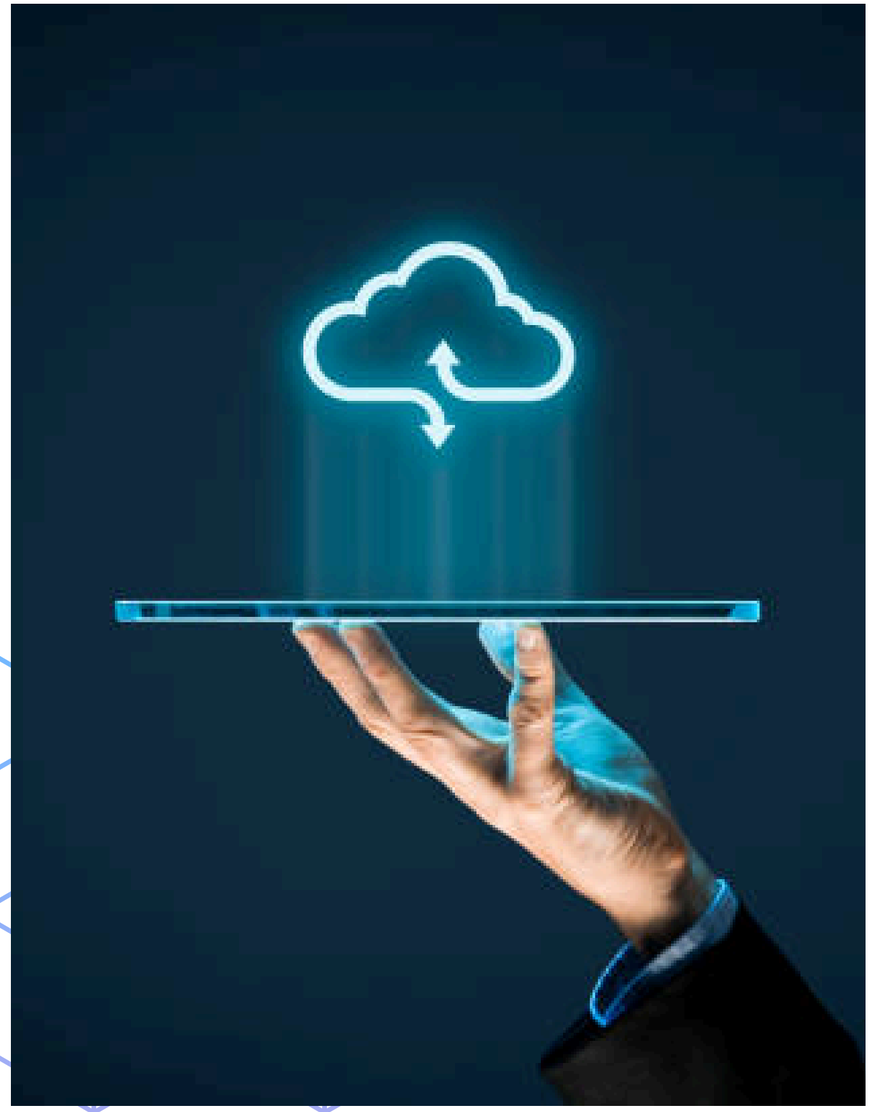
DISTRIBUTED COMPUTING

CLOUD STORAGE : MAP REDUCE

Presentation by:

Gobika R (22z220)
Sandhiya R (22z258)
Thamina anzum A (22z267)
Vijeyasri T (22z272)
Vinithaa P (22z273)



Cloud storage & Map reduce

What is Cloud Storage?

- A distributed storage model where data is housed on remote servers rather than local infrastructure.
- Enables ubiquitous access through an internet interface.
- Offers redundancy and fault tolerance, mitigating risks of data loss.
- Essential for enterprise-grade data archival, replication, and retrieval.

Types of Cloud Storage

- **Object Storage –** Stores data as objects with metadata, ideal for unstructured data like multimedia (**e.g.**, Amazon S3, Google Cloud Storage).
- **Block Storage –** Divides data into fixed-size blocks for low-latency, high-performance applications.
- **File Storage –** Uses a hierarchical structure for easy file management, common in NAS systems.

Key Features of Cloud Storage

- Elastic Scalability
- Global Accessibility
- Multi-Redundancy
- Robust Security
- Cost Optimization

Cloud Storage Providers

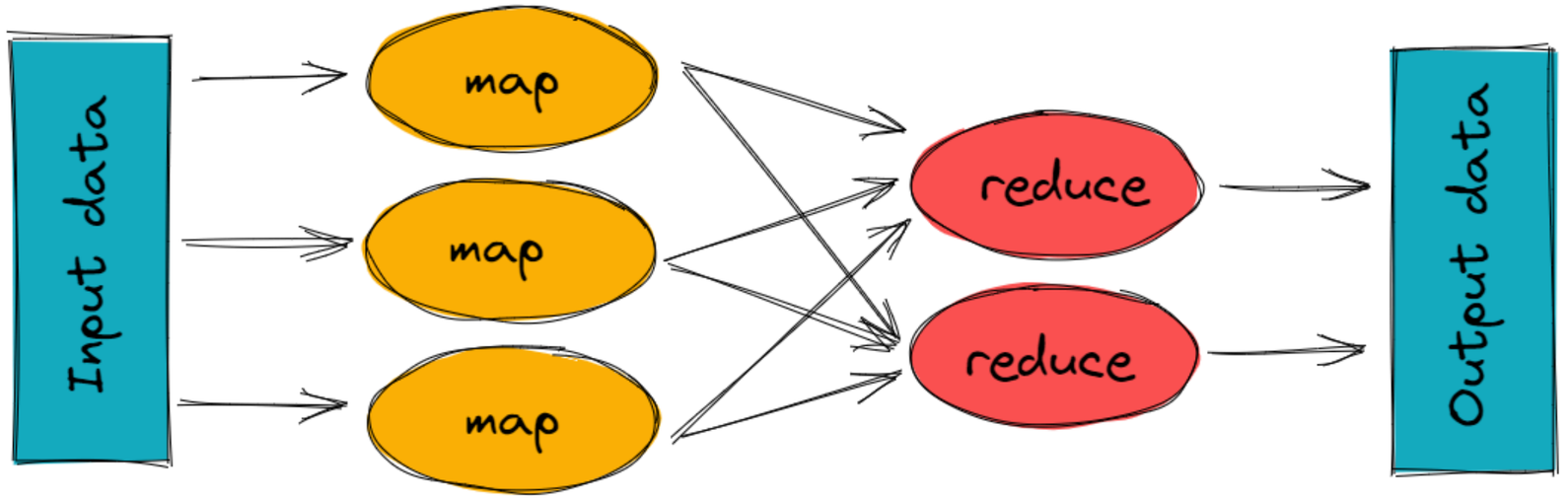
- Amazon S3
- Google Cloud Storage
- Microsoft Azure Blob Storage
- Dropbox & OneDrive

Introduction to MapReduce

- Handling large datasets requires **distributed parallelism** for efficient processing.
- MapReduce, pioneered by **Google**, facilitates fault-tolerant batch processing.
- Decomposes complex queries into independent map and reduce functions, executing them across a distributed cluster.
- Essential for **big data analytics**, real-time log processing, and predictive modeling.

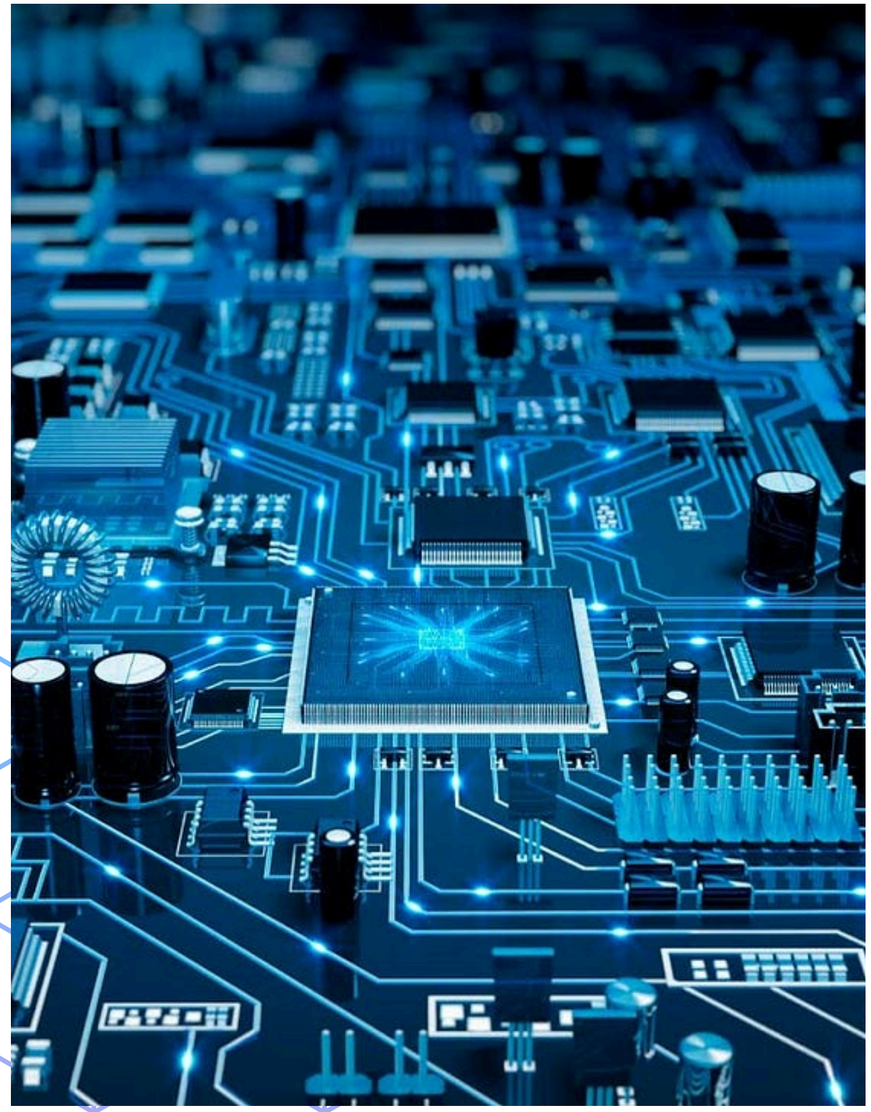
How MapReduce Works?

- **Map Phase** – Splits the input dataset into smaller key-value pairs, enabling parallel execution.
- **Shuffle & Sort Phase** – Groups, redistributes, and organizes the intermediate data for efficient aggregation.
- **Reduce Phase** – Consolidates mapped data, applying aggregation, filtering, and transformation techniques.



Real-World Applications of MapReduce

- **Big Data Analytics** – Processes large-scale user activity logs (**Google, Facebook).**
- **Search Engines** – Powers webpage indexing for efficient search **(Google Search).**
- **Financial Fraud Detection** – Detects anomalies in real-time transactions.
- **E-commerce Personalization** – Enhances AI-driven recommendations **(Amazon, Netflix, Spotify).**



Cloud storage Architecture and Technologies

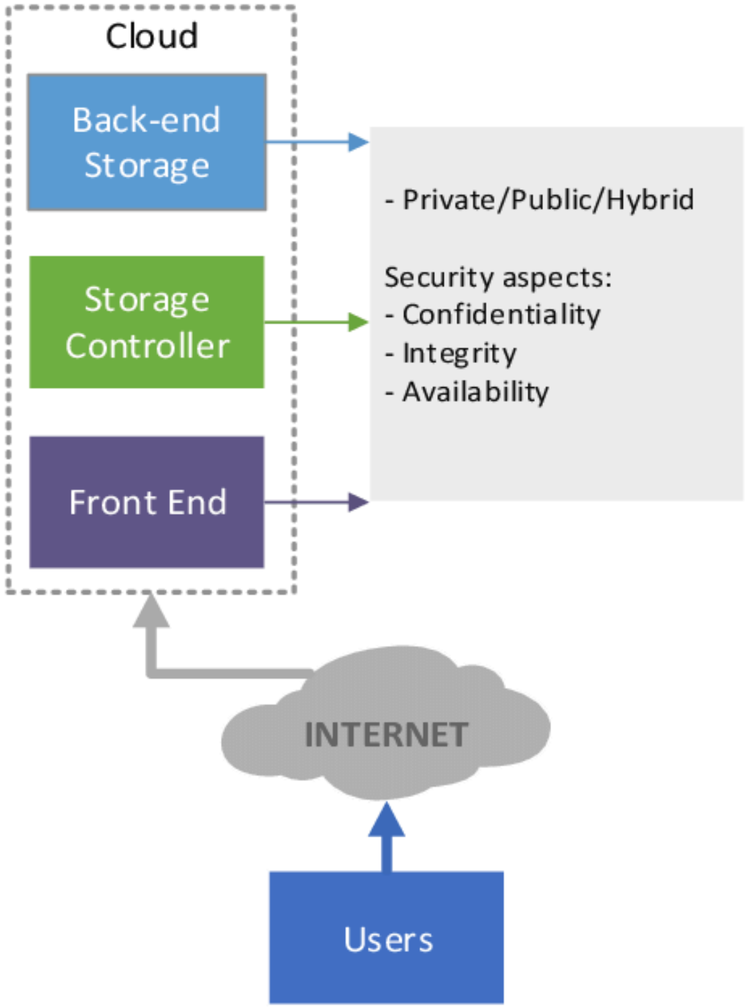
Cloud Storage Architecture

Cloud storage architecture is the framework that makes it possible for users and organizations to store, manage, and access their data online. It’s designed to handle massive amounts of data while ensuring security, accessibility, and scalability.

Main Components of Cloud Storage Architecture

1. Frontend Layer
2. Backend Layer
3. Control Layer
4. Network Layer

CLOUD STORAGE ARCHITECTURE



1. Front-End Layer

- User interface (Web apps, APIs, SDKs)
- Authentication and access control mechanisms

2. Storage Nodes & Data Center Layer(Backend Layer)

- Distributed servers and data centers
- Redundant storage mechanisms for high availability
- Storage virtualization and resource pooling

3.Management & Security Layer(Control Layer)

- Data encryption, backup, and disaster recovery
- Monitoring, auditing, and compliance tools

4.Networking & Connectivity(Network Layer)

- High-speed internet, VPN, and dedicated connections
- Content delivery networks (CDNs) for faster access

Data Consistency in Cloud Storage

What is Data Consistency?

- Ensures that all copies of stored data remain synchronized across different cloud locations.
- Prevents users from accessing outdated or conflicting data.

Why is Data Consistency Important?

- Ensures accuracy and reliability in cloud-based applications.
- Critical for banking systems, e-commerce transactions, and real-time apps.

Key Challenge:

Balancing Consistency, Availability, and Performance in distributed cloud environment

Types of Data Consistency Models

1. Strong Consistency

- Guarantees immediate update visibility across all nodes.
- Used in banking, stock trading, and financial applications.

2. Eventual Consistency

- Updates propagate over time; slight delays occur.
- Used in social media posts, collaborative apps, DNS services.

3. Causal Consistency

- Ensures operations that are related are executed in order.
- Used in distributed databases and real-time collaborative tools.

4. Read-Your-Writes Consistency

- Guarantees that a user always sees their own latest changes.
- Used in cloud storage services like Google Drive and OneDrive.

Types of Cloud Storage Technologies

1. Object Storage (e.g., AWS S3, Google Cloud Storage)

- Data stored as objects with metadata
- Ideal for unstructured data like images, videos, and backups
- Highly scalable and cost-effective

2. Block Storage (e.g., Amazon EBS, Google Persistent Disk)

- Data stored in fixed-sized blocks
- Used for databases and virtual machines
- High-performance, low-latency storage

3. File Storage (e.g., Google Drive, Dropbox, OneDrive)

- Data stored in a hierarchical folder structure
- Common for user-generated files and collaboration tools

Cloud Storage Security & Compliance

- Data Encryption
- Access Control
- Backup & Disaster Recovery
- Regulatory Compliance

Benefits of Cloud Storage

- Scalability – Adjust storage as needed
- Accessibility – Access data from any device, anywhere
- Cost Efficiency – Pay only for storage used
- Data Redundancy – Multiple copies of data ensure safety
- Collaboration – Multiple users can access and edit files

Future Trends in Cloud Storage

1. AI-Powered Storage Management

- Automated data categorization and optimization
- Intelligent caching and retrieval

2. Edge Computing & Distributed Storage

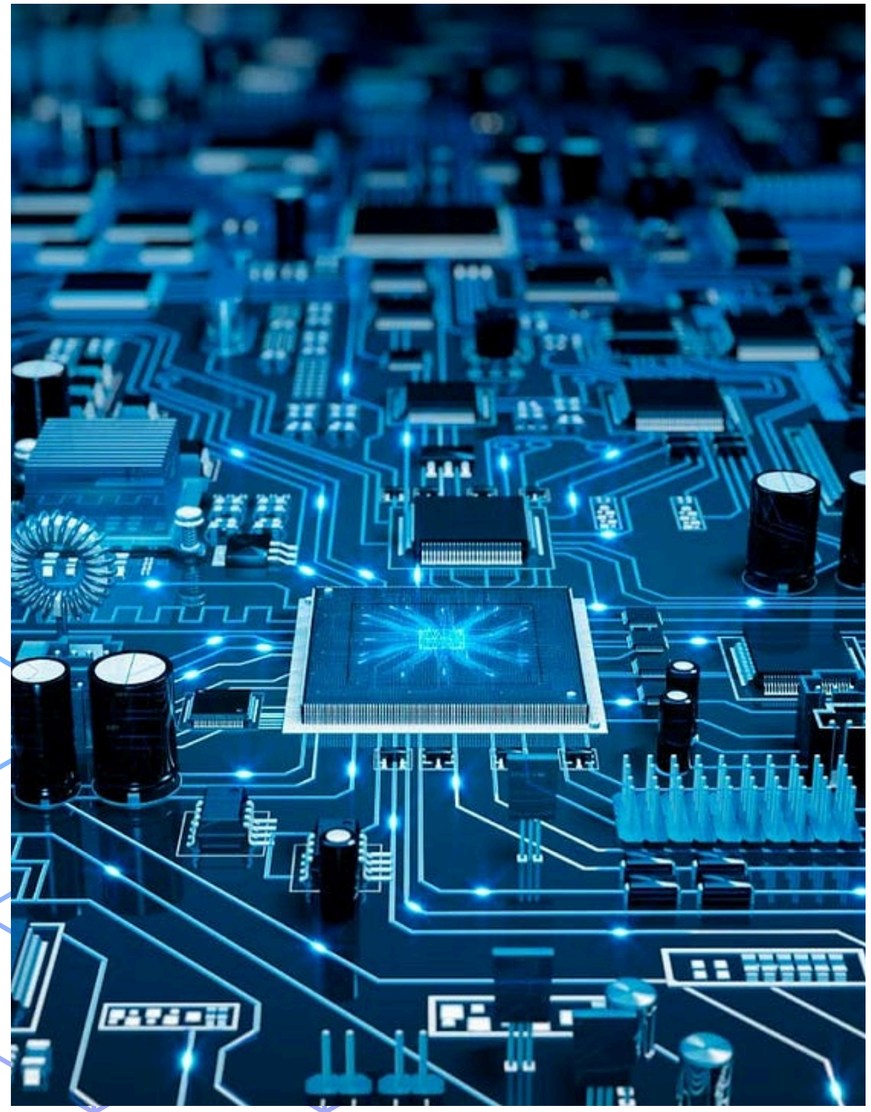
- Storing data closer to users for lower latency
- Used in IoT and 5G networks

3. Quantum Storage Technologies

- Quantum encryption for ultra-secure storage
- High-speed data retrieval

4. Focus on Security & Privacy

- Advanced encryption and Zero Trust models



Map reduce framework & workflow

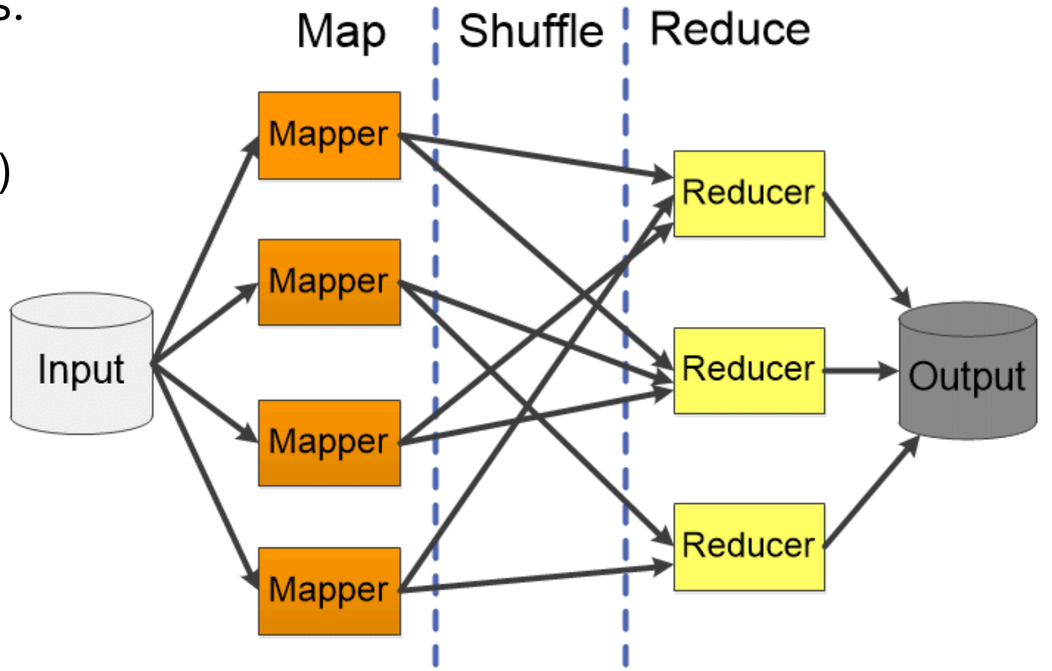
MAP REDUCE

- MapReduce is a programming model used for efficient processing in parallel over large data-sets in a distributed manner.
- The data is first split and then combined to produce the final result.
- The purpose of MapReduce in Hadoop is to Map each of the jobs and then it will reduce it to equivalent tasks.
- The MapReduce task is mainly divided into two phases:

    1. Map
    2. Reduce

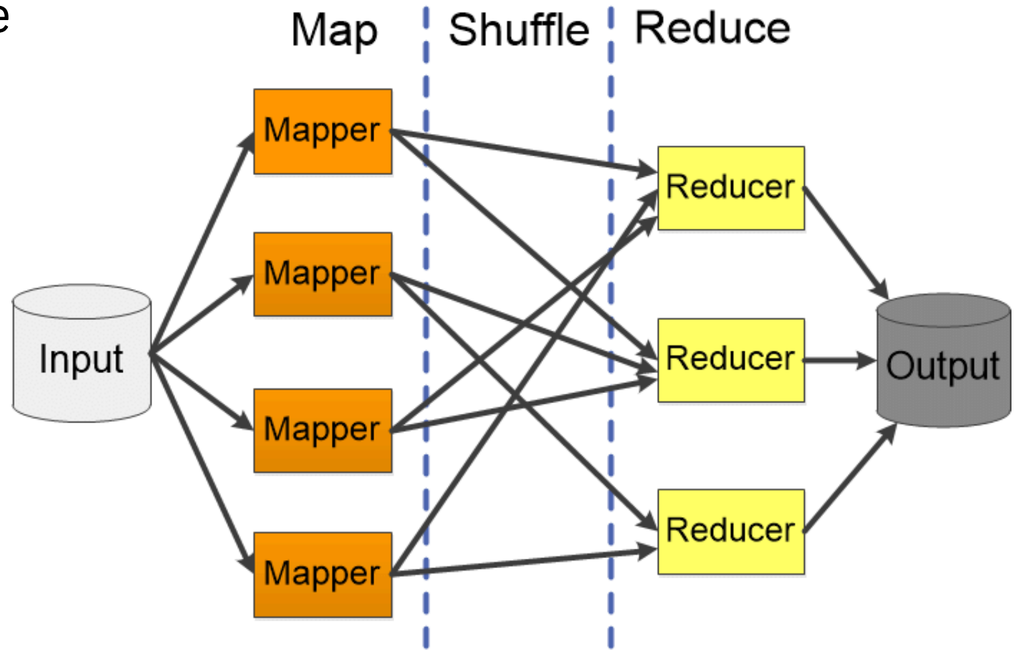
Map:

- MapReduce primarily maps input data into key-value pairs.
- The input to the Map function consists of key-value pairs, where the key can represent an identifier (e.g., an address) and the value holds the actual data.
- The **Map()** function processes each key-value pair independently in its memory repository.
- It generates **intermediate key-value pairs**, which are then passed to the **Reduce()** function.
- The **Reduce()** function aggregates and processes these intermediate key-value pairs to produce the final output.



Reduce:

- The intermediate key-value pairs from the Map phase serve as input for the **Reduce()** function.
- Key-value pairs are **shuffled and sorted** before being sent to the Reducer.
- The **Reducer aggregates or groups** the data based on keys.
- The processing in the Reduce phase follows the logic defined by the developer in the reducer algorithm.
- The final output is generated and stored in **HDFS or cloud storage** for further use.



Example : Word Count Problem

Input Data:

File1: cat dog cat

File2: dog cat dog

Map Phase

Mapper 1 Output: (cat, 1), (dog, 1), (cat, 1)

Mapper 2 Output: (dog, 1), (cat, 1), (dog, 1)

Shuffle and Sort Phase

cat → (1, 1, 1)

dog → (1, 1, 1)

Reduce Phase

Reducer Output:

cat → 3

dog → 3

Components of MapReduce Architecture:

Client: The MapReduce client is the one who brings the Job to the MapReduce for processing. There can be multiple clients available that continuously send jobs for processing to the Hadoop MapReduce Manager.

Job: The MapReduce Job is the actual work that the client wanted to do which is comprised of so many smaller tasks that the client wants to process or execute.

Hadoop MapReduce Master: It divides the particular job into subsequent job-parts.

Job-Parts: The task or sub-jobs that are obtained after dividing the main job. The result of all the job-parts combined to produce the final output.

Input Data: The data set that is fed to the MapReduce for processing.

Output Data: The final result is obtained after the processing.

Key Components of MapReduce Architecture:

Job Tracker :

- Manages all resources and jobs across the cluster.
- It schedules each Map task on the Task Tracker running on the same data node.
- Since there are multiple data nodes in the cluster, Job Tracker ensures efficient job distribution.

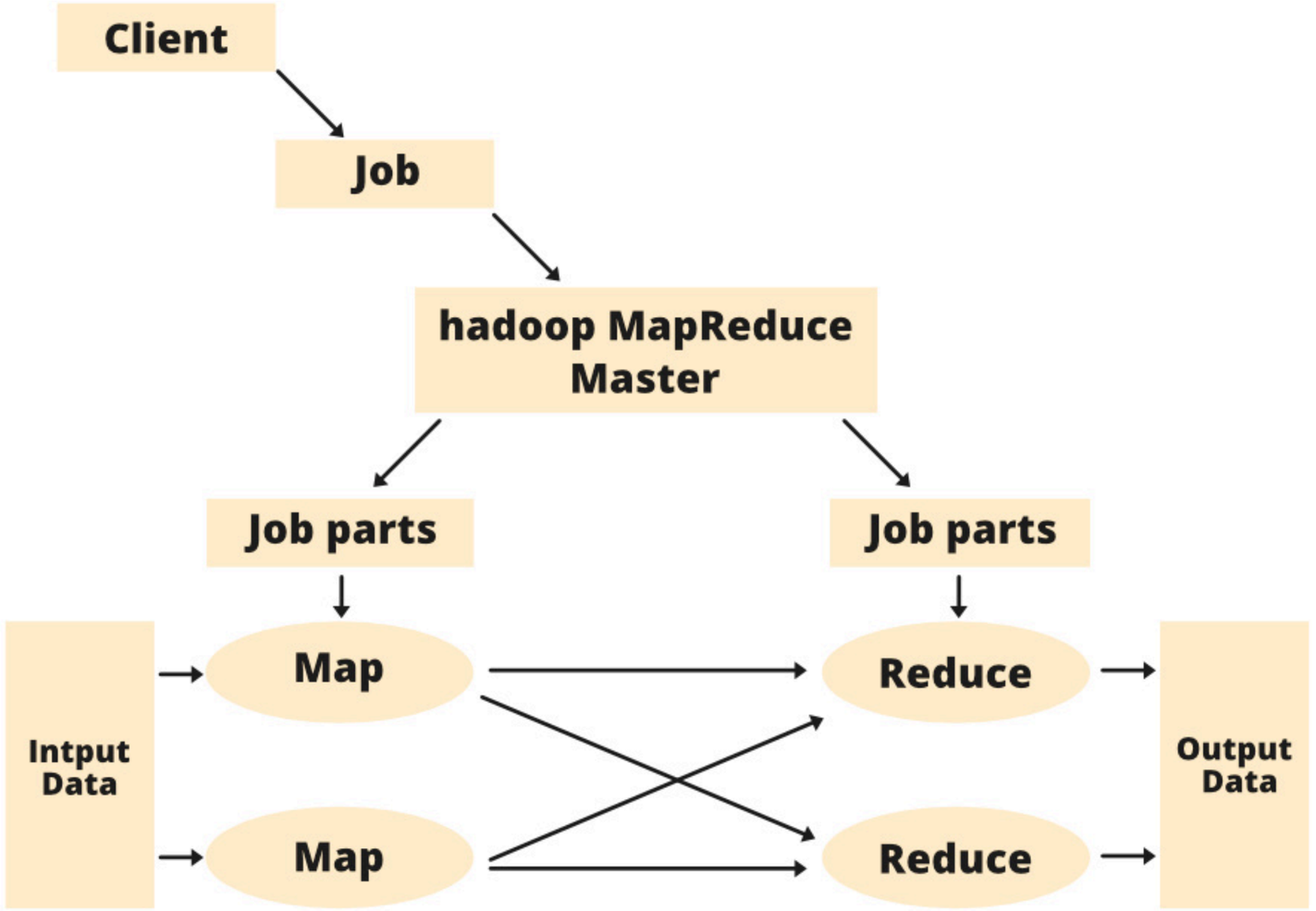
Task Tracker :

- Acts as a slave, executing Map and Reduce tasks based on Job Tracker's instructions.
- Each node in the cluster has a Task Tracker responsible for processing assigned tasks.

Job History Server:

- It is a daemon process that stores historical information about tasks and applications.
- It maintains logs and records details of completed jobs for monitoring and debugging.

Map Reduce Architecture



MAP REDUCE WORKFLOW

1. Job Submission – The client submits a job to the Hadoop MapReduce Master.

2. Job Splitting – The Master node divides the job into smaller tasks (Map tasks).

3. Input Data Splitting – The input data is split into chunks and distributed across nodes.

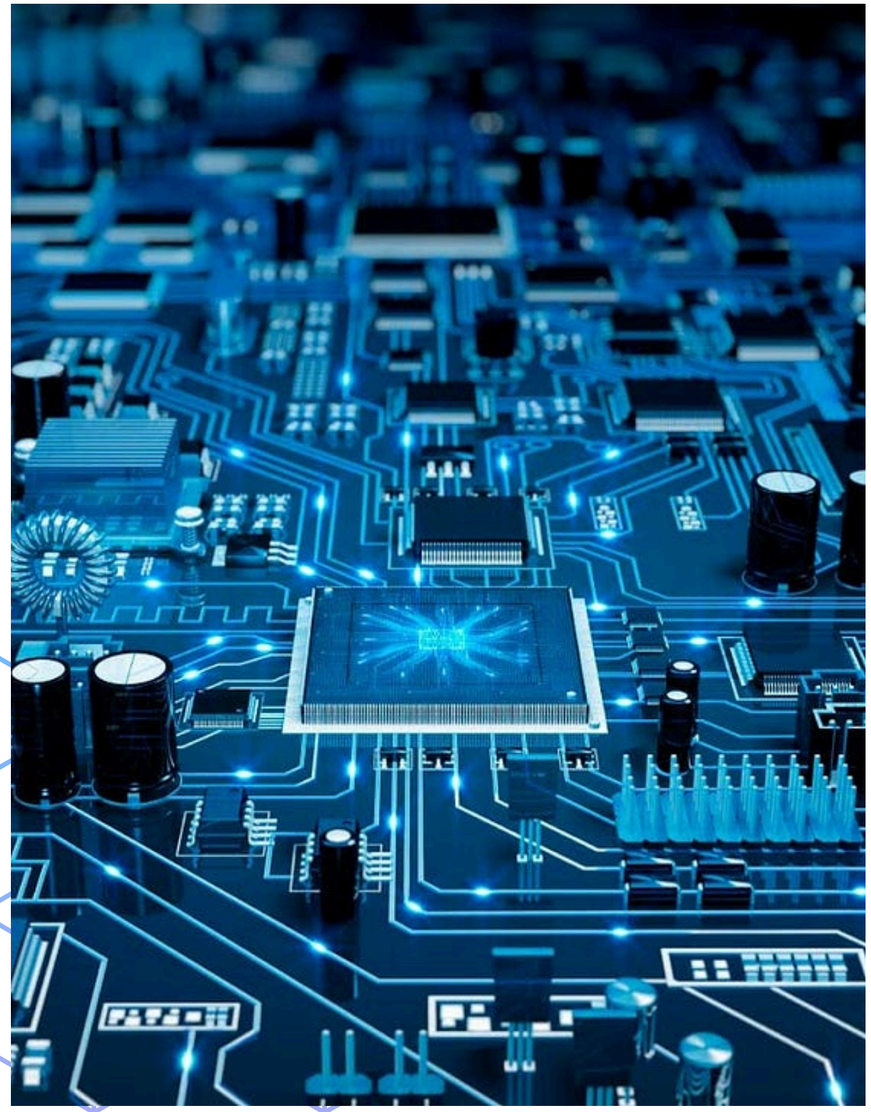
4. Map Phase Execution – Each Map task processes its assigned chunk and generates intermediate key-value pairs.

5. Shuffling & Sorting – The intermediate key-value pairs are grouped and sorted by key.

6. Reduce Phase Execution – The Reduce function processes the sorted data to generate the final output.

7. Output Storage – The final result is stored in HDFS or cloud storage for further use.

8. Job Completion – The client receives a notification once the job is successfully completed



Integration of MapReduce with Cloud Storage

WHY INTEGRATE MAP REDUCE WITH CLOUD STORAGE?

- **Scalability & Elasticity**
- **Cost-Effectiveness**
- **High Availability & Reliability**
- **Faster Processing & Seamless Integrationlity**
- **Security & Compliance**

HOW IT WORKS?

- The MapReduce framework operates exclusively on <key, value> pairs.
- Input to the job as a set of <key, value> pairs and produces a set of <key, value> pairs as the output of the job.
- The key and value classes have to be serializable by the framework and hence need to implement the Writable interface.
- The key classes have to implement the WritableComparable interface to facilitate sorting by the framework.
- Input and Output types of a MapReduce job:
    - (input) <k1, v1> -> map -> <k2, v2> -> combine -> <k2, v2> -> reduce -> <k3, v3> (output)

HOW IT WORKS?

Source Code

```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }
```

```
  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

HOW IT WORKS?

Assuming environment variables are set as follows:

```
export JAVA_HOME=/usr/java/default
export PATH=${JAVA_HOME}/bin:${PATH}
export HADOOP_CLASSPATH=${JAVA_HOME}/lib/tools.jar
```

Compile WordCount.java and create a jar:

```
$ bin/hadoop com.sun.tools.javac.Main WordCount.java
$ jar cf wc.jar WordCount*.class
```

Assuming

- `/user/joe/wordcount/input` - input directory in HDFS
- `/user/joe/wordcount/output` - output directory in HDFS

HOW IT WORKS?

Sample text-files as input:

```
$ bin/hadoop fs -ls /user/joe/wordcount/input/
/user/joe/wordcount/input/file01
/user/joe/wordcount/input/file02

$ bin/hadoop fs -cat /user/joe/wordcount/input/file01
Hello World Bye World

$ bin/hadoop fs -cat /user/joe/wordcount/input/file02
Hello Hadoop Goodbye Hadoop
```

Output:

```
$ bin/hadoop fs -cat /user/joe/wordcount/output/part-r-00000
Bye 1
Goodbye 1
Hadoop 2
Hello 2
World 2
```

HOW IT WORKS?

Output of first map:

< Bye, 1>
< Hello, 1>
< World, 2>

Output of second map:

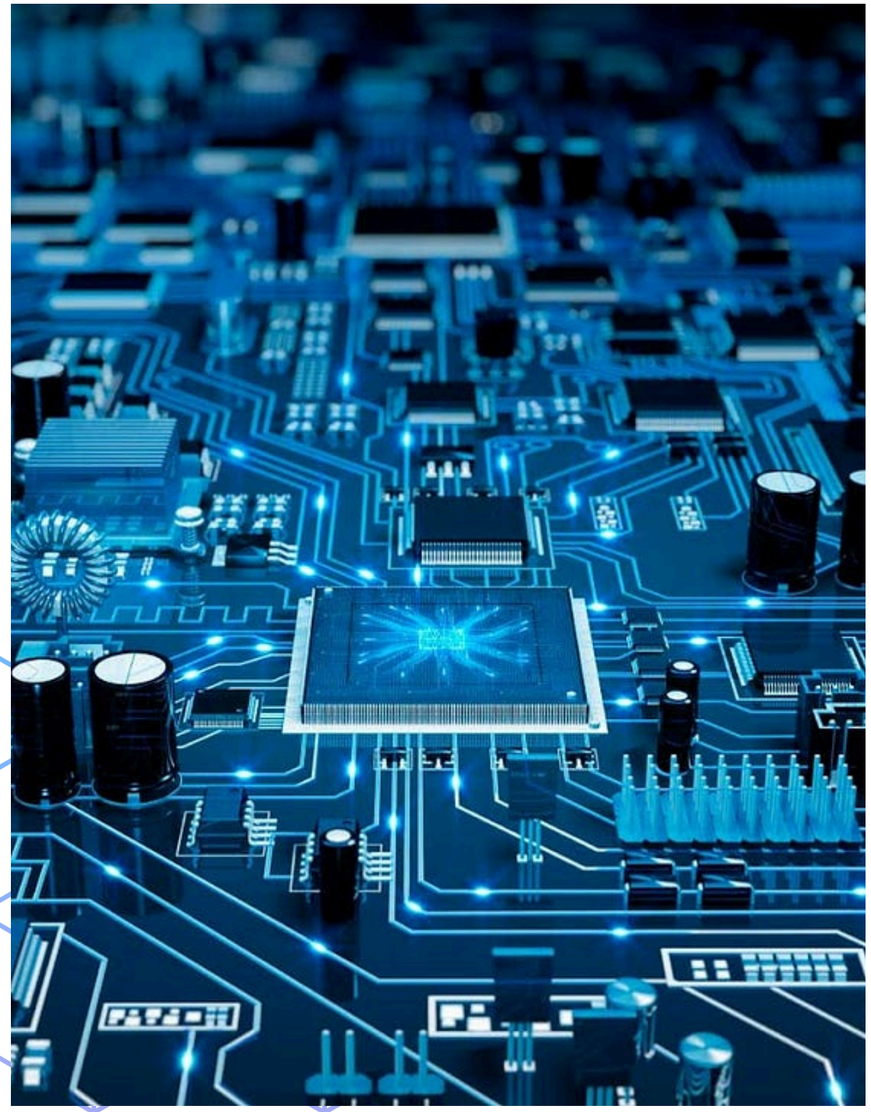
< Goodbye, 1>
< Hadoop, 2>
< Hello, 1>

output of the job is:

< Bye, 1>
< Goodbye, 1>
< Hadoop, 2>
< Hello, 2>
< World, 2>

```
public void map(Object key, Text value, Context context
                ) throws IOException, InterruptedException {
StringTokenizer itr = new StringTokenizer(value.toString());
                while (itr.hasMoreTokens()) {
                   word.set(itr.nextToken());
                   context.write(word, one);
                }
                }
```

```
public void reduce(Text key, Iterable<IntWritable> values,
                                Context context
                ) throws IOException, InterruptedException {
                        int sum = 0;
                for (IntWritable val : values) {
                        sum += val.get();
                }
                result.set(sum);
                context.write(key, result);
                }
```



Optimization Techniques for Efficient MapReduce Processing

WHY IS OPTIMIZATION NECESSARY?

- **MapReduce is powerful but not inherently efficient**
- **Challenges faced in large-scale data processing:**

1. **High execution time**
2. **Uneven workload distribution**
3. **High network I/O due to shuffling**
4. **Resource underutilization**

KEY OPTIMIZATION STRATEGIES

- **Combiners – Reduce intermediate data**
- **Partitioning & Load Balancing – Distribute work efficiently**
- **Data Locality Optimization – Minimize network overhead**
- **Speculative Execution – Handle slow tasks**
- **Job Scheduling & Resource Allocation – Optimize job execution**
- **Reducing Shuffle & Sort Overhead – Compress and aggregate data**

COMBINERS AND PARTITIONING

- A Combiner is a mini-reducer that runs on mapper output before shuffling. It reduces intermediate data size (between mapper and reducer) and improves efficiency.
- Word count: Instead of sending ("word", 1) multiple times, the combiner pre-aggregates counts per node
- Due to uneven distribution of keys, some reducers are overloaded. The solution is to use custom partitioners to evenly distribute keys.
- Example: In a sales dataset, partition by region instead of default hash partitioning

DATA LOCALITY OPTIMIZATION

- **Move computation closer to where data is stored because fetching data from remote nodes increases network latency**
- **Increase replication factor for better scheduling**
- **Faster job execution with lower network congestion**

SPECULATIVE EXECUTION FOR STRAGGLER TASKS

- Slow-running tasks (stragglers) delay job completion.
- The solution is to enable speculative execution to launch duplicate tasks on another node.
- Hadoop picks the faster one and kills the slower one.
- Use if heterogeneous hardware causes some nodes to be slower or if task failures are common due to network issues.

OPTIMIZED JOB SCHEDULING & RESOURCE ALLOCATION

- **Different Hadoop Schedulers:**

1. **FIFO Scheduler – Simple, but inefficient for multi-user clusters**
2. **Fair Scheduler – Balances resource allocation**
3. **Capacity Scheduler – Allows prioritization of critical jobs**

REDUCING SHUFFLE & SORT OVERHEAD

- **High network traffic due to large intermediate data**
- **Solutions:**

1. **Compression: Reduce shuffle size (mapreduce.map.output.compress=true)**
2. **Pre-sorting keys: Optimize sorting before reducers**
3. **Using In-Memory Data Structures: Cache frequently used data instead of repeatedly reading from disk**

THANK YOU