words like "to," "they," "the," "is," and so on. Frequently occurring words like those are called **terms**. The document-term matrix maintains a **term-level index** through this identification of unique words and deletion of unnecessary terms.

For each term, the index computes the following information:

- The list of documents in which the term appeared.
- The frequency with which the term appears in each document.
- The position of the term in each document.



distributed	([1, 3], [1, 1], [[4], [4]])
restful	([1], [1], [[5]])
search	([1, 2, 3], [1, 1, 1], [[6], [4], [5]])
analytics	([1, 3], [1, 1], [[8], [7]])
engine	([1, 2, 3], [1, 1, 1], [[9], [5], [8]])
heart	([1], [1], [[12]])
elastic	([1], [1], [[15]])
stack	([1], [1], [[16]])
lucene	([2, 3], [1, 1], [[9], [12]])
library	([2], [1], [[10]])
Apache	([3], [1], [[11]])

In the table above, the “Term” column contains all the unique terms that are extracted from all of the documents. Each entry in the “Mapping” column consists of three lists:

- A list of documents in which the term appeared.
- A list that counts the frequency with which the term appears in each document.
- A two-dimensional list that pinpoints the position of the term in each document. A term can appear multiple times in a single document, which is why a two-dimensional list is used.

> **Note:** Instead of lists, the mappings could also be in the form of tuples— such as doc, freq, and loc.

Inverted index is one of the most popular index mechanisms used in document retrieval. It enables efficient implementation of boolean, extended boolean, proximity, relevance, and many other types of search algorithms.

Advantages of using an inverted index

- An inverted index facilitates full-text searches.
- An inverted index reduces the time of counting the occurrence of a word in each document at the run time because we have mappings against each term.

Disadvantages of using an inverted index

- There is storage overhead for maintaining the inverted index along with the actual documents. However, we reduce the search time.
- Maintenance costs (processing) on adding, updating, or deleting a document. To add a document, we extract terms from the document. Then, for each extracted term, we either add a new row in the inverted index or update an existing one if that term already has an entry in the inverted index. Similarly, for deleting a document, we conduct processing to find the entries in the inverted index for the deleted document's terms and update the inverted index accordingly.

Searching from an inverted index#

Consider a system that has the following mappings when we search for the word "search engine:"

Term	Mapping
search	([1, 2, 3], [1, 1, 1], [[6], [4], [5]])
engine	([1, 2, 3], [1, 1, 1], [[9], [5], [8]])

Both of these words are found in documents 1, 2, and 3. Both words appear once in each document.

The word "search" is located at position 6 in document 1, at position 4 in document 2, and position 5 in document 3.

The word "engine" is located at position 9 in document 1, position 5 in document 2, and

position 8 in document 3.

A single term can appear in millions of documents. Thus, the list of documents returned against a search query could be very long.

Question

Would this technique work when too many documents are found against a single term?

Show Answer

Factors of index design#

Here are some of the factors that we should keep in mind while designing an index:

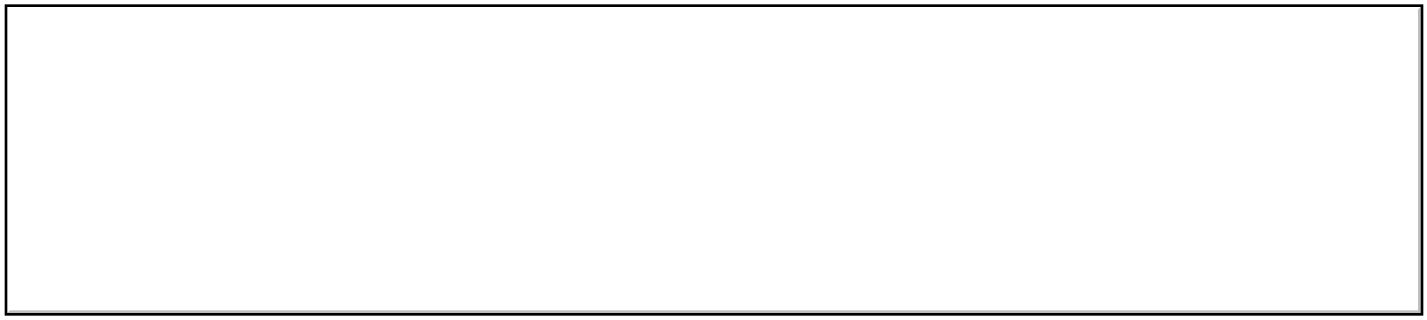
- **Size of the index**: How much computer memory, and RAM, is required to keep the index. We keep the index in the RAM to support the low latency of the search.
- **Search speed**: How quickly we can find a word from an inverted index.
- **Maintenance of the index**: How efficiently the index can be updated if we add or remove a document.
- **Fault tolerance**: How critical it is for the service to remain reliable. Coping with index corruption, supporting whether invalid data can be treated in isolation, dealing with defective hardware, partitioning, and replication are all issues to consider here.
- **Resilience**: How resilient the system is against someone trying to game the system and guard against search engine optimization (SEO) schemes, since we return only a handful of relevant results against a search.

In light of the design factors listed above, let’s look at some problems with building an

index on a centralized system.

Indexing on a centralized system#

In a **centralized search system**, all the search system components run on a single node, which is computationally quite capable. The architecture of a centralized search system is shown in the following illustration:



The architecture of a centralized search system

- The **indexing process** takes the documents as input and converts them into an inverted index, which is stored in the form of a binary file.
- The **query processing** or **search process** interprets the binary file that contains the inverted index. It also computes the intersection of the inverted lists for a given query to return the search results against the query.

These are the problems that come with the architecture of a centralized search system:

- SPOF (single point of failure)
- Server overload
- Large size of the index

SPOF: A centralized system is a single point of failure. If it’s dead, no search operation can be performed.

Server overload: If numerous users perform queries and the queries are complicated, it stresses the server (node).

Large size of the index: The size of the inverted index increases with the number of documents, placing resource demands on a single server. The bigger the computer system, the higher the cost and complexity of managing it.

> **Note:** With a distributed system, low-cost computer systems are utilized, which is cost effective overall.

An inverted index needs to be loaded into the main memory when adding a document or running a search query. A large portion of the inverted index must fit into the RAM of the machine for efficiency.

According to Google analytics in 2022, there are hundreds of billions of web pages, the total size of which is around 100 petabytes. If we make a search system for the worldwide web, the inverted index size will also be in petabytes. This means we have to load petabytes of data into the RAM. It's impractical and inefficient to increase the resources of a single machine for indexing a billion pages instead of shifting to a distributed system and utilizing the power of parallelization. Running a search query on a single, large inverted index results in a slow response time.

> **Note:** Searching a book from a shelf that holds a hundred books is easier than searching a book from a shelf holding a million books. The search time increases with the volume of data we search from.

Attacks on centralized indexing can have a higher impact than attacks on a distributed indexing system. Furthermore, the odds of bottlenecks (which can arise in server bandwidth or RAM) are also lower in a distributed index.

In this lesson, we learned about indexing, and we looked into the problems of indexing on a centralized system. The next lesson presents a distribution solution for indexing.

Requirements of a Distributed Search…

Next

Design of a Distributed Search

Mark as Completed

Report an Issue