**Big Data refers to extremely large, complex, and fast-growing datasets that cannot be processed using traditional tools and databases.**

| **V** | **Meaning** | **Example** |
| --- | --- | --- |
| **Volume** | **Huge amounts of data** | **Petabytes of logs** |
| **Velocity** | **High speed of data generation** | **Real-time tweets** |
| **Variety** | **Multiple data formats** | **Text, video, sensor data** |
| **Veracity** | **Data trustworthiness** | **Incomplete or conflicting logs** |
| **Value** | **Business insights and impact** | **Sales forecasting, personalization** |

**WHAT IS HADOOP?**

**Apache Hadoop** is an **open-source framework** used for **distributed storage and processing of large datasets** using a cluster of computers.

It works on a **scale-out** model: rather than scaling up (buying a bigger machine), you scale out by adding more machines (nodes).

**⚙️ HOW HADOOP WORKS**

Hadoop works based on **two main ideas**:

1. **Split** the data into smaller chunks and **store** them across a cluster (HDFS).
2. **Process** those chunks in parallel using a computation engine (MapReduce or others like Spark).

**HADOOP ARCHITECTURE COMPONENTS**

**1. HDFS (Hadoop Distributed File System) – *Storage Layer***

* Stores large files by breaking them into blocks (default: 128 MB or 256 MB).
* Stores each block **across multiple nodes** for fault tolerance.

Components:

* **NameNode**: Master – stores metadata (file names, block locations).
* **DataNode**: Worker – stores actual blocks of data.

**2. YARN (Yet Another Resource Negotiator) – *Resource Manager***

* Manages resources and job scheduling.

Components:

* **ResourceManager** – master for resource allocation.
* **NodeManager** – runs on each node to manage resources and report status.

**3. MapReduce – *Processing Layer***

* A programming model for **processing data in parallel** across the cluster.
  + **Map**: Takes input and converts it into key-value pairs.
  + **Reduce**: Aggregates the key-value pairs.

**4. Common Utilities**

* Shared libraries and modules used by all Hadoop components.

**WORKFLOW OF A HADOOP JOB**

1. **Client submits a job**.
2. HDFS **splits and stores** the data across nodes.
3. **YARN assigns resources** and starts MapReduce tasks on the nodes.
4. **Map tasks** process the data locally.
5. **Reduce tasks** aggregate intermediate results.
6. Final output is stored in HDFS.

**COMMON USE CASES OF HADOOP**

| **Use Case** | **Description** |
| --- | --- |
| **Data Warehousing** | Storing and querying petabyte-scale data |
| **Log Analysis** | Parsing and analyzing logs from servers or apps |
| **ETL Pipelines** | Transforming large datasets for downstream use |
| **Machine Learning (Batch)** | Used with tools like Mahout or Spark for ML models |
| **Web Indexing** | Powering search engines (e.g., indexing the web) |
| **Recommendation Engines** | Based on user data patterns |
| **IoT Data Processing** | Storing and analyzing time-series sensor data |

**KEY ADVANTAGES OF HADOOP**

* **Scalability**: Easily add more machines.
* **Fault Tolerance**: Replicates data blocks across nodes.
* **Cost-Effective**: Runs on commodity hardware.
* **Flexibility**: Can store structured, semi-structured, or unstructured data.

**LIMITATIONS**

* **High latency** (batch processing only – not real-time).
* **Complex** to manage manually (now often abstracted by tools like Spark, Hive, etc.).
* **Not suitable for small, frequent jobs** (unlike Spark or Flink).

**Summary**

* **HDFS** → Storage
* **YARN** → Resource Management
* **MapReduce** → Computation
* **Hadoop** is best for **large-scale, batch-oriented data processing** and forms the **foundation for many modern big data systems**.

**Replication in Hadoop (HDFS)**

**💡 What is it?**

Replication is the **process of storing multiple copies** of each block of data across different DataNodes in a Hadoop cluster.

**⚙️ Key Features:**

* **Default replication factor**: 3
* **Ensures data availability** even when nodes fail.
* **Replication policy**:
  + 1st copy on the local rack
  + 2nd copy on a different node in the same rack
  + 3rd copy on a different rack

**✅ Example:**

You upload a 256 MB file, with block size 128 MB. Hadoop splits it into:

* Block A
* Block B

With replication = 3:

* Block A → stored on Node1, Node2, Node3
* Block B → stored on Node4, Node5, Node6

**🧠 Why Replication?**

| **Goal** | **Benefit** |
| --- | --- |
| Fault tolerance | Node down? No problem — use replica. |
| Load balancing | Read from nearest node for speed. |
| Data reliability | Avoid data loss due to disk crash. |

**🔄 MapReduce in Hadoop**

**💡 What is it?**

**MapReduce** is the **processing engine** in Hadoop that lets you process large data sets in **parallel** across many nodes.

**⚙️ Architecture:**

1. **Input**: Data from HDFS (split into blocks)
2. **Map Phase**:
   * Each block is processed by a **Mapper**
   * Produces key-value pairs (<key, value>)
3. **Shuffle and Sort**:
   * Groups values by key across all Mappers
   * Moves data to appropriate **Reducers**
4. **Reduce Phase**:
   * Reducers process grouped data
   * Produces final output written back to HDFS

**✅ Features of MapReduce:**

| **Feature** | **Description** |
| --- | --- |
| **Parallel processing** | Processes data in distributed fashion |
| **Fault tolerance** | Failed tasks are re-executed automatically |
| **Data locality** | Processes data on the node where it is stored |

**🧠 Key Points:**

* **MapReduce processes the data that is replicated in HDFS**
* It benefits from **data locality** — compute moves to data, not vice versa.
* Works great for **batch processing** (not ideal for real-time)

**🧩 Replication & MapReduce: How They Work Together**

| **Component** | **Role** |
| --- | --- |
| **HDFS** | Stores data with replication |
| **MapReduce** | Reads replicated data from HDFS, processes it in parallel |
| **Fault Tolerance** | If a node fails, MapReduce re-runs the task on another node with replica data |

**WHAT IS APACHE SPARK?**

Apache Spark is an **open-source, distributed processing system** used for big data workloads. It offers in-memory computing capabilities, which makes it **much faster** than traditional big data engines like MapReduce.

**⚙️ SPARK ARCHITECTURE**

**🔹 1. Driver Program**

* The **main program** that defines transformations and actions.
* Responsible for converting user code into a **Directed Acyclic Graph (DAG)** and submitting it to the cluster.

**🔹 2. Cluster Manager**

* Allocates resources across Spark applications.
* Types:
  + Standalone
  + YARN (Hadoop)
  + Mesos
  + Kubernetes

**🔹 3. Executors**

* Run on worker nodes.
* **Execute tasks** assigned by the driver.
* Store data in memory/disk for reuse.

**🔹 4. Tasks and Jobs**

* A **job** is triggered by an action (e.g., count()).
* A job is divided into **stages**, which contain **tasks**.
* Tasks are distributed across executors.

**🔁 WORKING OF SPARK**

**Step-by-step:**

1. **Code Submission**:
   * You write a Spark job in Python, Scala, or Java.
2. **DAG Creation**:
   * Spark constructs a DAG of stages and tasks.
3. **Task Scheduling**:
   * Spark sends tasks to executors based on data locality.
4. **Execution**:
   * Executors process data and cache if required.
5. **Result Return**:
   * Final results are collected back to the driver or written to storage (HDFS, S3, etc.).

**🔄 DATA FLOW IN SPARK**

plaintext

CopyEdit

Data Source → RDD/DataFrame → Transformations → DAG → Stages/Tasks → Executors → Output

**💡 KEY FEATURES**

| **Feature** | **Description** |
| --- | --- |
| **In-memory computing** | Keeps data in RAM between operations (very fast) |
| **Lazy evaluation** | Executes only when an action is called |
| **Fault tolerance** | Recovers lost partitions via lineage |
| **Scalability** | Works on a single machine or 1000-node cluster |
| **Language support** | Python, Scala, Java, R |

**🧰 USE CASES OF SPARK**

| **Domain** | **Use Case** |
| --- | --- |
| **ETL** | Cleaning, transforming large datasets |
| **Machine Learning** | Training models using MLlib |
| **Real-Time Analytics** | Spark Streaming for real-time dashboards |
| **Data Lake Processing** | Query huge datasets with Spark SQL |
| **Graph Processing** | Analyze social networks or links (GraphX) |

**🔍 SPARK VS MAPREDUCE**

| **Feature** | **Spark** | **MapReduce** |
| --- | --- | --- |
| Speed | In-memory (10x–100x faster) | Disk-based (slower) |
| Ease of use | APIs in Python/Scala | Java only (complex) |
| Reusability | Caches data between steps | Reloads data every step |
| Libraries | SQL, MLlib, GraphX, Streaming | None (basic compute only) |
|  |  |  |

**What is Apache Kafka?**

Apache Kafka is a **distributed event streaming platform** used to **ingest, store, process, and forward high volumes of data** in real-time. It’s ideal for building real-time data pipelines and streaming applications.

**🧠 Kafka Architecture**

Kafka has a simple but powerful distributed architecture based on the **publish-subscribe** model:

**🔹 1. Producer**

* Sends data (messages) to Kafka topics.
* Can push data at high volume and speed.

**🔹 2. Topic**

* Logical channels to which producers send data and from which consumers read.
* Topics are split into **partitions** for scalability and parallelism.

**🔹 3. Broker**

* A Kafka server that stores topic data.
* Kafka clusters consist of multiple brokers.

**🔹 4. Partition**

* Each topic is divided into partitions.
* Enables parallel processing and high throughput.
* Each partition has an **offset** (unique ID for each message).

**🔹 5. Consumer**

* Subscribes to topics and reads messages.
* Can be grouped into **consumer groups** for load balancing.

**🔹 6. Zookeeper (Kafka ≤ 2.x)**

* Manages cluster metadata (leader election, health checks).
* Kafka is gradually moving away from ZooKeeper with **KRaft (Kafka Raft)** in newer versions.

**⚙️ Working of Kafka**

Here’s the flow:

plaintext

CopyEdit

Producers → Kafka Brokers (Topics → Partitions) → Consumers

**Step-by-step:**

1. **Producer** sends a message to a **topic**.
2. Kafka assigns the message to a **partition** within that topic.
3. The **broker** stores the message in the partition, appending it to a log.
4. **Consumer** pulls the message from the topic partition using the **offset**.
5. Kafka retains messages for a configurable time, even after they're read.

**📦 Data Retention & Durability**

Kafka can **persist messages to disk**, and due to **replication** across brokers, data is **fault-tolerant** and **highly available**.

**💡 Key Features**

| **Feature** | **Description** |
| --- | --- |
| High throughput | Handles millions of messages per second |
| Scalability | Scales horizontally with more brokers/partitions |
| Durability | Stores data on disk, with replication |
| Real-time | Delivers data with low latency |
| Fault-tolerance | Replicated partitions prevent data loss |