BIG DATA TECHNOLOGIES

INTRODUCTION TO BIG DATA AND HADOOP

INTRODUCTION TO BIG DATA

Big Data Definition: Big Data refers to datasets that are too large or complex for traditional data-processing software to handle. This encompasses data volume, variety, velocity, and veracity.
In an era where data is generated at an unprecedented scale from social media interactions, IoT devices, and e-commerce transactions, traditional data processing methods fall short. Big Data technologies enable us to harness these vast datasets, uncovering insights that drive innovation and strategic decision-making. The early 2000s marked the beginning of this revolution, with significant technological advancements paving the way for what we now call Big Data.

Characteristics of Big Data

• Volume: The sheer amount of data generated every second. For example, social media platforms generate terabytes of data daily.
• Variety: Data comes in various formats, including structured (databases), semi-structured (XML, JSON), and unstructured (text, images, videos).
• Velocity: The speed at which data is generated and processed. Real-time data processing is crucial for applications like fraud detection.
• Veracity: The quality and trustworthiness of data. Ensuring data accuracy and reliability is a significant challenge.

WHY BIG DATA IS IMPORTANT

Enhanced Decision Making: Organizations can make more informed decisions by analyzing large datasets.
Predictive Analytics: Big Data enables predictive analytics, helping businesses forecast trends and behaviors.
Operational Efficiency: Streamlines operations by identifying inefficiencies and optimizing processes.

BIG DATA SKILLS AND SOURCES OF BIG DATA

Key skills include data analysis, programming (Python, R), and proficiency with Big Data tools (Hadoop, Spark).
Major Data Sources:

- Social Media: Generates vast amounts of user-generated content daily.
- IoT Devices: Collect data from sensors and smart devices.
- E-Commerce Transactions: Record customer behaviors and preferences.
- Scientific Research: Produces extensive data from experiments and simulations.

BIG DATA ADOPTION

Industry Usage: Finance, healthcare, retail, telecommunications, and manufacturing are leading adopters.

Challenges: Data privacy concerns, high initial setup costs, and the need for skilled professionals.
Timeline: Significant adoption began in the mid-2000s, with increasing acceleration in the 2010s.

RESEARCH AND CHANGING NATURE OF DATA REPOSITORIES

Traditional Repositories: Relational databases and data warehouses designed for structured data.

Modern Repositories: Data lakes that store raw data in its native format, NoSQL databases for unstructured data, and cloud storage solutions for scalability.

With the growth of data volume and variety, new storage solutions emerged. Traditional relational databases, while effective for structured data, are less suitable for the flexibility required by Big Data. NoSQL databases and distributed file systems like HDFS gained prominence in the late 2000s.

DATA SHARING AND REUSE PRACTICES AND THEIR IMPLICATIONS FOR REPOSITORY DATA CURATION

Effective data sharing and reuse are vital for collaboration and innovation. Ensuring data quality, creating comprehensive metadata, and maintaining long-term accessibility require robust data curation strategies, which gained focus in the early 2010s.

Collaborative Platforms: Tools and platforms that facilitate data sharing among researchers, such as GitHub, Zenodo, and Figshare.

Licensing and Compliance: Ensuring data is shared in compliance with legal and ethical standards, using licenses like Creative Commons.

DATA CURATION SERVICES IN ACTION

Data curation involves cleaning data, creating metadata, and establishing standards to ensure usability. Organizations such as data archives and libraries play crucial roles in providing these services, ensuring long-term data accessibility.

THE CURRENT STATE OF META-REPOSITORIES FOR DATA

Meta-repositories aggregate metadata from various data repositories, making it easier to discover and access datasets. These centralized access points provide significant value by simplifying the search for relevant data.

Examples: DataCite, OpenAIRE, and re3data.

Benefits: Provide a centralized access point for diverse datasets.

INTRODUCTION TO HADOOP

A BRIEF HISTORY OF HADOOP

Hadoop's development was inspired by Google's research on the Google File System (2003) and MapReduce (2004). Doug Cutting and Mike Cafarella created Hadoop in 2005, initially as part of the Apache Nutch project. By 2006, it became a subproject of Lucene, and in 2008, Yahoo! adopted Hadoop for its web search operations.

The history of Hadoop dates back to the early 2000s:

• 2003: Google published the Google File System (GFS) paper, detailing a scalable

distributed file system.
• 2004: Google published the MapReduce paper, which described a programming model for processing large data sets.
• 2005: Doug Cutting and Mike Cafarella created the open-source Hadoop project. Initially, it was a part of the Nutch project, an open-source web search engine.
• 2006: Yahoo! adopted Hadoop and created a dedicated team to further its development.
• 2008: Hadoop became a top-level project at the Apache Software Foundation

EVOLUTION OF HADOOP

Hadoop evolved from a single-node framework into a robust, distributed computing platform. Significant milestones include the introduction of YARN in Hadoop 2.x (2012), which improved resource management and job scheduling, enabling Hadoop to handle more complex and large-scale data processing tasks.

Hadoop 1.0: This version included the Hadoop Distributed File System (HDFS) and MapReduce for data processing.

Hadoop 2.0: Introduced YARN (Yet Another Resource Negotiator), which decoupled resource management from the data processing framework, allowing for more flexible and efficient use of resources.

Hadoop 3.0: Brought significant improvements, including support for erasure coding for fault tolerance, multiple NameNodes, and containerized applications.

INTRODUCTION TO HADOOP AND ITS COMPONENTS

Hadoop is an open-source framework designed for efficient storage and processing of large datasets. Its core components are:

HDFS (Hadoop Distributed File System): Provides scalable and fault-tolerant storage by distributing data across multiple nodes.

MapReduce: A programming model for parallel processing of large datasets.

YARN (Yet Another Resource Negotiator): Manages cluster resources and job scheduling, enhancing Hadoop's scalability and efficiency.

COMPARISON WITH OTHER SYSTEMS

Hadoop is often compared to traditional relational databases and other Big Data frameworks. Relational databases excel with structured data and vertical scalability, while Hadoop handles unstructured data and offers horizontal scalability. Apache Spark, another Big Data framework, provides in-memory processing, which can be faster than Hadoop's disk-based processing for certain tasks.

Apache Spark: Spark provides in-memory processing, which can be faster than Hadoop's disk-based processing. It supports batch and real-time processing, while Hadoop is traditionally batch-oriented.

Apache Flink: Similar to Spark, Flink is designed for stream processing but also supports batch processing.

Apache Storm: Focuses on real-time processing, offering low-latency and distributed

computing.
Google BigQuery: A fully-managed data warehouse with SQL capabilities, offering fast querying on large datasets.
Amazon Redshift: A fully-managed data warehouse service in the cloud, designed for large-scale data storage and analysis.

HADOOP RELEASES
Major Versions:
Hadoop 1.x: Initial version with basic HDFS and MapReduce functionality.
Hadoop 2.x: Introduced YARN, improving resource management and scalability.
Hadoop 3.x: Enhanced storage efficiency, support for multiple standby NameNodes, and other performance improvements.

HADOOP DISTRIBUTIONS AND VENDORS
Commercial Distributions: Companies offer enhanced Hadoop distributions with additional tools and support.
Popular Distributions:
Cloudera CDH: Comprehensive Hadoop distribution with enterprise support.
Hortonworks Data Platform (HDP): Open-source distribution with a focus on community-driven innovation.
MapR: Known for its unique file system and enterprise-grade features.

HADOOP DISTRIBUTED FILE SYSTEM (HDFS)
DISTRIBUTED FILE SYSTEM
A Distributed File System (DFS) allows data to be stored across multiple nodes in a cluster, enabling high availability, fault tolerance, and scalability. HDFS is a DFS designed specifically for handling large datasets with high throughput access patterns.

WHAT IS HDFS
HDFS (Hadoop Distributed File System) is an open-source framework designed to store and manage large volumes of data across multiple machines. It provides fault tolerance and high availability by distributing data redundantly.

WHERE DOES HDFS FIT IN
HDFS is a core component of the Hadoop ecosystem, providing the foundational storage layer upon which data processing frameworks like MapReduce, Hive, and Spark operate. It is designed to handle large-scale data storage and processing tasks efficiently.

Core Components of HDFS

- NameNode: Manages the file system metadata and namespace.
- DataNode: Stores the actual data blocks.

- Secondary NameNode: Periodically merges the NameNode's namespace with the edit logs to prevent log file size from becoming too large.

HDFS DAEMONS

NameNode Daemon: Manages the metadata and directory structure.
DataNode Daemon: Handles read/write requests from clients and manages the storage of data blocks.
Secondary NameNode Daemon: Assists the NameNode by taking periodic snapshots of its metadata.
HADOOP SERVER ROLES: NAME NODE, SECONDARY NAME NODE, AND DATA NODE
NameNode: Centralized master node managing file system metadata and operations.
Secondary NameNode: Works with the NameNode to perform housekeeping tasks, not a true standby node.
DataNode: Worker nodes storing data blocks and handling client read/write requests.

HDFS ARCHITECTURE

HDFS architecture consists of a single NameNode and multiple DataNodes. The NameNode manages metadata and the DataNodes store actual data. This master-slave architecture enables efficient, scalable data storage and retrieval.
Scaling and Rebalancing:HDFS scales by adding more DataNodes to the cluster. Rebalancing is performed to ensure data blocks are evenly distributed across DataNodes, maintaining performance and storage efficiency.
Replication:HDFS ensures fault tolerance by replicating data blocks across multiple DataNodes. By default, each block is replicated three times, ensuring data availability even if some nodes fail.
Rack Awareness:HDFS employs a rack awareness algorithm to improve data reliability and network bandwidth utilization. It places replicas on different racks to prevent data loss due to rack failures and to reduce network traffic during read/write operations.
Data Pipelining: Data pipelining in HDFS involves writing data to a DataNode, which then forwards the data to the next DataNode in the pipeline, continuing until all replicas are written. This process ensures efficient data replication and distribution.
Node Failure Management:HDFS is designed to handle node failures gracefully. When a DataNode fails, the NameNode re-replicates the data blocks stored on the failed node to other healthy nodes, ensuring data availability.

HDFS HIGH AVAILABILITY NAMENODE

HDFS High Availability (HA) feature eliminates the NameNode as a single point of failure by providing a standby NameNode. In the event of a failure, the standby NameNode takes over, ensuring continuous availability of the HDFS services.

HADOOP INSTALLATION AND CLUSTER CONFIGURATION

HADOOP OPERATION MODES

Local (Standalone) Mode: Runs on a single node using the local file system. Ideal for debugging and development. No HDFS is involved.
Pseudo-Distributed Mode: Each Hadoop daemon runs as a separate Java process on a single node,

simulating a distributed environment. Useful for testing and development.
Fully Distributed Mode: Hadoop runs on a cluster of nodes with separate nodes for NameNode, Secondary NameNode, and DataNodes. Used for production environments.

SETTING UP A HADOOP CLUSTER
Cluster Planning: Define the number of nodes, hardware requirements (CPU, memory, storage), and network configurations.
Install Java: Ensure Java is installed on all nodes as Hadoop relies on Java.
Download Hadoop: Obtain the Hadoop distribution from the Apache Hadoop website.
Configure Hadoop:
core-site.xml: Configure general Hadoop settings, such as the default filesystem name.
hdfs-site.xml: Configure HDFS settings, including replication factor and data directories.
mapred-site.xml: Configure MapReduce settings.
yarn-site.xml: Configure YARN settings.

CLUSTER SPECIFICATION
Master Nodes: Includes NameNode (manages file system metadata) and ResourceManager (manages job scheduling and resource allocation).
Slave Nodes: Includes DataNodes (store data blocks) and NodeManagers (manage application containers and resource usage).
Hardware Requirements: Plan for sufficient CPU, memory, and storage based on the expected workload.
Network Configuration: Ensure high-speed network connections between nodes to facilitate efficient data transfer and communication.

SINGLE AND MULTI-NODE CLUSTER SETUP ON VIRTUAL & PHYSICAL MACHINES
Single-Node Setup:
1. Install Java: Ensure JDK is installed.
2. Download Hadoop: Get Hadoop binaries from the official Apache Hadoop website.
3. Configure Hadoop: Edit configuration files (core-site.xml, hdfs-site.xml, mapred-site.xml, and yarn-site.xml).
4. Format the Namenode: Run hdfs namenode -format.
5. Start Hadoop Services: Use start-dfs.sh and start-yarn.sh scripts.
6. Verify Installation: Access Hadoop web interfaces (HDFS: http://localhost:9870, YARN: http://localhost:8088).
Multi-Node Setup:
1. Prepare Machines: Ensure all nodes have Java installed and SSH enabled.
2. Configure Networking: Set up hostname resolution for all nodes.
3. Download Hadoop: Install Hadoop on all nodes.
4. Configure Master and Slave Nodes:
o On the master, configure core-site.xml, hdfs-site.xml, mapred-site.xml, and yarn-site.xml.

o On slaves, update hdfs-site.xml and yarn-site.xml with the master node's details.
5. Distribute Configuration: Copy configuration files from the master to all slave nodes.
6. Start Hadoop Services: Start HDFS and YARN services on the master, followed by slaves.

REMOTE LOGIN USING PUTTY/MAC TERMINAL/UBUNTU TERMINAL

Putty (Windows): Use Putty to SSH into Hadoop nodes from a Windows machine.
Mac Terminal/Ubuntu Terminal: Use built-in SSH clients for remote login from macOS or Linux.
SSH Key Setup: Configure SSH key-based authentication for password-less login between nodes.

HADOOP CONFIGURATION, SECURITY IN HADOOP, ADMINISTERING HADOOP

Hadoop Configuration: Modify configuration files to set up paths, replication factors, and other cluster parameters.
Security in Hadoop: Implement Kerberos authentication, enable HDFS encryption, and set up access control lists (ACLs) for secure data management.
Administering Hadoop: Use tools like Apache Ambari, Cloudera Manager, or custom scripts to manage and monitor the Hadoop cluster.

HDFS – MONITORING & MAINTENANCE

Monitoring Tools: Use monitoring tools like Ambari, Ganglia, or Nagios to track HDFS performance and health.
Maintenance Tasks:
Data Balancing: Ensure data is evenly distributed.
Data Integrity: Run HDFS fsck to check for file system inconsistencies.
Upgrade and Patching: Regularly update Hadoop to the latest stable version..

HADOOP BENCHMARKS

TestDFSIO: Measures the I/O performance of HDFS.
TeraSort: Measures the time taken to sort a large dataset (e.g., 1TB of data).
MRBench: Evaluates the performance of MapReduce jobs.

HADOOP IN THE CLOUD

Cloud Providers: AWS (EMR), Microsoft Azure (HDInsight), and Google Cloud (Dataproc) offer managed Hadoop services.
Benefits: Quick provisioning, cost-effective scalability, and simplified management.
Considerations: Data security, network latency, and cost management.

HADOOP ARCHITECTURE

Hadoop is designed as a distributed storage and processing framework, capable of handling vast

amounts of data across clusters of computers. Its architecture consists of several key components that work together to provide reliable, scalable, and efficient data processing.

1. **Hadoop Distributed File System** (HDFS): A scalable and fault-tolerant file system designed to run on commodity hardware.
2. **MapReduce**: A programming model for processing large datasets in parallel across a Hadoop cluster.

CORE COMPONENTS OF HADOOP

HDFS (Hadoop Distributed File System): A distributed file system that provides high-throughput access to data. It stores large files across multiple machines, ensuring fault tolerance by replicating data blocks.
YARN (Yet Another Resource Negotiator): The resource management layer of Hadoop. It manages and schedules computing resources across the cluster.
MapReduce: A programming model used for processing large datasets. It divides tasks into smaller sub-tasks (Map) and aggregates the results (Reduce).
Hadoop Common: A collection of utilities and libraries that support the other Hadoop modules. These include scripts, configuration files, and error-handling utilities.

COMMON HADOOP SHELL COMMANDS

HDFS Commands

1. **Basic File Operations**:
o hdfs dfs -ls /path: List files in the specified directory.
o hdfs dfs -mkdir /path: Create a directory in HDFS.
o hdfs dfs -rm /path: Delete a file from HDFS.
o hdfs dfs -rmdir /path: Remove a directory from HDFS.
o hdfs dfs -copyFromLocal local_path hdfs_path: Copy a file from the local file system to HDFS.
o hdfs dfs -copyToLocal hdfs_path local_path: Copy a file from HDFS to the local file system.
2**. File Management**:
o hdfs dfs -put local_path hdfs_path: Upload files from the local file system to HDFS.
o hdfs dfs -get hdfs_path local_path: Download files from HDFS to the local file system.
o hdfs dfs -moveFromLocal local_path hdfs_path: Move files from the local file system to HDFS.
o hdfs dfs -moveToLocal hdfs_path local_path: Move files from HDFS to the local file system.
3. **File System Checks:**
o hdfs fsck /path: Check the health of the file system.
o hdfs dfsadmin -report: Get a detailed report on the HDFS cluster.

YARN Commands

1. **ResourceManager Commands:**
o yarn rmadmin -refreshQueues: Refresh the list of queues from the resource manager configuration.
o yarn rmadmin -refreshNodes: Refresh the list of nodes in the cluster.
2. **NodeManager Commands:**
o yarn nodemanager: Start the NodeManager daemon.
o yarn node -list: List all nodes in the cluster.
3. **Application Management:**
o yarn application -list: List all applications in the cluster.
o yarn application -status application_id: Get the status of a specific application.
o yarn application -kill application_id: Kill a specific application.
4. **Cluster Management:**
o yarn cluster -metrics: Get cluster-wide metrics.
o yarn cluster -report: Get a detailed report on the cluster's status.

HDFS DATA STORAGE PROCESS

HDFS stores data across multiple nodes in a distributed manner. Files are split into large blocks (default 128 MB) and distributed across DataNodes. Each block is replicated on multiple nodes to ensure reliability and fault tolerance.

ANATOMY OF WRITING AND READING FILE IN HDFS

Writing to HDFS:
A client divides a file into blocks.
Blocks are sent to DataNodes, where they are stored.
The NameNode manages the metadata, keeping track of block locations.
Reading from HDFS:
The client requests the file from the NameNode.
The NameNode provides the locations of the blocks.
The client retrieves the blocks from the corresponding DataNodes and reassembles the file.

Handling Read/Write Failures

HDFS is designed to handle failures gracefully:
• **Write Failures:**
o DataNode Failure: If a DataNode fails while writing a block, the client writes the block to another DataNode.
o Pipeline Failure: If a DataNode in the replication pipeline fails, the client reconstructs the pipeline and continues writing.
• **Read Failures:**
o DataNode Unavailability: If a DataNode holding a block becomes unavailable, the client retrieves the block from another DataNode holding a replica.

o Block Corruption: HDFS periodically checks the integrity of data
blocks using checksums. If corruption is detected, the block is re replicated from another replica.

HDFS USER AND ADMIN COMMANDS

User Commands:

hdfs dfs -ls: Lists files in HDFS.
hdfs dfs -put: Uploads files to HDFS.
Admin Commands:
hdfs dfsadmin -report: Displays a summary of HDFS cluster health.
hdfs dfsadmin -safemode: Puts the cluster into safe mode, where no changes can be made.
HDFS WEB INTERFACE
The HDFS web interface provides a graphical view of the cluster, including file system status, DataNode health, and block distribution. It can be accessed via a web browser using the NameNode's IP address and port.

MAPREDUCE

HADOOP MAPREDUCE PARADIGM

MapReduce is a programming model used for processing large datasets in parallel across a Hadoop cluster. The process involves two main steps:
Map: Processes input data and converts it into a set of key-value pairs.
Reduce: Aggregates and processes the key-value pairs to produce the final output.
MAP AND REDUCE TASKS
Map Tasks: Each Map task processes a split of the input data and generates intermediate key-value pairs.
Reduce Tasks: Reduce tasks aggregate the intermediate key-value pairs generated by the Map tasks to produce the final output.

MAPREDUCE EXECUTION FRAMEWORK

The execution framework manages the distribution of Map and Reduce tasks across the cluster. YARN handles resource allocation, job scheduling, and monitoring.

MAPREDUCE DAEMONS

JobTracker: (Legacy) Used in Hadoop 1.x for tracking the progress of MapReduce jobs and coordinating task execution. Replaced by ResourceManager in Hadoop 2.x.
TaskTracker: (Legacy) Used in Hadoop 1.x for executing MapReduce tasks. Replaced by NodeManager in Hadoop 2.x.
ResourceManager: Manages resource allocation across the cluster.
NodeManager: Manages execution of tasks on each node.

ANATOMY OF A MAPREDUCE JOB RUN

A MapReduce job is submitted to the ResourceManager, which assigns tasks to NodeManagers. The job progresses through:

Input Splitting: Data is split into chunks for processing.
Mapping: Map tasks process the input splits.
Shuffling: Intermediate key-value pairs are sorted and grouped.
Reducing: Reduce tasks aggregate the sorted key-value pairs.
Output: The final output is written to HDFS.

Example of a MapReduce Job
Word Count Example:
1. **Mapper Class:**

```
public 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);
  }
  }
}
```

2. **Reducer Class:**

```
public 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);
  }
}
```

3. **Driver Class:**

```
public class WordCount {
  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);
  }
}
```

PARTITIONERS AND COMBINERS

Partitioners: Control how the output of the map tasks is distributed among the reduce tasks. They determine which key-value pairs are sent to which reducer.

Combiners: Local aggregators that reduce the volume of data sent from the Map tasks to the Reduce tasks, improving performance.

Input Formats:

Input Splits and Records: Determine how input files are divided into splits for processing.

Text Input: Default format where each line of input is treated as a record.

Binary Input: Handles binary data.

Multiple Inputs: Allows combining different types of input data formats.

Output Formats:

Text Output: Default format for writing output as text.

Binary Output: Used for writing binary data.

Multiple Output: Allows writing output to multiple destinations.

DISTRIBUTED CACHE

A mechanism for distributing files needed by MapReduce tasks across the cluster. These files can be accessed locally by each task, improving efficiency.

BASICS OF MAPREDUCE PROGRAMMING

HADOOP DATA TYPES

Hadoop uses specific data types (e.g., IntWritable, Text, LongWritable) that are optimized for network transmission and serialization.

Java and MapReduce:MapReduce programs are typically written in Java. Developers define Mapper and Reducer classes, specifying how input data is processed.

MAPREDUCE PROGRAM STRUCTURE

A typical MapReduce program includes:

1. Mapper Class:

o Processes input data and emits key-value pairs.

```
public class MyMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();
  public void map(LongWritable 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);
  }
  }
}
```

2. Reducer Class:

o Processes intermediate key-value pairs and emits final output.

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

3. Driver Class:

o Configures and runs the MapReduce job.

```
public class MyDriver {
  public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();
  Job job = Job.getInstance(conf, "my job");
  job.setJarByClass(MyDriver.class);
  job.setMapperClass(MyMapper.class);
  job.setCombinerClass(MyReducer.class);
  job.setReducerClass(MyReducer.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);
  }
}
```

Map-only Program, Reduce-only Program

1. Map-only Program:

o Only uses the Mapper class. No Reducer is specified, which means the output of the Mapper is the final output.

```
public class MyMapOnlyDriver {
  public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();
  Job job = Job.getInstance(conf, "map only job");
  job.setJarByClass(MyMapOnlyDriver.class);
  job.setMapperClass(MyMapper.class);
  job.setNumReduceTasks(0); // No reducers
  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);
  }
}
```

2**. Reduce-only Program:**

o In some rare cases, you might need to run a Reduce job without a preceding Map job. This can be done by generating the initial key-value pairs outside of Hadoop or using an existing data set.

```
public class MyReduceOnlyDriver {
  public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();
  Job job = Job.getInstance(conf, "reduce only job");
  job.setJarByClass(MyReduceOnlyDriver.class);
  job.setMapperClass(Mapper.class); // Identity Mapper
  job.setReducerClass(MyReducer.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);
  }
}
```

Use of Combiner and Partitioner

1. Combiner:

o Acts as a mini-reducer to perform local aggregation on the output of the Mapper. This reduces the amount of data transferred to the Reducer.

```
public class MyCombiner extends Reducer<Text, IntWritable, Text, IntWritable> {
  public void reduce(Text key, Iterable<IntWritable> values, Context context) throws
IOException, InterruptedException {
  int sum = 0;
  for (IntWritable val : values) {
```

```
  sum += val.get();
  }
  context.write(key, new IntWritable(sum));
  }
}
```

2. Partitioner:

o Determines the reducer that each key-value pair will go to. By default, Hadoop uses HashPartitioner.

```
public class MyPartitioner extends Partitioner<Text, IntWritable> {
  @Override
  public int getPartition(Text key, IntWritable value, int numReduceTasks) {
  // Custom partitioning logic
  return (key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;
  }
}
```

Counters

• Counters are used for gathering statistics about the job, such as the number of processed records or error occurrences.

```
public class MyMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
  enum Counters { NUM_RECORDS }
  public void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {
  context.getCounter(Counters.NUM_RECORDS).increment(1);
  // Map logic
  }
}
```

Schedulers (Job Scheduling)

• FIFO Scheduler: The default scheduler that schedules jobs in the order they arrive.

• Capacity Scheduler: Allows setting resource capacity for different queues, ensuring that multiple jobs can run concurrently.

• Fair Scheduler: Distributes resources fairly among all jobs, ensuring that all jobs get an equal share of resources over time.

Custom Writables

• Custom Writable classes can be created to handle complex data types.

```
public class CustomWritable implements Writable {
  private IntWritable intField;
  private Text textField;
  public CustomWritable() {
```

```
  this.intField = new IntWritable();
  this.textField = new Text();
  }
  @Override
  public void write(DataOutput out) throws IOException {
  intField.write(out);
  textField.write(out);
  }
  @Override
  public void readFields(DataInput in) throws IOException {
  intField.readFields(in);
  textField.readFields(in);
  }
}
```

Compression

• Compression reduces the size of intermediate data and final output, saving storage space and speeding up data transfer.

1. Input Compression:

```
conf.set("mapreduce.input.fileinputformat.split.minsize", "128000000");
```

2. Output Compression:

```
conf.set("mapreduce.output.fileoutputformat.compress", "true");
conf.set("mapreduce.output.fileoutputformat.compress.codec",
"org.apache.hadoop.io.compress.GzipCodec");
```

3. Intermediate Compression:

```
conf.set("mapreduce.map.output.compress", "true");
conf.set("mapreduce.map.output.compress.codec",
"org.apache.hadoop.io.compress.SnappyCodec");
public class CustomPartitioner extends Partitioner<Text, IntWritable> {
  @Override
  public int getPartition(Text key, IntWritable value, int numReduceTasks) {
  // Custom logic to determine partition number
  return (key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;
  }
}
```

Combiners:

• Combiners are optional components that perform local aggregation of intermediate outputs to reduce the amount of data transferred to the reducers.

• They run after the map phase and before the shuffle and sort phase. The combiner's logic is similar to that of a reducer.

```
public class MyCombiner extends Reducer<Text, IntWritable, Text, IntWritable> {
```

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

MAPREDUCE STREAMING

MapReduce Streaming is a versatile utility in the Hadoop ecosystem that allows developers to write MapReduce jobs using any executable or script, rather than being limited to Java. This flexibility enables the use of languages like Python, Perl, or Bash for creating Map and Reduce tasks, making it accessible to a broader range of developers. Through Streaming, users can process text-based data efficiently and integrate custom data processing scripts into Hadoop's distributed computing framework. This capability is especially useful for handling non-Java environments, processing complex data formats, and leveraging existing scripts for data transformation and analysis.

HADOOP ETL DEVELOPMENT

ETL stands for Extract, Transform, and Load. It's a process used in data warehousing to extract data from various sources, transform it into a suitable format, and load it into a data warehouse or other storage systems. Hadoop is particularly suited for ETL processes due to its ability to handle large volumes of data across distributed systems.

1. **ETL Development in Hadoop:**

o ETL development in Hadoop involves using tools like Apache Hive, Apache Pig, Apache Spark, and custom MapReduce jobs to perform data extraction, transformation, and loading.

o It leverages Hadoop's distributed storage (HDFS) and processing capabilities to handle big data efficiently.

2. Typical Workflow:

o Extract: Retrieve data from various sources like databases, flat files, APIs, and log files.

o Transform: Clean, filter, aggregate, and manipulate the data to fit the desired format or schema.

o Load: Save the transformed data into a Hadoop-based data warehouse (e.g., Hive), HBase, or other storage systems.

ETL Process in Hadoop

1. Data Extraction:

o Tools: Apache Sqoop for extracting data from relational databases, Flume for collecting log data, and custom scripts or applications for extracting data from other sources.
o Techniques: Incremental extraction for new or updated data, full extraction for complete datasets.
2. **Data Transformation:**
o Tools: Apache Pig (scripting platform for data transformation), Apache Hive (SQL-like queries for transformation), Apache Spark (fast and general-purpose cluster-computing system), and custom MapReduce jobs.
o Processes: Data cleaning, filtering, joining, sorting, aggregating, and applying business rules.
3. **Data Loading:**
o Tools: Hive for loading data into tables, HBase for loading into NoSQL databases, HDFS commands for direct file operations.
o Approaches: Batch loading for large datasets, real-time loading for streaming data.

Discussion of ETL Functions

1. Extract:
o Purpose: To collect raw data from multiple sources, ensuring that it is accurately and efficiently retrieved.
o Challenges: Handling different data formats, ensuring data consistency and completeness, managing large data volumes.
2. Transform:
o Purpose: To convert raw data into a meaningful format, applying transformations such as filtering, aggregating, and enriching data.
o Challenges: Ensuring data quality, handling complex transformation logic, maintaining performance.
3. Load:
o Purpose: To store the transformed data into a target system where it can be used for analysis and reporting.
o Challenges: Ensuring data integrity, optimizing loading performance, managing storage efficiently.

Data Extractions

Data extraction involves retrieving data from various sources. In Hadoop ETL, this typically involves:

1. Structured Data:
o Using Sqoop to import data from relational databases (e.g., MySQL, Oracle).
o Example:

sqoop import --connect jdbc:mysql://hostname/dbname --username user --password pass --table tablename --target-dir /path/to/hdfs

2. Unstructured Data:

o Using Flume to collect and move log data to HDFS.

o Example:

flume-ng agent --conf ./conf --name a1 --conf-file example.conf

3. **Semi-Structured Data:**

o Using custom scripts or tools to extract data from APIs, XML, JSON files.

Need of ETL Tools

1. Scalability: ETL tools are designed to handle large volumes of data across distributed systems, making them ideal for big data environments like Hadoop.
2. Efficiency: They provide optimized methods for extracting, transforming, and loading data, reducing the time and resources required.
3. Complex Transformations: ETL tools offer a variety of built-in functions for data transformation, making it easier to implement complex business rules.
4. Integration: They can connect to various data sources and target systems, facilitating seamless data movement.
5. Automation: ETL tools support scheduling and automation, enabling regular and unattended data processing.

Advantages of ETL Tools

1. User-Friendly: Many ETL tools come with graphical interfaces that simplify the process of designing and managing ETL workflows.
2. Error Handling: Built-in error handling and logging capabilities help identify and resolve issues quickly.
3. Reusability: ETL workflows and components can be reused across different projects, saving development time.
4. Performance Optimization: ETL tools often include performance optimization features, such as parallel processing and in-memory computing.
5. Security: They provide security features to ensure data privacy and integrity during the ETL process.