Big Data and the Hadoop Ecosystem: The Complete Guide

Table of Contents

Introduction to Big Data

Big Data refers to datasets that are too large or complex to be dealt with by traditional data processing applications. These massive volumes of data can be used to address business problems that were previously difficult to tackle.

The 5 V's of Big Data

1. **Volume**: Refers to the vast amounts of data generated every second. Organizations collect data from various sources, including business transactions, social media, and information from sensor or machine-to-machine data. The scale of data has grown from gigabytes to petabytes and even zettabytes.
2. **Velocity**: Refers to the speed at which data is generated and processed. With the advent of IoT devices and real-time applications, data streams in at unprecedented speed and must be dealt with in a timely manner.
3. **Variety**: Refers to the different types of data available. Traditional data types were structured and could fit in a relational database. Today's data comes in unstructured and semi-structured formats, including text, audio, video, and more.
4. **Veracity**: Refers to the quality and accuracy of data. With many forms of big data, quality and accuracy are less controllable, but big data technology now allows us to work with these types of data.
5. **Value**: The most important aspect of big data. After addressing volume, velocity, variety, and veracity, you need to derive value from your data through analysis and interpretation.

Types of Big Data

1. **Structured Data**: Data that has a defined length and format, like numbers, dates, and strings stored in relational databases. Examples include SQL databases and spreadsheets with well-defined schemas.
2. **Semi-structured Data**: Data that doesn't conform to a rigid structure but contains tags or markers to separate semantic elements. Examples include XML, JSON files, and email messages.
3. **Unstructured Data**: Data that doesn't have a pre-defined format or organization. It's typically text-heavy but may contain numbers and dates as well. Examples include social media posts, videos, and images.

Challenges in Traditional Data Processing

1. **Storage**: Traditional systems struggle to store massive amounts of data efficiently.
2. **Processing Speed**: Analyzing vast datasets requires significant computational resources that traditional systems can't provide.
3. **Data Integration**: Combining data from multiple sources with different formats and structures is complex.
4. **Scalability**: Traditional databases don't scale horizontally to accommodate growing data volumes.

5. **Cost**: Maintaining and scaling traditional systems for big data applications is prohibitively expensive.
6. **Real-time Processing**: Traditional batch processing systems can't handle the need for real-time insights.

The Hadoop Ecosystem

What is Hadoop?

Apache Hadoop is an open-source framework designed for distributed storage and processing of large datasets using simple programming models. It allows for the distributed processing of large data sets across clusters of computers using simple programming models.

The core components of Hadoop include:

- **HDFS** (Hadoop Distributed File System)
- **MapReduce** (distributed processing framework)
- **YARN** (resource management platform)

HDFS (Hadoop Distributed File System)

HDFS is the primary storage system used by Hadoop applications. It creates multiple replicas of data blocks and distributes them on compute nodes in a cluster to enable reliable and extremely rapid computations.

Key Characteristics of HDFS:

1. **Distributed Storage**: Data is stored across multiple machines in a cluster.
2. **Fault Tolerance**: HDFS replicates data blocks (default replication factor is 3) to ensure data availability even if individual nodes fail.
3. **High Throughput**: HDFS is optimized for high throughput rather than low latency access.
4. **Large Files**: Optimized for handling large files (gigabytes to terabytes).
5. **Write-Once-Read-Many**: Files in HDFS are write-once and have strictly one writer at any time.

HDFS Architecture:

- **NameNode**: The master server that manages the file system namespace and regulates access to files by clients.
- **DataNodes**: The worker nodes that store and retrieve blocks when they are told to by the NameNode.

- **Secondary NameNode**: Performs periodic checkpoints of the namespace and helps keep the size of the file system edit log within a defined limit.

Example of HDFS Commands:

```bash
# List files in HDFS
hdfs dfs -ls /

# Create a directory in HDFS
hdfs dfs -mkdir /user/example

# Copy a file from local file system to HDFS
hdfs dfs -put local_file.txt /user/example/

# Read a file from HDFS
hdfs dfs -cat /user/example/file.txt

# Remove a file from HDFS
hdfs dfs -rm /user/example/file.txt
```

MapReduce

MapReduce is a programming model and processing technique for distributed computing based on Java. It divides the processing into two phases: the Map phase and the Reduce phase.

How MapReduce Works:

1. **Map Phase**: The "map" function processes input key/value pairs to generate intermediate key/value pairs.
2. **Shuffle and Sort**: The system groups all intermediate values associated with the same intermediate key.
3. **Reduce Phase**: The "reduce" function merges these values to form a possibly smaller set of values.

Example: Word Count in MapReduce

Map Phase:

```java
public class WordCountMapper 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
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
        }
    }
}
```

Reduce Phase:

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

Advantages of MapReduce:

1. **Scalability**: Can process petabytes of data by adding more nodes to the cluster.
2. **Fault Tolerance**: If a task fails, it is automatically redistributed to other nodes.
3. **Data Locality**: Processing is moved to the data rather than moving data to the processing, reducing network congestion.

Limitations of MapReduce:

1. **Performance**: Not suitable for iterative algorithms or real-time processing.
2. **Complexity**: Writing MapReduce code can be complex compared to SQL or other high-level abstractions.

YARN (Yet Another Resource Negotiator)

YARN is the resource management layer of Hadoop. It separates the resource management and job scheduling/monitoring functions.

YARN Components:

1. **ResourceManager**: The ultimate authority that arbitrates resources among all applications in the system.
2. **NodeManager**: Per-node agent responsible for containers, monitoring resource usage, and reporting to the ResourceManager.
3. **ApplicationMaster**: Per-application entity negotiating resources from the ResourceManager and working with the NodeManager(s) to execute and monitor tasks.
4. **Container**: A collection of physical resources (RAM, CPU cores, etc.) on a single node.

YARN Workflow:

1. A client submits an application to the ResourceManager.
2. The ResourceManager allocates a container to launch the ApplicationMaster.
3. The ApplicationMaster negotiates resources with the ResourceManager.
4. The ApplicationMaster works with the NodeManagers to execute tasks.
5. Upon completion, the ApplicationMaster unregisters with the ResourceManager.

Benefits of YARN:

1. **Multi-tenancy**: Multiple applications can share the same Hadoop cluster.
2. **Resource Utilization**: Improves cluster utilization by dynamically allocating resources.
3. **Scalability**: Can scale to thousands of nodes and concurrent applications.
4. **Compatibility**: Supports MapReduce and other processing frameworks like Spark, Flink, etc.

Core Components of the Hadoop Ecosystem

Hive

Apache Hive is a data warehouse infrastructure built on top of Hadoop for providing data summarization, query, and analysis. It provides an SQL-like interface called HiveQL.

Key Features:

1. **SQL-like Interface**: Offers familiar SQL syntax for querying data stored in HDFS.
2. **Schema on Read**: Imposes a structure on data at query time, not when data is loaded.

3. **Custom Functions**: Supports User Defined Functions (UDFs) for custom processing.
4. **Storage Formats**: Compatible with various file formats like Parquet, ORC, Avro, etc.

Example HiveQL Query:

```sql
-- Create a table
CREATE TABLE sales (
    transaction_id INT,
    customer_id INT,
    product_id INT,
    quantity INT,
    price DECIMAL(10,2),
    transaction_date TIMESTAMP
);

-- Load data into the table
LOAD DATA INPATH '/user/sales/data.csv' INTO TABLE sales;

-- Query the table
SELECT product_id, SUM(quantity * price) as total_revenue
FROM sales
WHERE transaction_date >= '2023-01-01'
GROUP BY product_id
ORDER BY total_revenue DESC
LIMIT 10;
```

When to Use Hive:

- For SQL-like batch processing of large datasets
- When you need to query structured data stored in HDFS
- For ETL operations on large datasets
- When you want to provide SQL access to non-technical users

Pig

Apache Pig is a high-level platform for creating programs that run on Hadoop. It provides a scripting language called Pig Latin that abstracts the Java code needed for MapReduce.

Key Features:

1. **Dataflow Language**: Pig Latin describes a directed acyclic graph (DAG) for data processing.

2. **Built-in Operators**: Provides operations like join, filter, sort, etc.
3. **Extensibility**: Supports User Defined Functions (UDFs) in Java, Python, etc.
4. **Optimization**: Automatically optimizes execution plans.

Example Pig Latin Script:

```
pig
-- Load the data
sales = LOAD '/user/sales/data.csv' USING PigStorage(',') AS (
    transaction_id:int,
    customer_id:int,
    product_id:int,
    quantity:int,
    price:double,
    transaction_date:chararray
);

-- Filter data
filtered_sales = FILTER sales BY transaction_date >= '2023-01-01';

-- Calculate revenue by product
product_revenue = FOREACH filtered_sales GENERATE
    product_id,
    (quantity * price) AS revenue;

-- Group by product
grouped = GROUP product_revenue BY product_id;

-- Calculate total revenue per product
total_revenue = FOREACH grouped GENERATE
    group AS product_id,
    SUM(product_revenue.revenue) AS total_revenue;

-- Sort by total revenue
sorted = ORDER total_revenue BY total_revenue DESC;

-- Take top 10
top_products = LIMIT sorted 10;

-- Store the result
STORE top_products INTO '/user/sales/top_products';
```

When to Use Pig:

- For ETL (Extract, Transform, Load) operations
- When you need a procedural data flow approach rather than declarative SQL
- For rapid prototyping of big data transformations
- When working with complex, multi-stage data pipelines

HBase

Apache HBase is a distributed, scalable, big data store, modeled after Google's BigTable. It provides random, real-time read/write access to large datasets.

Key Features:

1. **Column-oriented**: Stores data in column families, which are grouped columns.
2. **Scalability**: Scales horizontally by adding more nodes.
3. **Real-time Access**: Provides low-latency access to single rows from billions of records.
4. **Sparse Data**: Efficiently stores sparse data (tables with many null values).

HBase Data Model:

- **Table**: A collection of rows.
- **Row**: A collection of column families identified by a row key.
- **Column Family**: A collection of columns.
- **Column**: A name-value pair with a timestamp.
- **Cell**: The intersection of row and column, containing versioned values.

Example HBase Shell Commands:

```bash
# Create a table
create 'users', 'info', 'activity'

# Insert data
put 'users', 'user1', 'info:name', 'John Doe'
put 'users', 'user1', 'info:email', 'john@example.com'
put 'users', 'user1', 'activity:login', '2023-05-01 10:30:00'

# Get a single row
get 'users', 'user1'

# Scan the table (get all rows)
scan 'users'

# Delete data
delete 'users', 'user1', 'info:email'

# Drop the table
disable 'users'
drop 'users'
```

When to Use HBase:

- For random, real-time read/write access to your Big Data
- When you need to store large amounts of data (billions of rows X millions of columns)
- For storing semi-structured or sparse data
- For high write and update throughput use cases
- When you need versioning of data

ZooKeeper

Apache ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services for distributed applications.

Key Features:

1. **Simple Interface**: Provides a simple API for distributed coordination.
2. **Reliability**: Implements redundancy to prevent single points of failure.
3. **Ordering**: Guarantees the order of all transactions across the cluster.
4. **Speed**: Optimized for read-heavy workloads.

ZooKeeper Data Model:

ZooKeeper organizes data in a hierarchical namespace similar to a file system, where each node (called a znode) can contain data and have children.

Common Use Cases:

1. **Configuration Management**: Storing and distributing configuration across nodes.
2. **Leader Election**: Electing a master/coordinator in a distributed system.
3. **Distributed Locking**: Implementing mutexes across distributed processes.
4. **Service Discovery**: Registering and discovering services in a distributed environment.

Example ZooKeeper Operations:

```java
// Connect to ZooKeeper ensemble
ZooKeeper zk = new ZooKeeper("localhost:2181", 3000, watcher);

// Create a znode
zk.create("/mynode", "mydata".getBytes(), ZooDefs.Ids.OPEN_ACL_UNSAFE, CreateMode.PER

// Get data from a znode
byte[] data = zk.getData("/mynode", true, null);

// Set data in a znode
zk.setData("/mynode", "newdata".getBytes(), -1);

// List children of a znode
List<String> children = zk.getChildren("/mynode", false);

// Delete a znode
zk.delete("/mynode", -1);

// Close connection
zk.close();
```

When to Use ZooKeeper:

- For coordinating distributed systems
- When you need reliable centralized configuration management
- For implementing distributed synchronization primitives
- As a building block for high-availability services

Flume

Apache Flume is a distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of log data from many different sources to a centralized data store.

Key Features:

1. **Reliability**: Provides reliable message delivery.
2. **Scalability**: Horizontally scalable and fault-tolerant.
3. **Extensibility**: Supports custom sources, sinks, and channels.
4. **Streaming Architecture**: Designed for streaming data flows.

Flume Architecture:

- **Source**: Receives data from external sources and puts it into channels.
- **Channel**: Stores events until they are consumed by sinks.
- **Sink**: Removes events from channels and puts them into an external repository or forwards to another source.
- **Agent**: A JVM process that hosts sources, channels, and sinks.

Example Flume Configuration:

```properties
# Define the components of the agent
agent.sources = webserver-log-source
agent.channels = memory-channel
agent.sinks = hdfs-sink

# Configure the source
agent.sources.webserver-log-source.type = exec
agent.sources.webserver-log-source.command = tail -F /var/log/nginx/access.log
agent.sources.webserver-log-source.channels = memory-channel

# Configure the channel
agent.channels.memory-channel.type = memory
agent.channels.memory-channel.capacity = 1000
agent.channels.memory-channel.transactionCapacity = 100

# Configure the sink
agent.sinks.hdfs-sink.type = hdfs
agent.sinks.hdfs-sink.channel = memory-channel
agent.sinks.hdfs-sink.hdfs.path = hdfs://namenode/flume/logs/%Y/%m/%d/%H
agent.sinks.hdfs-sink.hdfs.filePrefix = access_log
agent.sinks.hdfs-sink.hdfs.fileType = DataStream
agent.sinks.hdfs-sink.hdfs.writeFormat = Text
agent.sinks.hdfs-sink.hdfs.rollInterval = 3600
agent.sinks.hdfs-sink.hdfs.rollSize = 134217728
agent.sinks.hdfs-sink.hdfs.rollCount = 0
```

When to Use Flume:

- For collecting and moving large amounts of log data
- When you need a reliable way to ingest streaming data into Hadoop
- For aggregating data from multiple sources
- When you need guaranteed data delivery with transaction support

Sqoop

Apache Sqoop is a tool designed for efficiently transferring bulk data between Apache Hadoop and structured datastores such as relational databases, enterprise data warehouses, and NoSQL systems.

Key Features:

1. **Parallel Import/Export**: Transfers data in parallel for improved performance.

2. **Incremental Loads**: Supports incremental imports based on timestamps or sequence columns.
3. **Direct Import**: Can directly import data into Hive or HBase.
4. **Compression**: Supports various compression codecs during data transfer.

Common Sqoop Commands:

bash

```bash
# Import a table from MySQL to HDFS
sqoop import \
  --connect jdbc:mysql://localhost/retail \
  --username retail_user \
  --password password \
  --table customers \
  --target-dir /user/retail/customers

# Import a table from MySQL to Hive
sqoop import \
  --connect jdbc:mysql://localhost/retail \
  --username retail_user \
  --password password \
  --table customers \
  --hive-import \
  --hive-table retail.customers

# Export data from HDFS to MySQL
sqoop export \
  --connect jdbc:mysql://localhost/retail \
  --username retail_user \
  --password password \
  --table customers_backup \
  --export-dir /user/retail/customers

# Incremental import based on a timestamp column
sqoop import \
  --connect jdbc:mysql://localhost/retail \
  --username retail_user \
  --password password \
  --table transactions \
  --target-dir /user/retail/transactions \
  --incremental append \
  --check-column transaction_date \
  --last-value '2023-01-01'
```

When to Use Sqoop:

- For importing data from relational databases into Hadoop
- For exporting processed data from Hadoop back to relational databases
- When you need efficient, parallel data transfer between Hadoop and databases
- For regular incremental data imports based on timestamps or sequence IDs

Oozie

Apache Oozie is a workflow scheduler system to manage Hadoop jobs. It allows users to define a series of jobs with dependencies and schedule them to run at a specified time.

Key Features:

1. **Workflow Engine**: Defines and executes workflow jobs as Directed Acyclic Graphs (DAGs).
2. **Coordinator Engine**: Schedules workflow jobs based on time and data availability.
3. **Bundle Engine**: Groups multiple coordinator jobs into a single entity.
4. **Parameterization**: Supports parameterized job definitions.

Oozie Workflow Types:

1. **Control Flow Nodes**: Start, End, Decision, Fork, Join.
2. **Action Nodes**: MapReduce, Pig, Hive, Sqoop, Shell, Java, etc.

Example Oozie Workflow (workflow.xml):

xml

```
<workflow-app xmlns="uri:oozie:workflow:0.5" name="data-processing-workflow">
    <global>
        <job-tracker>${jobTracker}</job-tracker>
        <name-node>${nameNode}</name-node>
    </global>

    <start to="data-import"/>

    <action name="data-import">
        <sqoop xmlns="uri:oozie:sqoop-action:0.4">
            <job-tracker>${jobTracker}</job-tracker>
            <name-node>${nameNode}</name-node>
            <command>import --connect jdbc:mysql://localhost/retail --table customers
        </sqoop>
        <ok to="data-processing"/>
        <error to="fail"/>
    </action>

    <action name="data-processing">
        <pig>
            <script>process_data.pig</script>
            <param>INPUT=${targetDir}/customers</param>
            <param>OUTPUT=${targetDir}/processed_data</param>
        </pig>
        <ok to="data-analysis"/>
        <error to="fail"/>
    </action>

    <action name="data-analysis">
        <hive xmlns="uri:oozie:hive-action:0.6">
            <script>analyze_data.hql</script>
            <param>INPUT=${targetDir}/processed_data</param>
            <param>OUTPUT=${targetDir}/analysis_results</param>
        </hive>
        <ok to="end"/>
        <error to="fail"/>
    </action>

    <kill name="fail">
        <message>Workflow failed, error message: [${wf:errorMessage(wf:lastErrorNode(
    </kill>
```

```
    <end name="end"/>
</workflow-app>
```

Example Oozie Coordinator (coordinator.xml):

```xml
<coordinator-app xmlns="uri:oozie:coordinator:0.4" name="data-processing-coordinator"
                 frequency="${coord:days(1)}" start="${startTime}" end="${endTime}" ti
    <controls>
        <timeout>1440</timeout>
        <concurrency>1</concurrency>
        <execution>FIFO</execution>
    </controls>

    <datasets>
        <dataset name="input-dataset" frequency="${coord:days(1)}"
                 initial-instance="${startTime}" timezone="UTC">
            <uri-template>${nameNode}/user/retail/raw_data/${YEAR}/${MONTH}/${DAY}</u
            <done-flag>_SUCCESS</done-flag>
        </dataset>
    </datasets>

    <input-events>
        <data-in name="input" dataset="input-dataset">
            <instance>${coord:current(0)}</instance>
        </data-in>
    </input-events>

    <action>
        <workflow>
            <app-path>${nameNode}/user/oozie/workflows/data-processing</app-path>
            <configuration>
                <property>
                    <name>targetDir</name>
                    <value>${nameNode}/user/retail/processed/${YEAR}/${MONTH}/${DAY}<
                </property>
            </configuration>
        </workflow>
    </action>
</coordinator-app>
```

When to Use Oozie:

- For scheduling and coordinating Hadoop jobs
- When you need to define complex workflows with dependencies
- For time-based job scheduling
- When you need to chain multiple Hadoop jobs together
- For managing data pipeline workflows

Ambari

Apache Ambari is an open-source administration tool for provisioning, managing, and monitoring Hadoop clusters.

Key Features:

1. **Cluster Installation**: Provides step-by-step wizards for installing Hadoop services.
2. **Service Configuration**: Centralizes configuration management for all Hadoop services.
3. **Monitoring**: Offers a dashboard for monitoring cluster health and performance.
4. **Alerting**: Sends notifications about service status and health issues.
5. **REST API**: Allows programmatic access to cluster management functions.

Ambari Architecture:

- **Ambari Server**: Central server that orchestrates the installation and management of the cluster.
- **Ambari Agents**: Installed on each node in the cluster to execute tasks and report status.
- **Ambari Web UI**: Web interface for cluster administration.
- **Ambari REST API**: API for programmatic access to cluster management.

When to Use Ambari:

- For simplifying Hadoop cluster deployment
- When you need centralized cluster management and monitoring
- For enterprises that require a user-friendly interface for Hadoop operations
- When you want to automate cluster management tasks

Hadoop vs. Other Big Data Technologies

Apache Spark

Apache Spark is a unified analytics engine for large-scale data processing, with built-in modules for SQL, streaming, machine learning, and graph processing.

Comparison with Hadoop:

1. **Processing Model**:
   - Hadoop: Disk-based batch processing using MapReduce.
   - Spark: In-memory processing that can handle batch, interactive, and streaming workloads.
2. **Performance**:
   - Hadoop: Slower due to disk I/O between MapReduce steps.
   - Spark: Up to 100x faster for in-memory processing, 10x faster for disk operations.
3. **Ease of Use**:
   - Hadoop: Requires complex Java code for MapReduce jobs.
   - Spark: High-level APIs in Java, Scala, Python, and R, plus interactive shells.
4. **Iterative Processing**:
   - Hadoop: Inefficient for iterative algorithms because each iteration requires a new MapReduce job.
   - Spark: Optimized for iterative processing through in-memory data sharing.
5. **Storage**:
   - Hadoop: Tightly coupled with HDFS.
   - Spark: Can work with various storage systems (HDFS, S3, Azure Blob Storage, Cassandra, HBase, etc.).

When to Use Spark:

- For real-time stream processing
- For iterative algorithms (machine learning, graph processing)
- When you need faster data processing than traditional MapReduce
- For unified processing across batch, streaming, and interactive queries

Example Spark Code (Word Count in Python):

```python
from pyspark import SparkContext
sc = SparkContext("local", "WordCount")

# Read text file and split into words
text_file = sc.textFile("hdfs:///user/example/data.txt")
words = text_file.flatMap(lambda line: line.split(" "))

# Count occurrences of each word
word_counts = words.map(lambda word: (word, 1)).reduceByKey(lambda a, b: a + b)

# Save the results
word_counts.saveAsTextFile("hdfs:///user/example/word_count_output")
```

Apache Kafka

Apache Kafka is a distributed streaming platform that publishes and subscribes to streams of records, similar to a message queue or enterprise messaging system.

Comparison with Hadoop:

1. **Processing Model**:
   - Hadoop: Batch processing of static data.
   - Kafka: Real-time streaming platform for continuous data flow.
2. **Use Case**:
   - Hadoop: Historical data analysis and batch processing.
   - Kafka: Event streaming, real-time data pipelines, and messaging.
3. **Data Retention**:
   - Hadoop: Long-term storage of large datasets.
   - Kafka: Configurable retention of streaming data (can be short-term or long-term).
4. **Latency**:
   - Hadoop: High latency, optimized for throughput.
   - Kafka: Low latency, optimized for real-time processing.

When to Use Kafka:

- For building real-time data pipelines between systems or applications
- For building real-time streaming applications
- When you need reliable message delivery between producers and consumers

- For event sourcing and log aggregation

Example Kafka Producer (Java):

```java
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer")
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer

Producer<String, String> producer = new KafkaProducer<>(props);

ProducerRecord<String, String> record = new ProducerRecord<>("topic-name", "key", "va
producer.send(record);

producer.close();
```

Example Kafka Consumer (Java):

```java
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", "consumer-group");
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializ
props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserial

Consumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("topic-name"));

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("offset = %d, key = %s, value = %s%n",
                          record.offset(), record.key(), record.value());
    }
}
```

Apache Flink

Apache Flink is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams.

Comparison with Hadoop:

1. **Processing Model**:
   - Hadoop: Batch processing using MapReduce.
   - Flink: Stream processing first, with batch processing as a special case of stream processing.
2. **State Management**:
   - Hadoop: Stateless processing between MapReduce jobs.
   - Flink: Sophisticated state management with checkpointing for fault tolerance.
3. **Latency**:
   - Hadoop: High latency batch processing.
   - Flink: Low latency stream processing with event time semantics.
4. **Iterative Processing**:
   - Hadoop: Inefficient for iterative algorithms.
   - Flink: Native support for iterative processing with delta iterations.

When to Use Flink:

- For event-driven applications requiring stateful processing
- When you need both stream and batch processing in a single framework
- For applications requiring exactly-once semantics and sophisticated time handling
- For complex event processing (CEP) and real-time analytics

Example Flink Code (Java):

```java
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment()

// Create a stream from a source
DataStream<String> text = env.socketTextStream("localhost", 9999);

// Transform the stream
DataStream<Tuple2<String, Integer>> counts = text
    .flatMap(new Tokenizer())
    .keyBy(0)
    .timeWindow(Time.seconds(5))
    .sum(1);

// Print the results
counts.print();

env.execute("Streaming Word Count");

// User-defined function
public static class Tokenizer implements FlatMapFunction<String, Tuple2<String, Integ
    @Override
    public void flatMap(String value, Collector<Tuple2<String, Integer>> out) {
        for (String word : value.split("\\s")) {
            out.collect(new Tuple2<>(word, 1));
        }
    }
}
```

Apache Storm

Apache Storm is a distributed real-time computation system for processing unbounded streams of data.

Comparison with Hadoop:

1. **Processing Model**:
   - Hadoop: Batch processing with MapReduce.
   - Storm: Real-time stream processing with micro-batching.
2. **Latency**:
   - Hadoop: High latency, minutes to hours.
   - Storm: Low latency, sub-second processing.

3. **Guarantees**:
    - Hadoop: Batch completion guarantees.
    - Storm: At-least-once or exactly-once processing guarantees.
4. **Programming Model**:
    - Hadoop: Map and Reduce phases.
    - Storm: Spouts (sources) and Bolts (processors) forming a topology.

When to Use Storm:

- For real-time analytics and monitoring
- When you need very low latency processing
- For continuous computation on streams of data
- For applications requiring high reliability and fault tolerance

Example Storm Topology (Java):

```java
TopologyBuilder builder = new TopologyBuilder();

builder.setSpout("word-spout", new RandomSentenceSpout(), 5);
builder.setBolt("word-splitter", new SplitSentenceBolt(), 8).shuffleGrouping("word-sp
builder.setBolt("word-counter", new WordCountBolt(), 12).fieldsGrouping("word-splitte

Config conf = new Config();
conf.setDebug(true);

// Submit the topology to the cluster
if (args != null && args.length > 0) {
    conf.setNumWorkers(3);
    StormSubmitter.submitTopologyWithProgressBar(args[0], conf, builder.createTopolog
} else {
    conf.setMaxTaskParallelism(3);
    LocalCluster cluster = new LocalCluster();
    cluster.submitTopology("word-count", conf, builder.createTopology());
    Utils.sleep(10000);
    cluster.shutdown();
}
```

Presto

Presto is an open-source distributed SQL query engine for big data, designed for analytical queries against data sources of various sizes.

Comparison with Hadoop:

1. **Query Processing**:
   - Hadoop/Hive: MapReduce-based SQL queries with high latency.
   - Presto: In-memory SQL processing with much lower latency.
2. **Architecture**:
   - Hadoop: Disk-based distributed processing.
   - Presto: Memory-based distributed processing with pipelined execution.
3. **Data Sources**:
   - Hadoop: Primarily focused on HDFS data.
   - Presto: Connects to multiple data sources including HDFS, S3, Cassandra, MongoDB, etc.
4. **Use Case**:
   - Hadoop: Complex ETL jobs and batch processing.
   - Presto: Interactive analytics and ad-hoc queries.

When to Use Presto:

- For interactive analytics and ad-hoc querying
- When you need to query data across multiple storage systems
- For BI tools and dashboards requiring low-latency queries
- As an alternative to Hive for faster SQL queries

Example Presto Query:

```sql
-- Query across multiple data sources
SELECT
  orders.order_id,
  orders.customer_id,
  customers.name,
  SUM(order_items.price * order_items.quantity) AS total_amount
FROM
  mysql.sales.orders AS orders
JOIN
  postgres.customers.info AS customers
  ON orders.customer_id = customers.id
JOIN
  hive.inventory.order_items AS order_items
  ON orders.order_id = order_items.order_id
WHERE
  orders.order_date >= DATE '2023-01-01'
GROUP BY
  orders.order_id, orders.customer_id, customers.name
ORDER BY
  total_amount DESC
LIMIT 100;
```

The Big Data Pipeline

A typical big data pipeline includes the following stages:

Data Ingestion

Data ingestion is the process of obtaining and importing data for immediate use or storage in a database or data warehouse.

Ingestion Patterns:

1. **Batch Ingestion**: Collecting data periodically in batches.
2. **Real-time Ingestion**: Collecting data as it is generated.
3. **Lambda Architecture**: Combining batch and real-time processing.
4. **Kappa Architecture**: Treating all data as streams.

Tools for Data Ingestion:

1. **Apache Kafka**: For real-time data streaming.
2. **Apache Flume**: For collecting, aggregating, and moving log data.

3. **Apache Sqoop**: For transferring data between Hadoop and relational databases.
4. **Apache NiFi**: For automating data flow between systems.
5. **Logstash**: For collecting and transforming logs.

Best Practices:

1. **Validate Data**: Implement data quality checks during ingestion.
2. **Handle Failures**: Design for resilience and error handling.
3. **Scalability**: Ensure your ingestion layer can scale with data volume.
4. **Monitoring**: Implement monitoring for ingestion pipelines.
5. **Schema Evolution**: Plan for changes in data schemas over time.

Data Storage

Data storage involves persisting data in various formats and storage systems depending on the use case.

Storage Options:

1. **HDFS (Hadoop Distributed File System)**:
   - For storing large volumes of data.
   - Supports write-once-read-many operations.
   - High throughput access to data.
2. **HBase**:
   - For random, real-time read/write access.
   - Column-oriented NoSQL database.
   - Good for sparse data.
3. **Kudu**:
   - For fast analytics on rapidly changing data.
   - Combines low-latency writes with analytics performance.
4. **Cloud Storage**:
   - Amazon S3, Google Cloud Storage, Azure Blob Storage.
   - Cost-effective for long-term storage.
   - Separation of storage and compute.

File Formats:

1. **Avro**:

- Row-based format.
- Schema evolution support.
- Good for write-heavy workloads.

2. **Parquet**:
   - Column-based format.
   - Efficient compression and encoding.
   - Good for read-heavy analytical workloads.
3. **ORC (Optimized Row Columnar)**:
   - Column-based format optimized for Hive.
   - Efficient compression.
   - Good for read-heavy analytical workloads.
4. **JSON/CSV**:
   - Human-readable formats.
   - Less efficient for storage and processing.
   - Good for data interchange.

Best Practices:

1. **Choose the Right Format**: Match file format to access patterns.
2. **Partitioning**: Partition data for efficient querying.
3. **Compression**: Use appropriate compression codecs.
4. **Data Lifecycle**: Implement policies for archiving and deleting data.
5. **Backup and Recovery**: Plan for disaster recovery.

Data Processing

Data processing transforms raw data into a useful format for analysis and decision-making.

Processing Types:

1. **Batch Processing**:
   - Processing collected data in batches.
   - High throughput, higher latency.
   - Examples: Hadoop MapReduce, Spark batch jobs.
2. **Stream Processing**:
   - Processing data as it arrives.

- Lower throughput, lower latency.
- Examples: Spark Streaming, Flink, Storm.

3. **Interactive Processing**:
   - Ad-hoc queries and analysis.
   - Examples: Presto, Impala, Drill.

Processing Tools:

1. **MapReduce**:
   - For batch processing of large datasets.
   - High fault tolerance.
   - Java-based programming model.
2. **Spark**:
   - For batch, interactive, and stream processing.
   - In-memory processing for faster execution.
   - Support for SQL, machine learning, and graph processing.
3. **Flink**:
   - For real-time stream processing.
   - Stateful computations with exactly-once semantics.
   - Advanced windowing and time handling.
4. **Hive**:
   - For SQL-like queries on large datasets.
   - Converts queries to MapReduce or Tez jobs.
5. **Pig**:
   - For data flow scripting on large datasets.
   - Converts Pig Latin scripts to MapReduce jobs.

Best Practices:

1. **Choose the Right Tool**: Match processing framework to your use case.
2. **Optimize Performance**: Tune configurations for better performance.
3. **Monitor and Debug**: Implement proper logging and monitoring.
4. **Handle Failures**: Design for resilience and fault tolerance.
5. **Resource Management**: Allocate resources efficiently using YARN or similar.

Data Analytics

Data analytics involves examining data to draw conclusions and support decision-making.

Analytics Types:

1. **Descriptive Analytics**:
   - Understanding what happened.
   - Examples: Reports, dashboards, visualizations.
2. **Diagnostic Analytics**:
   - Understanding why it happened.
   - Examples: Drill-down analysis, correlation analysis.
3. **Predictive Analytics**:
   - Forecasting what might happen.
   - Examples: Machine learning models, forecasting.
4. **Prescriptive Analytics**:
   - Recommending actions.
   - Examples: Optimization algorithms, recommendation engines.

Analytics Tools:

1. **Hive**:
   - For SQL-based analytics on Hadoop data.
   - Supports complex queries and transformations.
2. **Spark SQL**:
   - For interactive SQL queries on Spark.
   - Integrates with other Spark components.
3. **Presto**:
   - For interactive queries across multiple data sources.
   - Low-latency SQL engine.
4. **Druid**:
   - For real-time analytics on large datasets.
   - Optimized for time series data.
5. **Machine Learning Libraries**:
   - Spark MLlib: For scalable machine learning.
   - Mahout: For machine learning on Hadoop.

- TensorFlow, PyTorch: For deep learning.

Best Practices:

1. **Data Quality**: Ensure data is clean and reliable.
2. **Feature Engineering**: Create meaningful features for ML models.
3. **Model Validation**: Validate models using appropriate metrics.
4. **Interpretability**: Ensure results can be explained to stakeholders.
5. **Continuous Improvement**: Iteratively refine analytics over time.

Data Visualization

Data visualization is the graphical representation of information and data to understand patterns, trends, and insights.

Visualization Tools:

1. **Tableau**:
   - For interactive dashboards and visualizations.
   - Easy-to-use interface for non-technical users.
2. **Power BI**:
   - For business intelligence and reporting.
   - Integration with Microsoft products.
3. **Superset**:
   - Open-source business intelligence web application.
   - Interactive exploration and visualization.
4. **Kibana**:
   - For visualizing Elasticsearch data.
   - Real-time analytics and monitoring.
5. **Grafana**:
   - For monitoring and observability.
   - Time-series data visualization.
6. **Custom Applications**:
   - Web applications using D3.js, Plotly, etc.
   - Tailored to specific business needs.

Best Practices:

1. **Choose the Right Visualization**: Match the visualization to the data type and message.
2. **Keep It Simple**: Avoid cluttering visualizations with unnecessary elements.
3. **Use Color Effectively**: Use color to highlight important insights.
4. **Provide Context**: Include relevant context for proper interpretation.
5. **Interactive Exploration**: Allow users to drill down and explore data.

Security and Governance

Kerberos Authentication

Kerberos is a network authentication protocol that provides strong authentication for client/server applications using secret-key cryptography.

How Kerberos Works in Hadoop:

1. **Authentication**: User authenticates to the Kerberos Key Distribution Center (KDC).
2. **Ticket Granting**: KDC issues a Ticket Granting Ticket (TGT) to the user.
3. **Service Access**: User uses the TGT to request service tickets for Hadoop services.
4. **Service Interaction**: User presents service tickets to access Hadoop services.

Implementing Kerberos:

```xml
<!-- core-site.xml -->
<property>
    <name>hadoop.security.authentication</name>
    <value>kerberos</value>
</property>
<property>
    <name>hadoop.security.authorization</name>
    <value>true</value>
</property>

<!-- hdfs-site.xml -->
<property>
    <name>dfs.namenode.kerberos.principal</name>
    <value>hdfs/_HOST@REALM</value>
</property>
<property>
    <name>dfs.namenode.keytab.file</name>
    <value>/etc/hadoop/conf/hdfs.keytab</value>
</property>
```

Best Practices:

1. **Keytab Management**: Securely store and manage keytab files.
2. **Principal Naming**: Use consistent naming conventions for principals.
3. **Ticket Renewal**: Configure automated ticket renewal.
4. **Integration**: Integrate with enterprise directory services like Active Directory.

Apache Ranger

Apache Ranger provides a centralized security framework to manage fine-grained access control over Hadoop components.

Key Features:

1. **Centralized Administration**: Single console for security policies across components.
2. **Fine-Grained Authorization**: Control access at column, row, or cell level.
3. **Auditing**: Comprehensive audit logging for compliance.
4. **Integration**: Works with various Hadoop components (HDFS, Hive, HBase, etc.).

Example Ranger Policy:

```json
{
  "service": "hdfs",
  "name": "sales-data-access",
  "resources": {
    "path": {
      "values": ["/data/sales"],
      "isRecursive": true
    }
  },
  "policyItems": [
    {
      "users": ["sales_analyst"],
      "groups": ["sales_team"],
      "accesses": [
        {"type": "read", "isAllowed": true},
        {"type": "write", "isAllowed": false}
      ]
    }
  ]
}
```

Best Practices:

1. **Least Privilege**: Grant minimum necessary permissions.
2. **Group-Based Policies**: Use groups rather than individual users.
3. **Tag-Based Policies**: Implement tag-based access control for easier management.
4. **Regular Audits**: Review access logs and permissions regularly.

Apache Atlas

Apache Atlas provides metadata management and governance capabilities for Hadoop environments.

Key Features:

1. **Metadata Repository**: Central repository for technical and business metadata.
2. **Data Classification**: Classify data for governance and compliance.
3. **Lineage Tracking**: Track data flow across systems.
4. **Integration**: Works with various Hadoop components and external systems.

Example Atlas Use Cases:

1. **Data Discovery**: Search and browse metadata about data assets.
2. **Data Lineage**: Track data transformation across systems.
3. **Compliance**: Classify data according to compliance requirements.
4. **Impact Analysis**: Analyze the impact of changes to data structures.

Best Practices:

1. **Consistent Tagging**: Develop a consistent taxonomy for tagging data.
2. **Automated Classification**: Automate classification of sensitive data.
3. **Integration**: Integrate Atlas with data quality tools.
4. **Governance Process**: Establish clear data governance processes.

Fault Tolerance

Fault tolerance is the ability of a system to continue operating despite failures in its components.

Fault Tolerance Features in Hadoop:

1. **Data Replication**: HDFS replicates data blocks across multiple nodes.
2. **Automatic Failover**: NameNode high availability through standby NameNodes.
3. **Speculative Execution**: Hadoop can run backup tasks for slow-running tasks.
4. **Task Retries**: Failed tasks are automatically retried.

Implementing Fault Tolerance:

```xml
<!-- hdfs-site.xml for replication -->
<property>
    <name>dfs.replication</name>
    <value>3</value>
</property>

<!-- hdfs-site.xml for NameNode HA -->
<property>
    <name>dfs.nameservices</name>
    <value>mycluster</value>
</property>
<property>
    <name>dfs.ha.namenodes.mycluster</name>
    <value>nn1,nn2</value>
</property>
<property>
    <name>dfs.namenode.rpc-address.mycluster.nn1</name>
    <value>namenode1:8020</value>
</property>
<property>
    <name>dfs.namenode.rpc-address.mycluster.nn2</name>
    <value>namenode2:8020</value>
</property>
```

Best Practices:

1. **Regular Testing**: Test failure scenarios and recovery procedures.
2. **Monitoring**: Implement comprehensive monitoring for early failure detection.
3. **Backup Strategies**: Implement regular backups of critical data.
4. **Documentation**: Document failure recovery procedures.

Scalability

Scalability is the capability of a system to handle a growing amount of work by adding resources.

Scalability Features in Hadoop:

1. **Horizontal Scaling**: Add more nodes to increase capacity.
2. **Data Partitioning**: Distribute data across nodes for parallel processing.
3. **Resource Management**: Efficient allocation of resources with YARN.
4. **Elastic Scaling**: Scale resources up or down based on demand.

Implementing Scalability:

```xml
<!-- yarn-site.xml for resource management -->
<property>
    <name>yarn.resourcemanager.scheduler.class</name>
    <value>org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity.CapacityS
</property>
<property>
    <name>yarn.scheduler.capacity.maximum-am-resource-percent</name>
    <value>0.5</value>
</property>
```

Best Practices:

1. **Capacity Planning**: Regularly assess current and future resource needs.
2. **Load Balancing**: Distribute workloads evenly across the cluster.
3. **Performance Monitoring**: Monitor cluster performance metrics.
4. **Resource Allocation**: Allocate resources based on workload priorities.
5. **Data Locality**: Optimize data placement for processing efficiency.

Setting Up a Hadoop Cluster

Single-Node Setup

A single-node Hadoop setup is useful for development, testing, and learning purposes.

Prerequisites:

1. Java 8 or later
2. SSH
3. Adequate RAM (at least 4GB) and storage

Installation Steps:

1. **Install Java**:

   ```bash
   sudo apt-get update
   sudo apt-get install openjdk-8-jdk
   ```

2. **Configure SSH**:

```bash
ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
chmod 0600 ~/.ssh/authorized_keys
```

3. **Download Hadoop**:

```bash
wget https://downloads.apache.org/hadoop/common/hadoop-3.3.4/hadoop-3.3.4.tar.gz
tar -xzvf hadoop-3.3.4.tar.gz
mv hadoop-3.3.4 /usr/local/hadoop
```

4. **Configure Hadoop**: Edit `/usr/local/hadoop/etc/hadoop/hadoop-env.sh`:

```bash
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
```

Edit `/usr/local/hadoop/etc/hadoop/core-site.xml`:

```xml
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://localhost:9000</value>
    </property>
</configuration>
```

Edit `/usr/local/hadoop/etc/hadoop/hdfs-site.xml`:

xml

```xml
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
    <property>
        <name>dfs.namenode.name.dir</name>
        <value>/usr/local/hadoop/data/namenode</value>
    </property>
    <property>
        <name>dfs.datanode.data.dir</name>
        <value>/usr/local/hadoop/data/datanode</value>
    </property>
</configuration>
```

Edit `/usr/local/hadoop/etc/hadoop/mapred-site.xml`:

xml

```xml
<configuration>
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
</configuration>
```

Edit `/usr/local/hadoop/etc/hadoop/yarn-site.xml`:

xml

```xml
<configuration>
    <property>
        <name>yarn.nodemanager.aux-services</name>
        <value>mapreduce_shuffle</value>
    </property>
</configuration>
```

5. **Format the NameNode**:

bash

```bash
mkdir -p /usr/local/hadoop/data/namenode
mkdir -p /usr/local/hadoop/data/datanode
/usr/local/hadoop/bin/hdfs namenode -format
```

6. **Start Hadoop Services**:

```bash
/usr/local/hadoop/sbin/start-dfs.sh
/usr/local/hadoop/sbin/start-yarn.sh
```

7. **Verify Installation**:

```bash
jps  # Should show NameNode, DataNode, ResourceManager, NodeManager, etc.
```

Access the web interfaces:

- NameNode: http://localhost:9870
- ResourceManager: http://localhost:8088

Multi-Node Setup

A multi-node Hadoop cluster provides scalability and fault tolerance for production environments.

Cluster Architecture:

- **Master Node**: Runs NameNode, ResourceManager, and related services.
- **Worker Nodes**: Run DataNode, NodeManager, and execute tasks.

Prerequisites:

1. Multiple servers with Java installed
2. Network connectivity between nodes
3. SSH access across all nodes
4. Consistent user accounts across all nodes

Installation Steps:

1. **Install Java on all nodes**:

```bash
sudo apt-get update
sudo apt-get install openjdk-8-jdk
```

2. **Configure SSH on all nodes**:

```bash
# On master node
ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa

# Copy the key to all worker nodes
ssh-copy-id user@worker1
ssh-copy-id user@worker2
# ... and so on
```

3. **Download and extract Hadoop on all nodes**:

```bash
wget https://downloads.apache.org/hadoop/common/hadoop-3.3.4/hadoop-3.3.4.tar.gz
tar -xzvf hadoop-3.3.4.tar.gz
mv hadoop-3.3.4 /usr/local/hadoop
```

4. **Configure Hadoop on all nodes**: Edit `/usr/local/hadoop/etc/hadoop/hadoop-env.sh`:

```bash
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
```

Edit `/etc/hosts` on all nodes:

```
192.168.1.100 master
192.168.1.101 worker1
192.168.1.102 worker2
# ... and so on
```

Edit `/usr/local/hadoop/etc/hadoop/core-site.xml`:

```xml
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://master:9000</value>
    </property>
</configuration>
```

Edit `/usr/local/hadoop/etc/hadoop/hdfs-site.xml`:

xml

```xml
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>3</value>
    </property>
    <property>
        <name>dfs.namenode.name.dir</name>
        <value>/usr/local/hadoop/data/namenode</value>
    </property>
    <property>
        <name>dfs.datanode.data.dir</name>
        <value>/usr/local/hadoop/data/datanode</value>
    </property>
</configuration>
```

Edit `/usr/local/hadoop/etc/hadoop/mapred-site.xml`:

xml

```xml
<configuration>
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
    <property>
        <name>mapreduce.jobhistory.address</name>
        <value>master:10020</value>
    </property>
    <property>
        <name>mapreduce.jobhistory.webapp.address</name>
        <value>master:19888</value>
    </property>
</configuration>
```

Edit `/usr/local/hadoop/etc/hadoop/yarn-site.xml`:

```xml
<configuration>
    <property>
        <name>yarn.nodemanager.aux-services</name>
        <value>mapreduce_shuffle</value>
    </property>
    <property>
        <name>yarn.resourcemanager.hostname</name>
        <value>master</value>
    </property>
</configuration>
```

Create and edit `/usr/local/hadoop/etc/hadoop/workers`:

```
worker1
worker2
# ... and so on
```

5. **Format the NameNode** (on master node only):

```bash
mkdir -p /usr/local/hadoop/data/namenode
/usr/local/hadoop/bin/hdfs namenode -format
```

6. **Create DataNode directories** (on all worker nodes):

```bash
mkdir -p /usr/local/hadoop/data/datanode
```

7. **Start Hadoop Services** (on master node):

```bash
/usr/local/hadoop/sbin/start-dfs.sh
/usr/local/hadoop/sbin/start-yarn.sh
```

8. **Verify Installation**:

```bash
jps  # On master: should show NameNode, ResourceManager, etc.
     # On workers: should show DataNode, NodeManager, etc.
```

Access the web interfaces:

- NameNode: http://master:9870
- ResourceManager: http://master:8088

Basic Hadoop Commands

HDFS Commands:

```bash
# List files and directories
hdfs dfs -ls /

# Create a directory
hdfs dfs -mkdir -p /user/example/data

# Copy from local file system to HDFS
hdfs dfs -put local_file.txt /user/example/data/

# Copy from HDFS to local file system
hdfs dfs -get /user/example/data/file.txt local_copy.txt

# View file content
hdfs dfs -cat /user/example/data/file.txt

# View file details
hdfs dfs -stat /user/example/data/file.txt

# Check disk usage
hdfs dfs -du -h /user/example

# Delete a file or directory
hdfs dfs -rm /user/example/data/file.txt
hdfs dfs -rm -r /user/example/data

# Set replication factor for a file
hdfs dfs -setrep -w 3 /user/example/data/important_file.txt

# Check file integrity
hdfs fsck /user/example/data/file.txt
```

YARN Commands:

```bash
# View all applications
yarn application -list

# Kill an application
yarn application -kill application_1234567890_0001

# View the status of an application
yarn application -status application_1234567890_0001

# View cluster status
yarn node -list

# View resource usage
yarn top
```

Hadoop Administration Commands:

```bash
# Report of file system usage
hdfs dfsadmin -report

# Enter safe mode
hdfs dfsadmin -safemode enter

# Leave safe mode
hdfs dfsadmin -safemode leave

# Check safe mode status
hdfs dfsadmin -safemode get

# Refresh nodes
hdfs dfsadmin -refreshNodes

# Balance data across nodes
hdfs balancer -threshold 10
```

Configuration Best Practices

1. **Memory Configuration**:

xml

```xml
<!-- yarn-site.xml -->
<property>
    <name>yarn.nodemanager.resource.memory-mb</name>
    <value>24576</value>  <!-- 24GB for NodeManager -->
</property>
<property>
    <name>yarn.scheduler.maximum-allocation-mb</name>
    <value>
```