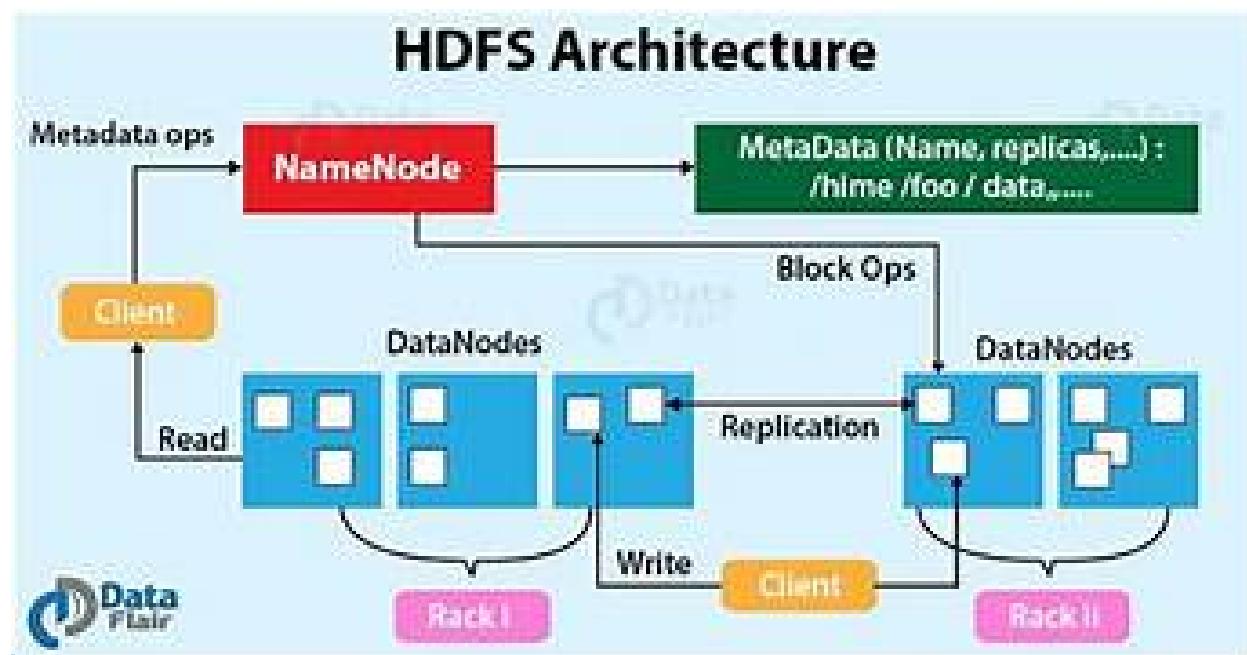


Explain the architecture and design of HDFS.

Explain the architecture and design of HDFS.

Architecture and Design of HDFS (Hadoop Distributed File System)

HDFS (Hadoop Distributed File System) is the storage layer of Hadoop, designed to store and manage **very large files** across a cluster of machines. It follows a **master-slave architecture** and is optimized for **high throughput, scalability, and fault tolerance**, rather than low-latency access.



Architecture and Design of HDFS (Short Version)

1. Architecture of HDFS

HDFS follows a **master-slave architecture**.

NameNode (Master)

- Manages file system metadata
- Maintains file names, directories, and block locations
- Controls file operations (open, close, delete)
- Does not store actual data
- Acts as the single point of control

DataNode (Slave)

- Stores actual data blocks
- Performs read and write operations
- Sends heartbeats and block reports to NameNode
- Handles block replication and deletion

Secondary NameNode

- Assists the NameNode
 - Periodically merges Edit logs and Fslimage
 - Reduces NameNode workload
 - Not a backup of the NameNode
-

2. Design of HDFS

- **Large Block Size**
 - Default block size is 128 MB
 - Improves throughput and reduces disk seeks
- **Data Replication**
 - Default replication factor is 3
 - Blocks stored on multiple DataNodes
 - Provides fault tolerance and high availability

- **Write-Once, Read-Many**
 - Files are written once and read multiple times
 - Simplifies consistency management
 - **Fault Tolerance**
 - Detects DataNode failures automatically
 - Re-replicates missing blocks
 - **Data Locality**
 - Processing is done near data location
 - Reduces network overhead
 - **Scalability**
 - Supports horizontal scaling
 - Can store petabytes of data
-

3. HDFS Read and Write Operations (Brief)

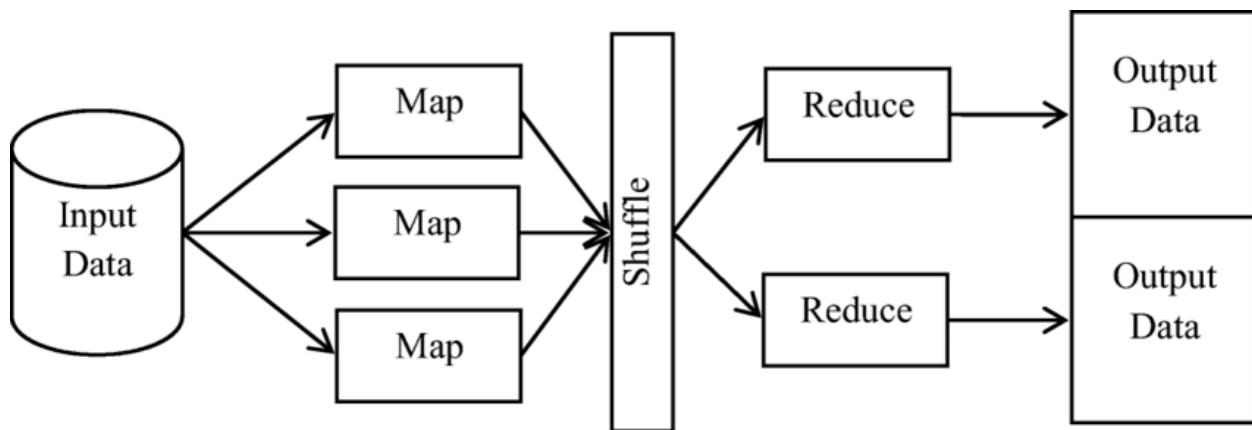
- **Write Operation**
 - Client requests NameNode
 - NameNode provides DataNode locations
 - Data is written and replicated
 - **Read Operation**
 - Client gets block locations from NameNode
 - Reads data directly from DataNodes
-

Advantages of HDFS

- High fault tolerance
- High throughput
- Cost-effective

- Suitable for very large datasets

Describe the MapReduce programming model and job execution flow



MapReduce Programming Model and Job Execution Flow

MapReduce is a distributed programming model used in Hadoop to process **large datasets in parallel** across a cluster of machines. It simplifies Big Data processing by dividing a job into independent tasks that run simultaneously on different nodes, ensuring scalability and fault tolerance.

1. MapReduce Programming Model

The MapReduce model consists of two main functions:

1.1 Map Function

- Takes input data as **key-value pairs**
- Processes input split independently

- Produces intermediate **key-value pairs**
- Runs in parallel on multiple nodes

Example:

Input: (line_number, text_line)

Output: (word, 1)

1.2 Reduce Function

- Takes intermediate key–value pairs from Map phase
- Groups values by key
- Performs aggregation or computation
- Produces final output

Example:

Input: (word, [1,1,1])

Output: (word, 3)

2. MapReduce Job Execution Flow

Step 1: Job Submission

- Client submits MapReduce job to the Hadoop cluster
- Job includes:
 - Mapper class
 - Reducer class
 - Input and output paths

Step 2: Input Splitting

- Input data stored in HDFS
- Data is divided into **input splits**

- Each split is processed by one Map task
-

Step 3: Map Phase

- Map tasks run on nodes where data is located (data locality)
 - Mapper processes input splits
 - Generates intermediate key–value pairs
-

Step 4: Shuffle and Sort

- Intermediate data is:
 - Transferred from Mappers to Reducers
 - Sorted by key
 - Ensures all values of the same key go to the same Reducer
-

Step 5: Reduce Phase

- Reducers process grouped data
 - Perform aggregation or computation
 - Generate final output key–value pairs
-

Step 6: Output Storage

- Final output is written back to **HDFS**
 - Stored in output files (one per Reducer)
-

3. Advantages of MapReduce

- Parallel and distributed processing
- High scalability
- Fault tolerance
- Simple programming model

MapReduce Program for Word Count Application (Python)

The **Word Count** application is a basic MapReduce program used to count the number of occurrences of each word in a given input file. It demonstrates how data is processed through different phases of the MapReduce model.

Program Code

Mapper (mapper.py)

```
import sys
for line in sys.stdin:
    for word in line.split():
        print(word, 1)
```

Reducer (reducer.py)

```
import sys
from itertools import groupby

for word, group in groupby(line.split()[0] for line in sys.stdin):
    print(word, sum(1 for _ in group))
```

Explanation of Each Step

1. Input

- Input is a text file stored in HDFS
- Example input:

```
BigData Big
```

- Each line of the file is provided as input to the Mapper

2. Mapping Phase

- Mapper reads input **line by line**
- Each line is split into words
- For every word, Mapper outputs a key–value pair:

```
(word,1)
```

Mapper Output for input “Big Data Big”:

```
(Big,1)  
(Data,1)  
(Big,1)
```

3. Shuffle and Sort Phase

- Automatically handled by Hadoop
- Groups all intermediate key–value pairs by key
- Sorts words alphabetically

- Sends grouped data to the Reducer

After Shuffle & Sort:

```
(Big,[1,1])  
(Data,[1])
```

4. Reducing Phase

- Reducer receives grouped key–value pairs
- Counts the number of values for each word
- Produces the final output

Reducer Output:

```
Big2  
Data1
```

Final Output

- Stored in HDFS as output files
- Contains each word and its total count

Conclusion

This Python-based MapReduce Word Count program illustrates how Hadoop processes large text data by dividing the work into mapping, shuffle and sort, and reducing phases, enabling parallel and efficient word frequency analysis.

Explain the role of Combiners, Partitioners, and

Counters in Hadoop.

1. Combiner

A Combiner is a local reducer that runs after the Map phase and before the Reduce phase.

Role of Combiner

- Performs **local aggregation** of Mapper output
- Reduces the amount of data transferred over the network
- Improves performance and efficiency

Key Points

- Runs on the **Mapper node**
- Optional component
- Uses same logic as Reducer (in most cases)
- Does not guarantee execution every time

Example

- In Word Count:
 - Mapper output: `(word,1),(word,1),(word,1)`
 - Combiner output: `(word,3)`

2. Partitioner

A Partitioner decides how intermediate key-value pairs are distributed to Reducers.

Role of Partitioner

- Assigns Mapper output keys to specific Reducers

- Ensures **all values of the same key go to the same Reducer**
- Enables parallel processing across Reducers

Key Points

- Runs between Map and Reduce phases
- Default partitioner: **HashPartitioner**
- Custom partitioners can be defined for better load balancing

Example

- Words starting with A–M → Reducer 1
- Words starting with N–Z → Reducer 2

3. Counters

Counters are used to collect statistics and monitor MapReduce job execution.

Role of Counters

- Track events and job progress
- Help in debugging and performance analysis
- Provide job-level metrics

Key Points

- Built-in and user-defined counters available
- Automatically aggregated across all nodes
- Accessible after job completion

Examples

- Number of input records processed
- Number of output records generated
- Count of invalid or skipped records

Comparison Summary

| Component | Purpose |
|-------------|--|
| Combiner | Reduces intermediate data size |
| Partitioner | Controls data distribution to Reducers |
| Counter | Monitors and tracks job statistics |