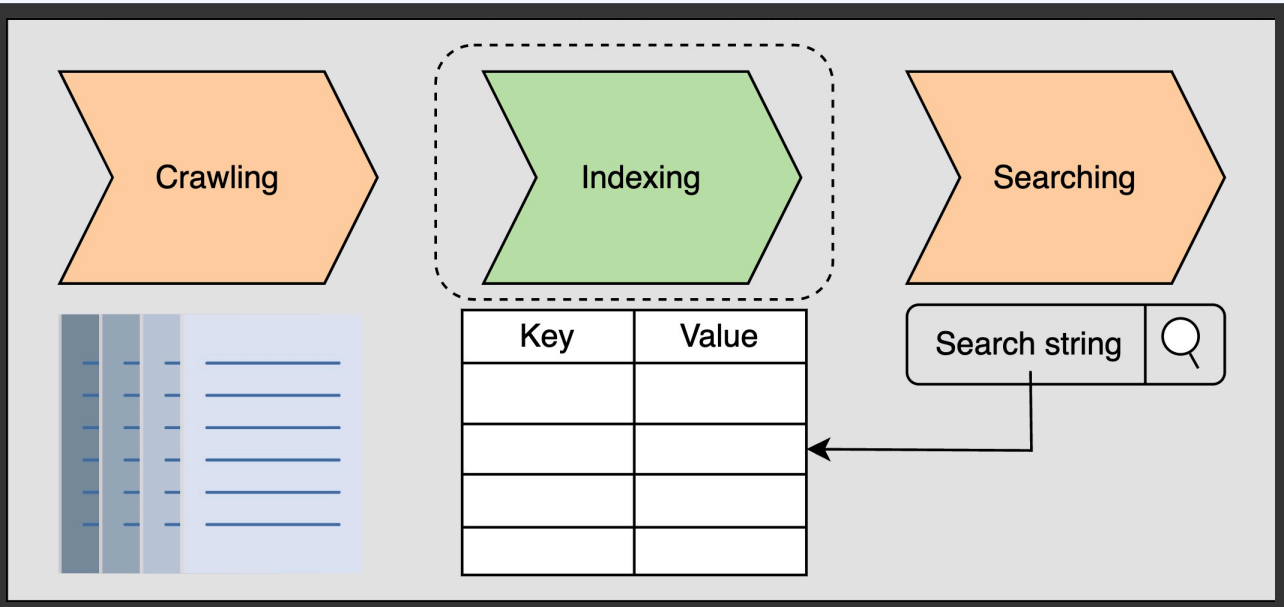
Why do we need a search system?

Nowadays, we see a search bar on almost every website. We use that search bar to pick out relevant content from the enormous amount of content available on that website. A search bar enables us to quickly find what we're looking for. For example, there are plenty of courses present on the Educative website. If we didn't have a search feature, users would have to scroll through many pages and read the name of each course to find the one they're looking for.

We have billions of websites on the Internet. Each website has many web pages and there is plenty of content on each of these web pages. With so much content, the Internet would practically be useless without search engines, and users would end up lost in a sea of irrelevant data

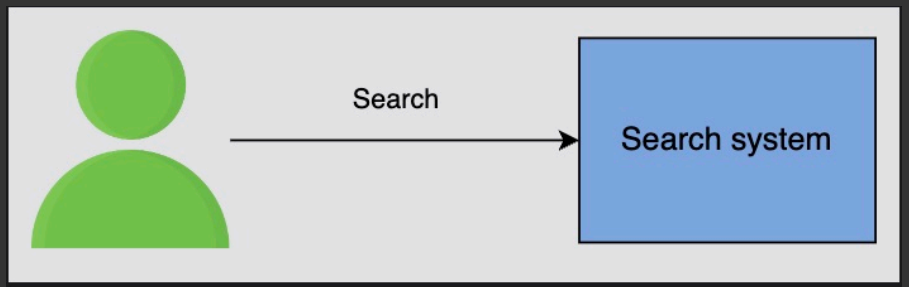
A search system is a system that takes some text input, a search query, from the user and returns the relevant content in a few seconds or less. There are three main components of a search system, namely:

A crawler, which fetches content and creates documents.
An indexer, which builds a searchable index.
A searcher, which responds to search queries by running the search query on the index created by the indexer.



Requirements of a distributed search system's Design

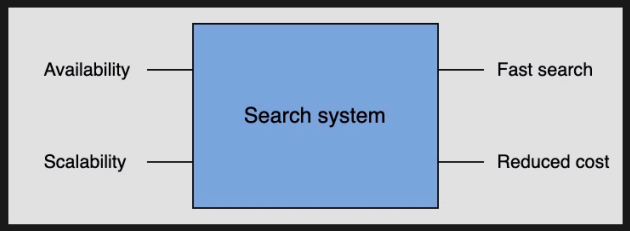
```
Functional requirements
The following is a functional requirement of a distributed search system:

Search: Users should get relevant content based on their search queries.
```



The functional requirement of a distributed search system

Non-Functional Requirement

- **Availability**: The system should be highly available to the users.
- **Scalability**: The system should have the ability to scale with the increasing amount of data. In other words, it should be able to index a large amount of data.
- **Fast search on big data**: The user should get the results quickly, no matter how much content they are searching.
- **Reduced cost**: The overall cost of building a search system should be less.



The non-functional requirement of a distributed search system

Number of servers estimation

To estimate the number of servers, we need to know how many daily active users per day are using the search feature on YouTube and how many requests per second our single server can handle. We assume the following numbers:

- The number of daily active users who use the search feature is three million.
- The number of requests a single server can handle is 1,000.

The number of servers required is calculated using this formula:

$$\frac{Number\ of\ active\ users}{queries\ handled\ per\ server} = 3K\ servers$$

Storage estimation

Each video's metadata is stored in a separate JSON document. Each document is uniquely identified by the video ID. This metadata contains the title of the video, its description, the channel name, and a transcript. We assume the following numbers for estimating the storage required to index one video:

- The size of a single JSON document is 200 KB.
- The number of unique terms or keys extracted from a single JSON document is 1,000.
- The amount of storage space required to add one term into the index table is 100 Bytes.

The following formula is used to compute the storage required to index one video:

$$Total_{storage/video} = Storage_{/doc} + (Terms_{/doc} \times Storage_{/term})$$

Total Storage Required to Index One Video on YouTube

Storage per JSON doc (KB)	No. of terms per doc	Storage per term (Bytes)	Total storage per video (KB)
200	1000	100	300

Total Storage Required to Index Videos per Day on YouTube

No. of videos per day	Total storage per video (KB)	Total storage per day(GB)
6000	300	1.8

Bandwidth estimation

The data is transferred between the user and the server on each search request. We estimate the bandwidth required for the incoming traffic on the server and the outgoing traffic from the server. Here is the formula to calculate the required bandwidth:

$$Total_{bandwidth} = Total_{requests\_second} \times Total_{query\_size}$$

Incoming traffic

To estimate the incoming traffic bandwidth, we assume the following numbers:

- The number of search requests per day is 150 million.
- The search query size is 100 Bytes.

We can use the formula given above to calculate the bandwidth required for the incoming traffic.

Bandwidth Required for Incoming Search Queries per Second

No. of requests per second	Query size (Bytes)	Bandwidth (Mb/s)
1736.11	100	*f* 1.39

Outgoing traffic

Outgoing traffic is the response that the server returns to the user on the search request. We assume that the number of suggested videos against a search query is 80, and one suggestion is of the size 50 Bytes. Suggestions consist of an ordered list of the video IDs.

To estimate the outgoing traffic bandwidth, we assume the following numbers:

- The number of search requests per day is 150 million.
- The response size is 4,000 Bytes.

We can use the same formula to calculate the bandwidth required for the outgoing traffic.

Bandwidth Required for Outgoing Traffic per Second

No. of requests per second	Query size (Bytes)	Bandwidth (Mb/s)
1736.11	4000	*f* 55.56

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, number of documents we have & size of documents.

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.

```
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.
```

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.

Inverted Index

Term	Mapping ([doc], [freq], [[loc])
elasticsearch	([1, 2, 3], [1, 1, 1], [[1], [1], [1]])
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.

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.

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

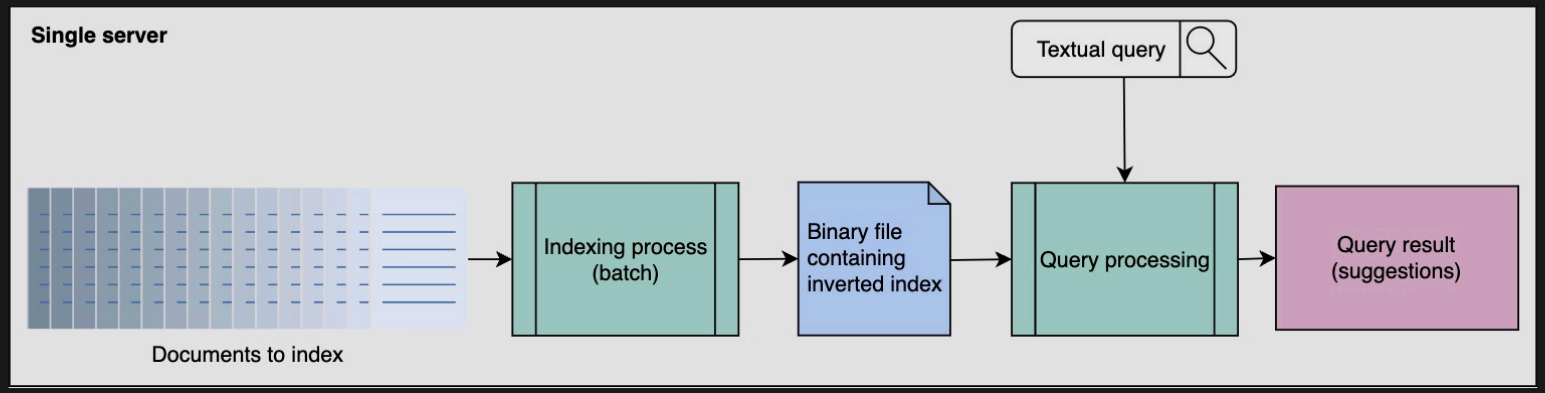
Hide Answer

It probably wouldn't work to return all the documents that are found. Instead, we should sort them based on the relevance to the search query. The top results should be returned to the user, instead of returning all the documents.

- **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.

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.

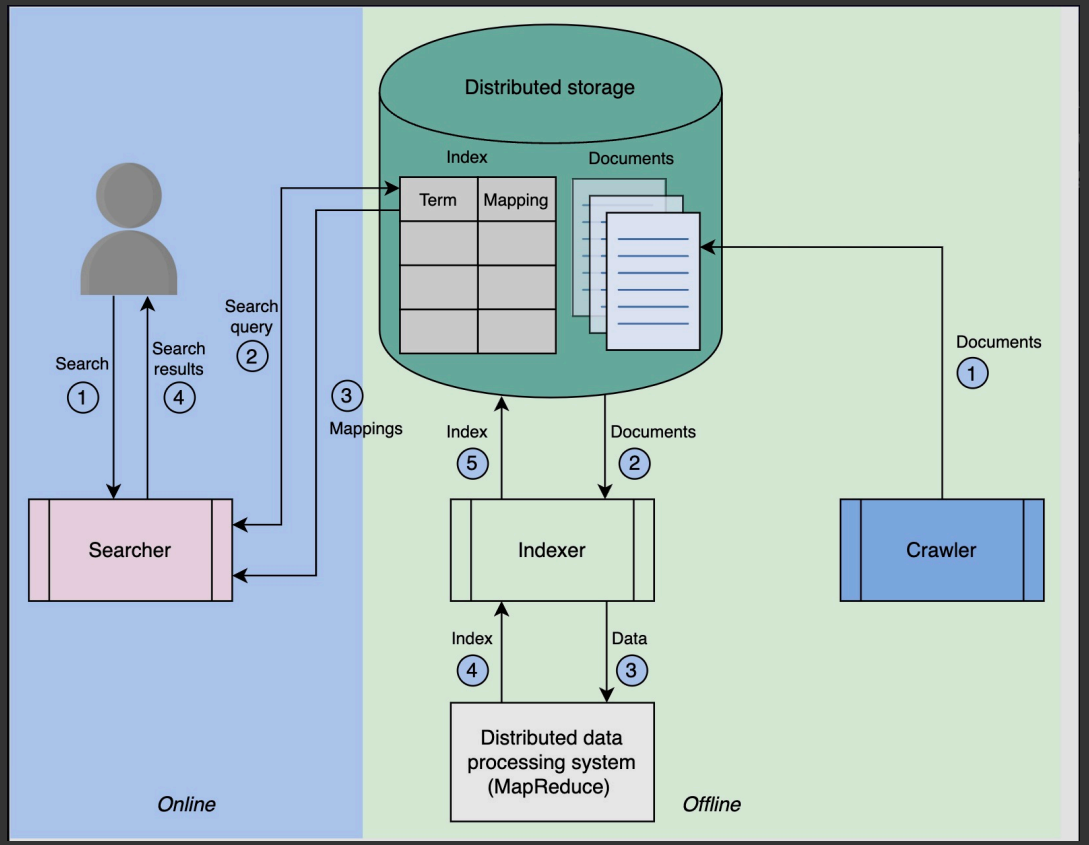
Server overload: If numerous users perform queries and the queries are complicated, it stresses the server (node).

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.

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.

Design of a distributed system

→ High level design



- 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

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

```
search(query)
```

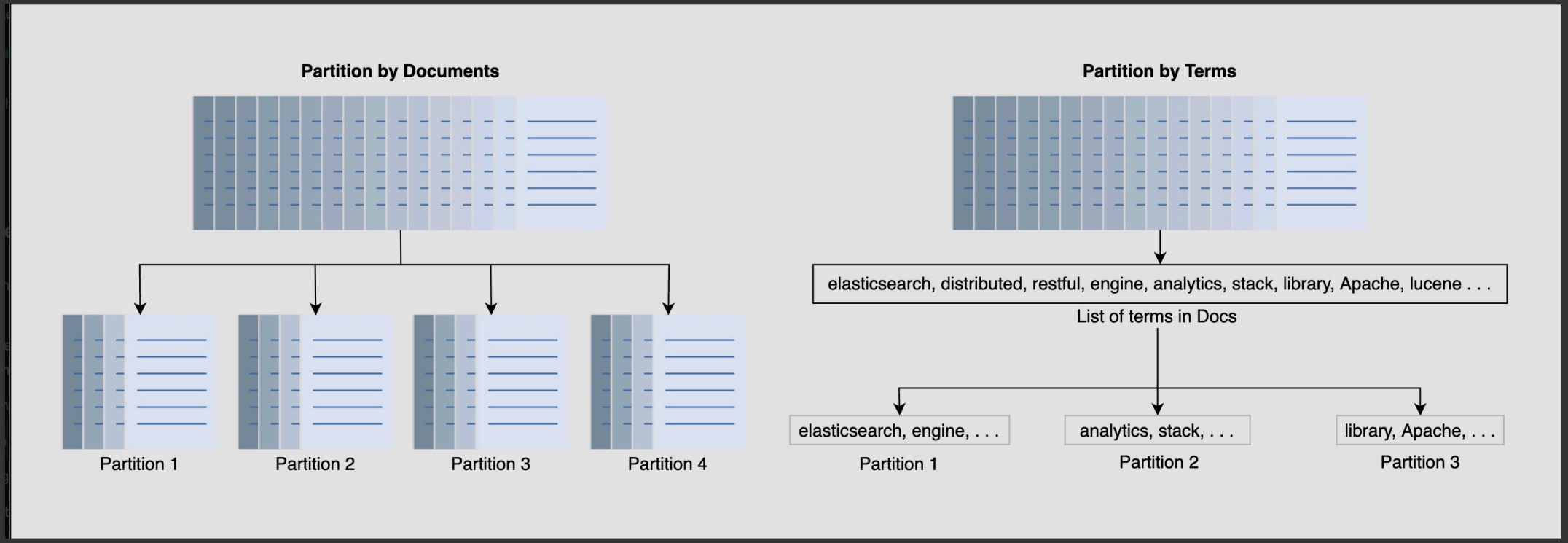
Parameter	Description
`query`	This is the textual query entered by the user in the search bar, based on which the results are found.

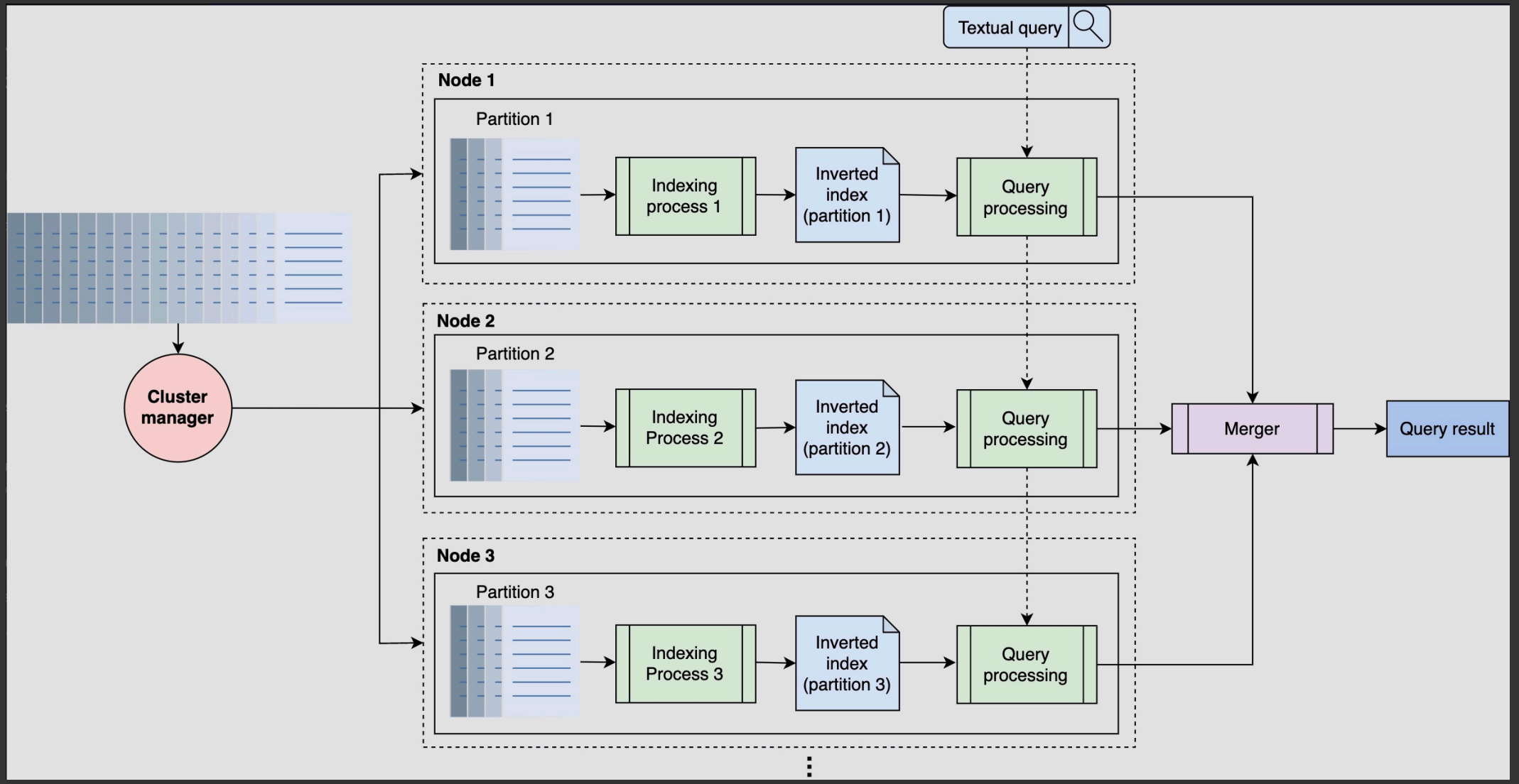
Since the indexer is the core component in a search system, we discussed an indexing technique and the problems associated with centralized indexing in the previous lesson. In this lesson, we consider a distributed solution for indexing and searching.

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.”




Textual query
Node 1
Partition 1
Indexing process 1
Inverted index (partition 1)
Query processing
Node 2
Partition 2
Indexing Process 2
Inverted index (partition 2)
Query processing
Node 3
Partition 3
Indexing Process 3
Inverted index (partition 3)
Query processing
Cluster manager
Merger
Query result

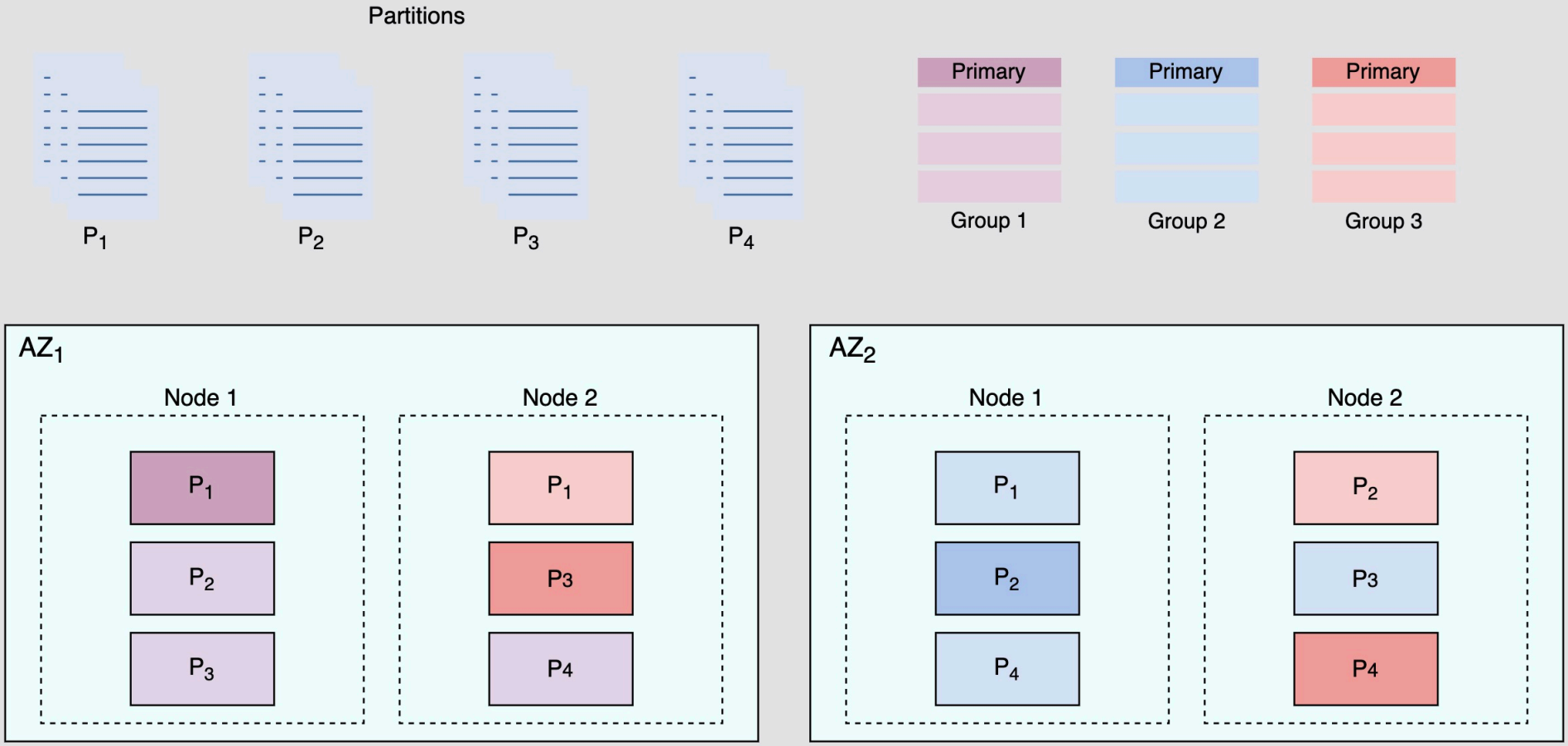
Replication

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.

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.



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.

Scaling Search & Indexing

- Colocated indexing & searching
- Index recomputations

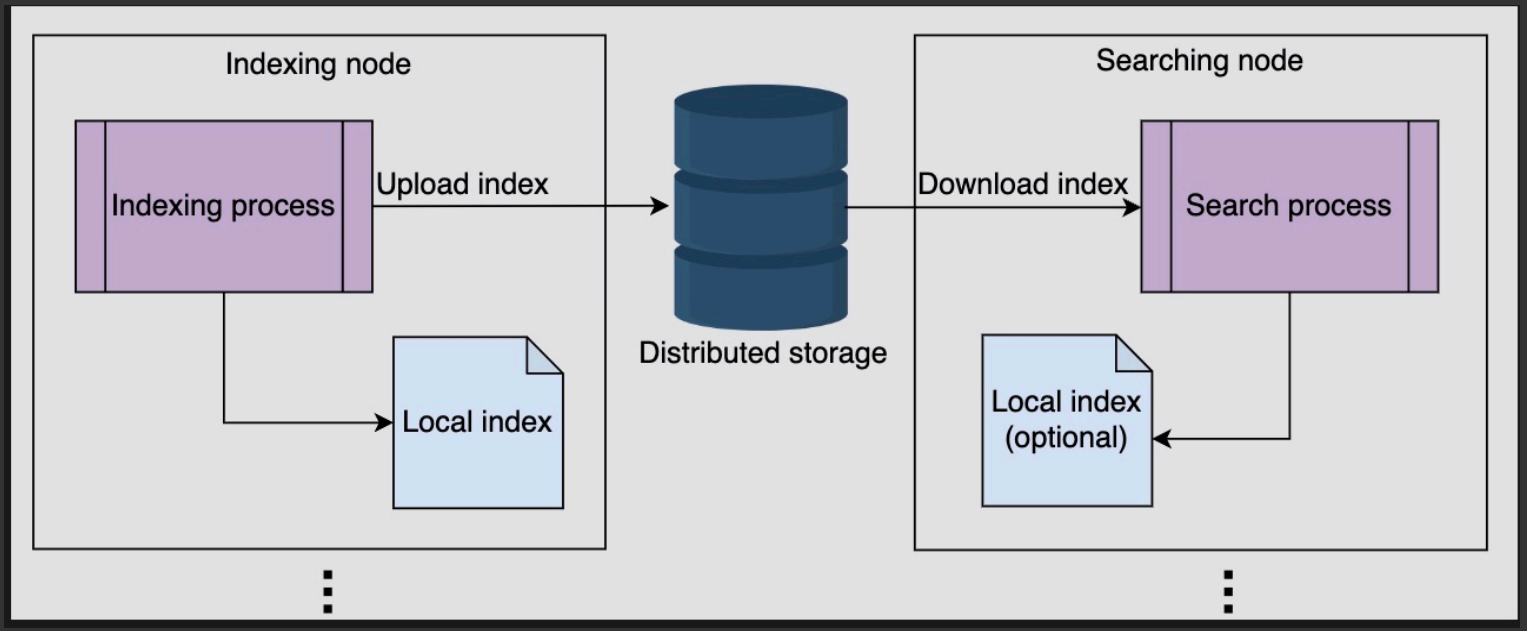
Rather than recomputing the index on each replica, we compute the inverted index on the primary node only. Next, we communicate the inverted index (binary blob/file) to the replicas. The key benefit of this approach is that it avoids using the duplicated amount of CPU and memory for indexing on replicas.

What are the disadvantages of the above-proposed solution?

Hide Answer
Since the inverted index will be transferred to the replicas, this will introduce a transmission latency to copy the inverted index file because the size of the index file can be very large.

When the primary node receives new indexing operations, the inverted index file changes. Each replica needs to fetch the latest version of the file after a certain amount of indexing operations reaches a defined threshold.

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.

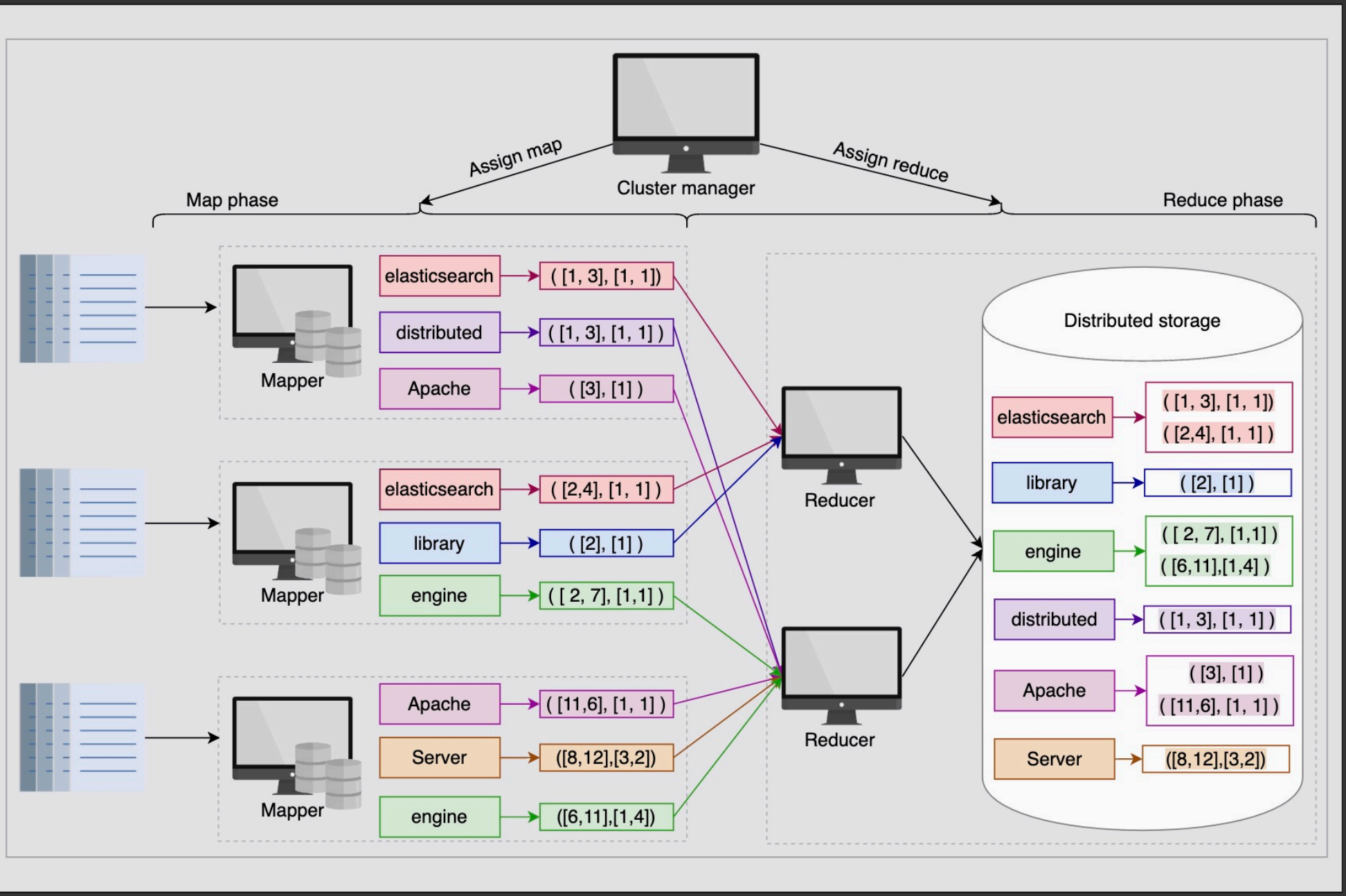


Indexing Explained

The map reduced framework is implemented with help of cluster manager. Set of worker nodes categorized as mappers & Reducers.

1. The map phase
2. The Reduction phase.

- **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.


Cluster manager
Assign map
Assign reduce
Map phase
Reduce phase
Mapper
Mapper
Mapper
elasticsearch
distributed
Apache
elasticsearch
library
engine
Apache
Server
engine
([1, 3], [1, 1])
([1, 3], [1, 1])
([3], [1])
([2,4], [1, 1])
([2], [1])
([2, 7], [1,1])
([11,6], [1, 1])
([8,12],[3,2])
([6,11],[1,4])
Reducer
Reducer
Distributed storage
elasticsearch
([1, 3], [1, 1])
([2,4], [1, 1])
library
([2], [1])
engine
([2, 7], [1,1])
([6,11],[1,4])
distributed
([1, 3], [1, 1])
Apache
([3], [1])
([11,6], [1, 1])
Server
([8,12],[3,2])

Evaluation of a distributed search's Design

Availability
We utilized distributed storage to store these items:

Documents crawled by the indexer.
Inverted indexes generated by the indexing nodes.

Data is replicated across multiple regions in distributed storage, making cross-region deployment for indexing and search easier. The group of indexing and search nodes merely needs to be replicated in different availability zones.

Scalability
Partitioning is an essential component of search systems to scale. When we increase the number of partitions and add more nodes to the indexing and search clusters, we can scale in terms of data indexing and querying.

Fast search on big data
We utilized a number of nodes, each of which performs search queries in parallel on smaller inverted indices.

Reduced cost
We used cheaper machines to compute indexes and perform searches. If one node fails, we don't have to recompute the complete index. Instead, some of the documents need to be indexed again.