Scaling Search and Indexing

Learn an efficient way to scale indexing and search in a search system.

We'll cover the following

Problems with the proposed design

Although the proposed design in the previous lesson seems reasonable, still, there are a couple of serious drawbacks. We'll discuss these drawbacks below:

1. **Colocated indexing and searching**: We've created a system that colocates indexing and searching on the same node. Although it seems like efficient usage of resources, it has its downsides as well. Searching and indexing are both resource-intensive operations. Both operations impact the performance of each other. Also, this colocated design doesn't scale efficiently with varying indexing and search operations over time. Colocating both these operations on the same machine can lead to an imbalance, and it results in scalability issues.
2. **Index recomputation**: We assume that each replica will compute the index individually, which leads to inefficient usage of resources.

Furthermore, index computation is a resource-intensive task with possibly hundreds of stages of pipelined operations. Thus, recomputing the same index over different replicas requires powerful machines. Instead, the logical approach is to compute the index once and replicate it across availability zones.

Because of these key reasons, we'll look at an alternative approach for distributed indexing and searching.

Solution

Rather than recomputing the index on each replica, we compute the inverted index on the primary node only. Next, we communicate the inverted index (binary blob/file) to the replicas. The key benefit of this approach is that it avoids using the duplicated amount of CPU and memory for indexing on replicas.

Point to Ponder

Question

What are the disadvantages of the above-proposed solution?

Show Answer

Separate the indexing and search

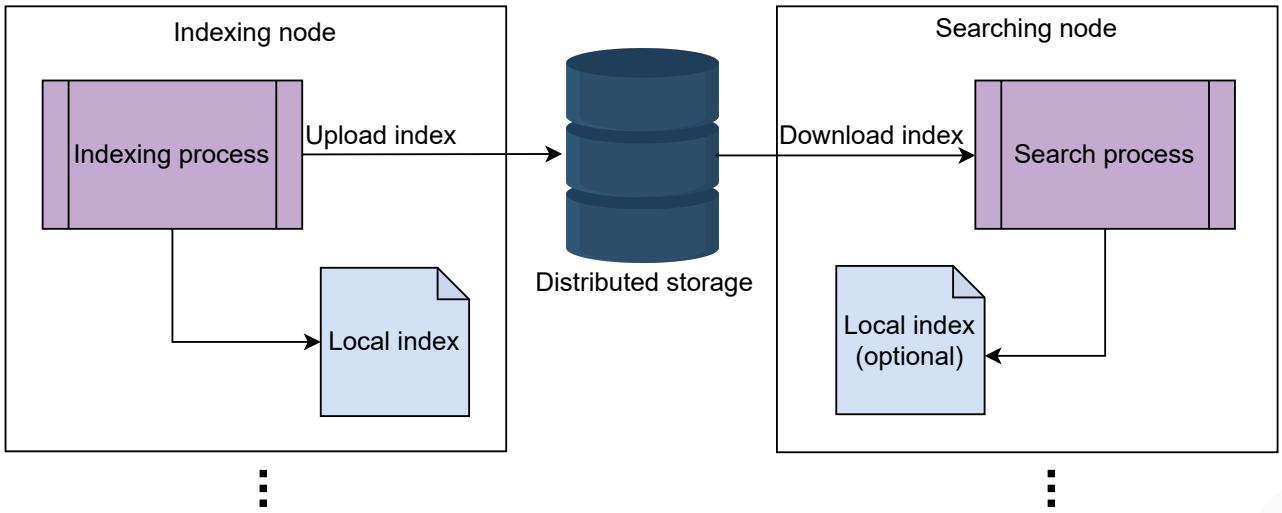
With the advent of networking and virtualization technologies, cloud computing has emerged as a successful technology. We have access to

massive amounts of bandwidth (up to 100 Gbps) and scalable distributed storage in such a technology. These advancements allow for a strong separation between indexing and search without the negative consequence of indexing latency. Because of this isolation, indexing wouldn't affect search scalability and vice versa. Also, instead of recomputing the index on the replica nodes, which wastes resources, we can just replicate the index files.

We'll use these technologies to redesign our distributed indexing and searching system. There are three components involved in this search system design:

1. **Indexer**: It consists of a group of nodes to compute the index.
2. **Distributed storage**: This system is used to store partitions and the computed index.
3. **Searcher**: It consists of a number of nodes to perform searching.

The illustration below depicts the generation and transfer of an inverted index between an indexer and a searcher node:



The indices produced by the indexing nodes are stored on the distributed storage, and the nodes involved in the search reads indices from the distributed storage to produce a result for the user's query

In the above illustration, a single node is shown for each indexing and searching operation. But, in reality, there would be an N number of nodes in the indexing phase, one node per partition (set of documents), that produces inverted indices. The inverted index is stored in the form of binary files on the nodes' local storage. Caching these blob files will result in performance improvement. These binary files are also pushed to a distributed storage. In the case of a hardware failure, a new searcher or indexer machine is added, and a copy of the data is retrieved from the distributed storage.

When the upload is complete, the searcher nodes download the index files. Depending upon user search patterns, the searching nodes will maintain a cache of frequently asked queries and serve data from RAM. A user search query will be extended to all searcher nodes, which will generate responses according to their respective indices. A merger node in the front-end servers will combine all search results and present them to the user.

The indexing process indexes the new documents as soon as they are available. At the same time, the searcher nodes fetch the updated indices to provide improved search results.

Indexing explained

Until now, we have explained the development of a highly scalable and performant design using low-cost nodes. However, we are unaware of the internals of the indexing nodes. In this section, we'll learn how indexing is performed with a MapReduce distributed model and parallel processing framework.

The **MapReduce framework** is implemented with the help of a cluster manager and a set of worker nodes categorized as Mappers and Reducers. As indicated by its name, MapReduce is composed of two phases:

1. The Map phase
2. The Reduction phase

Furthermore, the input to MapReduce is a number of partitions, or set of documents, whereas its output is an aggregated inverted index.

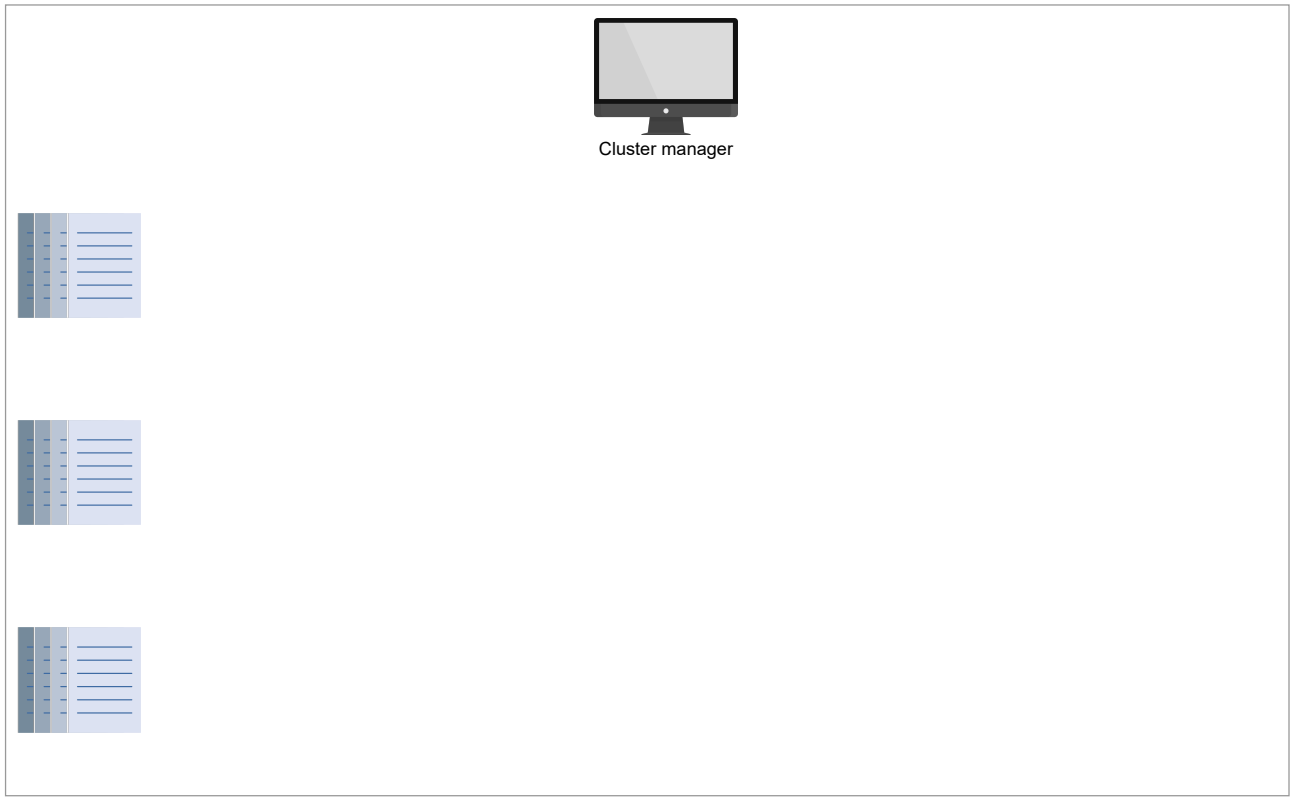
Let’s understand the purpose of the above components:

- **Cluster manager**: The manager initiates the process by assigning a set of partitions to Mappers. Once the Mappers are done, the cluster manager assigns the output of Mappers to Reducers.
- **Mappers**: This component extracts and filters terms from the partitions assigned to it by the cluster manager. These machines output inverted indexes in parallel, which serve as input to the Reducers.
- **Reducers**: The reducer combines mappings for various terms to generate a summarized index.

The cluster manager ensures that all worker nodes are efficiently utilized in the cluster. The MapReduce is built to work under partial failures. If one node fails, it reschedules the work on another node.

Note that the Reducers cannot start as long as the Mappers are working. This means that the cluster manager can use the same node as a Mapper as well as a Reducer.

The slides below depict a simplified setup of how MapReduce can be used to generate an inverted index:



A cluster manager and document partitions

1 of 5

To keep it simple, we have just shown two indicators for each term in the above illustration: the list of documents in which the term appears and the list of the frequency of the term in each document (refer to Indexing for details).

> **Note:** The above MapReduce setup is a simplified version of what happens in practice. A complex pipeline of the MapReduce framework is required to manage the complexities of a real-world

> search engine. However, the fundamental principles are the same as we presented here.

Summary

In this lesson, we have resolved two key problems of scalability (due to colocated indexing and searching) and resource wastage (due to index recomputation) by using dedicated nodes for indexing and searching. Both operations rely on distributed storage. Furthermore, we presented a