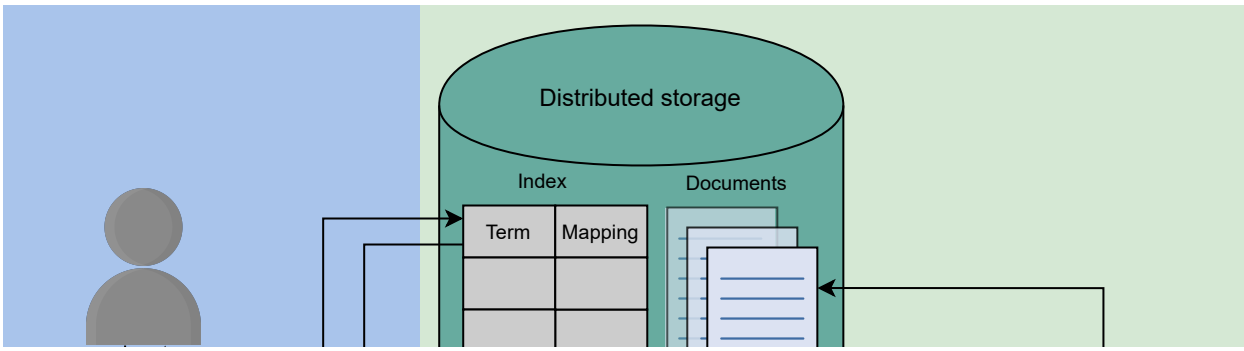
Design of a Distributed Search

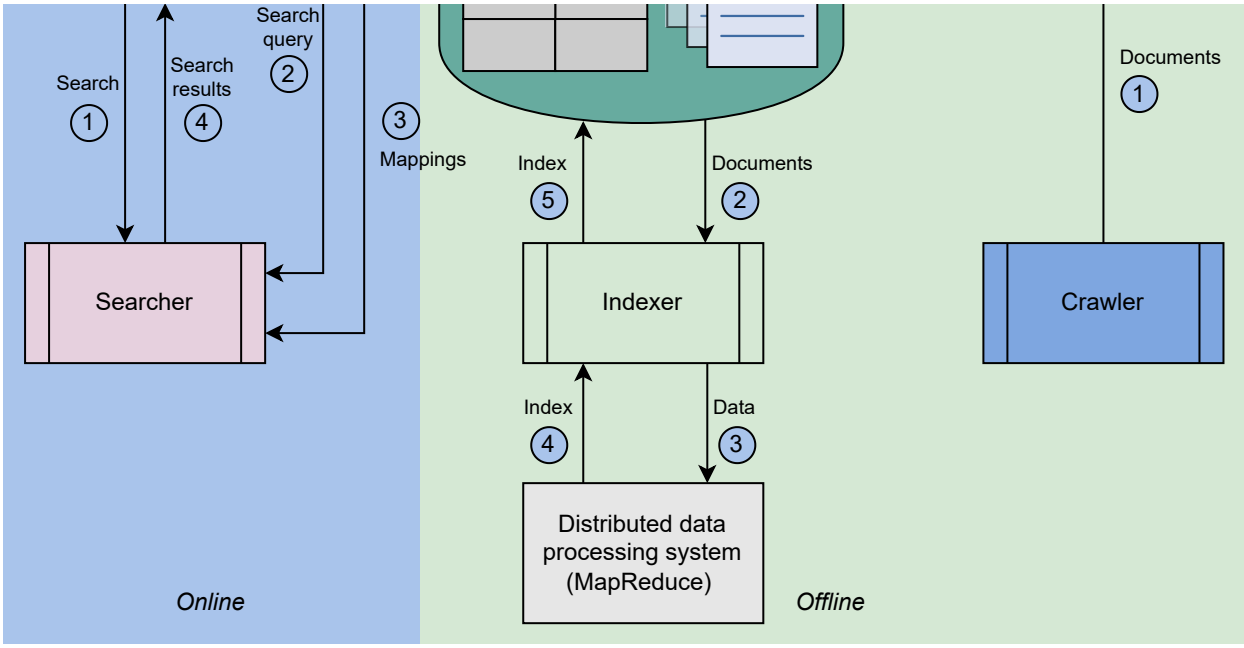
Get an overview of the design of a distributed search system that manages a large number of queries per second.

We'll cover the following

High-level design

Let’s shape the overall design of a distributed search system before getting into a detailed discussion. There are two phases of such a system, as shown in the illustration below. The ***offline phase*** involves data crawling and indexing in which the user has to do nothing. The ***online phase*** consists of searching for results against the search query by the user.





High-level design of a distributed search system

- The **crawler** collects content from the intended resource. For example, if we build a search for a YouTube application, the crawler will crawl through all of the videos on YouTube and extract textual content for each video. The content could be the title of the video, its description, the channel name, or maybe even the video's annotation to enable an intelligent search based not only on the title and description but also on the content of that video. The crawler formats the extracted content for each video in a JSON document and stores these JSON documents in a distributed storage.
- The **indexer** fetches the documents from a distributed storage and indexes these documents using **MapReduce**, which runs on a distributed cluster of commodity machines. The indexer uses a **distributed data processing system** like MapReduce for parallel and distributed index construction. The constructed index table is stored in the distributed storage.
- The **distributed storage** is used to store the documents and the index.
- The **user** enters the search string that contains multiple words in the search bar.

- The **searcher** parses the search string, searches for the mappings from the index that are stored in the distributed storage, and returns the most matched results to the user. The searcher intelligently maps the incorrectly spelled words in the search string to the closest vocabulary words. It also looks for the documents that include all the words and ranks them.

API design

Since the user only sends requests in the form of a string, the API design is quite simple.

Search: The `search` function runs when a user queries the system to find some content.

```
search(query)
```

Parameter	Description
`query`	This is the textual query entered by the user in the search bar, based o are found.

Detailed discussion

Since the indexer is the core component in a search system, we discussed an indexing technique and the problems associated with centralized indexing in the previous lesson. In this lesson, we consider a distributed solution fo indexing and searching.

Distributed indexing and searching

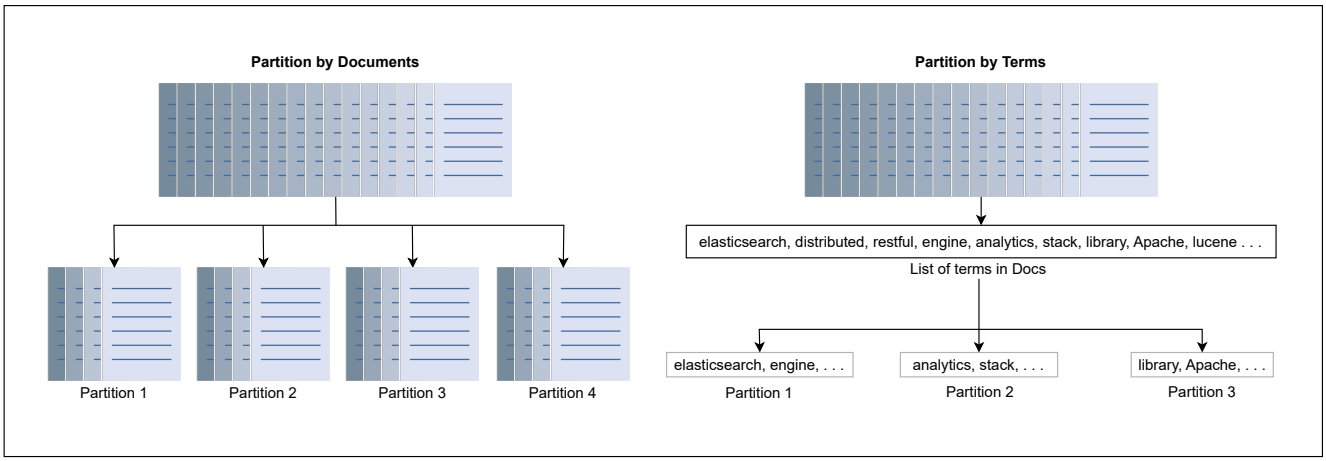
Let’s see how we can develop a distributed indexing and searching system. We understand that the input to an indexing system is the documents we created during crawling. To develop an index in a distributed fashion, we employ a large number of low-cost machines (nodes) and partition or divide the documents based on the resources they have. All the nodes are connected. A group of nodes is called a **cluster**.

Tip

We use numerous small nodes for indexing to achieve cost efficiency. This process requires us to partition or split the input data (documents) among these nodes. However, a key question needs to be addressed: How do we perform this partitioning?

The two most common techniques used for data partitioning in distributed indexing are these below:

- **Document partitioning**: In document partitioning, all the documents collected by the web crawler are partitioned into subsets of documents. Each node then performs indexing on a subset of documents that are assigned to it.
- **Term partitioning**: The dictionary of all terms is partitioned into subsets, with each subset residing at a single node. For example, a subset of documents is processed and indexed by a node containing the term “search.”

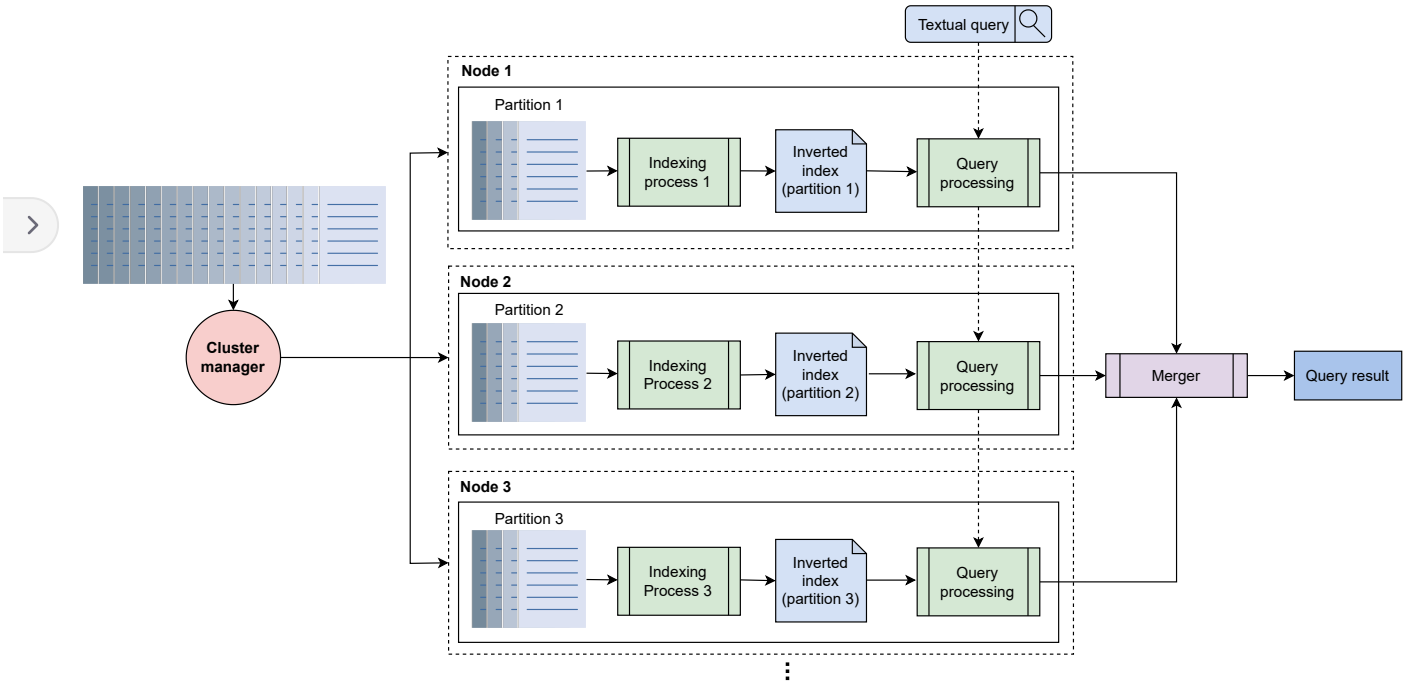


Types of data partitioning in a distributed search

In term partitioning, a search query is sent to the nodes that correspond to the query terms. This provides more concurrency because a stream of search queries with different query terms will be served by different nodes. However, term partitioning turns out to be a difficult task in practice. Multiword queries necessitate sending long mapping lists between groups of nodes for merging, which can be more expensive than the benefits from the increased concurrency.

In document partitioning, each query is distributed across all nodes, and the results from these nodes are merged before being shown to the user. This method of partitioning necessitates less inter-node communication. In our design, we use document partitioning.

Following document partitioning, let's look into a distributed design for index construction and querying, which is shown in the illustration below. We use a cluster that consists of a number of low-cost nodes and a cluster manager. The cluster manager uses a MapReduce programming model to parallelize the index's computation on each partition. MapReduce can wor on significantly larger datasets that are difficult to be handled by a single large server.



Distributed indexing and searching in a parallel fashion on multiple nodes in a cluster of commodity machines

The system described above works as follows:

Indexing

- We have a document set already collected by the crawler.
- The **cluster manager** splits the input document set into N number of partitions, where N is equal to three in the illustration above. The size of each partition is decided by the cluster manager given the size of the data, the computation, memory limits, and the number of nodes in the cluster. All the nodes may not be available for various reasons. The cluster manager monitors the health of each node through **periodic heartbeats**. To assign a document to one of the N partitions, a **hashing function** can be utilized.
- After making partitions, the cluster manager runs indexing algorithms for all the N partitions simultaneously on the N number of nodes in a cluster. Each indexing process produces a tiny inverted index, which is stored on the node's local storage. In this way, we produce N tiny inverted indices rather than one large inverted index.

Searching

- In the search phase, when a user query comes in, we run parallel searches on each tiny inverted index stored on the nodes' local storage generating N queries.
- The search result from each inverted tiny index is a mapping list against the queried term (we assume a single word/term user query). The **merger** aggregates these mapping lists.
- After aggregating the mapping lists, the merger sorts the list of documents from the aggregated mapping list based on the frequency of the term in each document.
- The sorted list of documents is returned to the user as a search result. The documents are shown in sorted (ascending) order to the user.

> **Note:** We've designed a search system where we utilized a distributed system and parallelized the indexing and searching process. This helped us handle large datasets by working on the smaller partitions of documents. It should be noted that both searching and indexing are performed on the same node. We refer to this idea as **colocation**.

The proposed design works, and we can replicate it across the globe in various data centers to facilitate all users. Thus, we can achieve the following advantages:

- Our design will not be subject to a single point of failure (SPOF).
- Latency for all users will remain small.
- Maintenance and upgrades in individual data centers will be possible.
- Scalability (serving more users per second) of our system will be improved.

Replication

We make replicas of the indexing nodes that produce inverted indices for the assigned partitions. We can answer a query from several sets of nodes with replicas. The overall concept is simple. We continue to use the same architecture as before, but instead of having only one group of nodes, we have R groups of nodes to answer user queries. R is the number of replicas. The number of replicas can expand or shrink based on the number of requests, and each group of nodes has all the partitions required to answer each query.

Each group of nodes is hosted on different availability zones for better performance and availability of the system in case a data center fails.

> **Note:** A load balancer component is necessary to spread the queries across different groups of nodes and retry in case of any error.

Replication factor and replica distribution

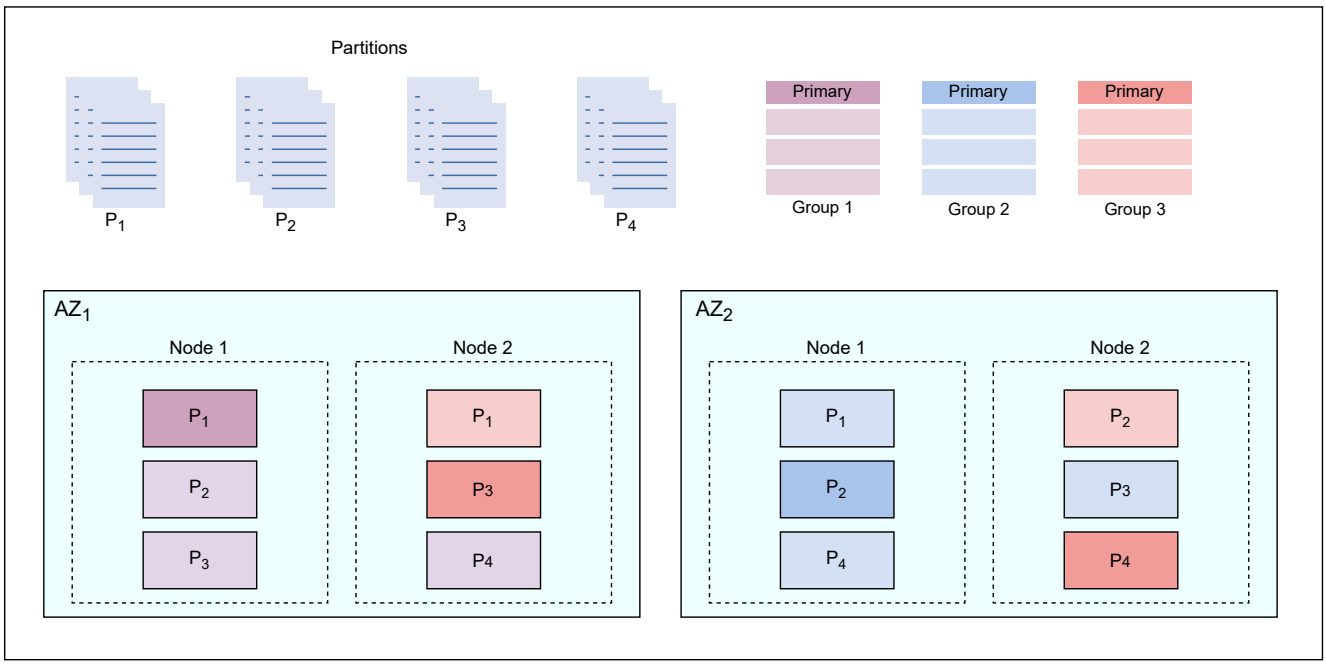
Generally, a replication factor of three is enough. A replication factor of three means three nodes host the same partition and produce the index. One of the three nodes becomes the primary node, while the other two are replicas. Each of these nodes produces indexes in the same order to converge on the same state.

To illustrate, let's divide the data, a document set, into four partitions. Since the replication factor is three, one partition will be hosted by three nodes. We'll assume that there are two availability zones (AZ_1 and AZ_2). And in each availability zone, we have two nodes. Each node acts as a primary for only one partition (For example, Node 1 in AZ_1 is the primary node for partition P_1). The three copies (pink, blue, and purple) for a partition are shared between the two AZ instances so that two copies are in one zone a

the third copy is in another zone. Three colors represent three replicas of each partition. For example, the following is true for partition P_4:

- The first replica, represented by the color pink, is placed in Node 2 of AZ_2
- The second replica, represented by the color blue, is placed in Node 1 of AZ_2
- The third replica, represented by the color purple, is placed in Node 2 of AZ_1

Each group in the illustration below consists of one replica from all of the four partitions (P_1, P_2, P_3, P_4)



The replica distribution: Each node contains one primary partition and two replicas

In the above illustration, the primary replica for P_1 is indicated by the dark purple color, the primary replica for P_2 is represented by the dark blue color, and the primary replica for P_3 and P_4 is represented by the dark pir color.

Now that we have completed replication, let's see how indexing and searching are performed in these replicas.

Indexing with replicas

From the diagram above, we assume that each partition is forwarded to each replica for index computation. Let's look at the example where we want to index partition P_1. This means that the same partition will be forwarded to all three replicas in both availability zones. Therefore, each node will compute the index simultaneously and reach the same state.

The advantage of this strategy is that the indexing operation will not suffer if the primary node fails.

Searching with replicas

We have three copies of each partition's index. The load balancer chooses one of the three copies of each partition to perform the query. An increased number of copies improves the scalability and availability of the system. Now, the system can handle three times more queries in the same amount of time.

Summary

In this lesson, we learned how to handle a large number of data, and a large number of queries with these strategies:

- Parallel indexing and searching, where both of these processes are colocated on the same nodes.
- Replicating each partition, which means that we replicate the indexing and searching process as well.

We successfully designed a system that scales with read (search) and write (indexing) operations colocated on the same node. But, this scaling method

brings some drawbacks. We'll look into the drawbacks and their solutions in