01 Big Data History and Evolution

📌 What is Big Data?

Big Data refers to datasets that are:

- **Too large**
- **Generated too fast**
- **Too complex or unstructured**

to be stored, processed, or analyzed using traditional systems like single-machine databases (MySQL, Oracle).

📊 The 3Vs of Big Data

1. Volume

Large quantities of data generated from multiple sources.

📍 Examples:

- YouTube: **500 hours uploaded every minute**
- Facebook: billions of messages, comments, and images daily
- Netflix: user watch history and streaming logs

2. Velocity

Speed at which data is generated and needs to be processed.

📍 Examples:

- Stock price updates in **milliseconds**
- Uber ride tracking and surge price calculation in real time
- Payment fraud detection (instant decisions)

3. Variety

Different formats and structures of data.

📌 Types of Data:

Type	Example
Structured	SQL tables, transactional data
Semi structured	JSON, XML, logs
Unstructured	Images, videos, PDFs, audio, emails

📍 Example Use Case:

Amazon:

- Orders and transactions → Structured
- Customer reviews → Text (Unstructured)
- Website clickstream logs → JSON (Semi structured)

🔥 Why Big Data Engineering Need New Technology?

Traditional systems struggle because they:

- Cannot handle **unstructured data**
- Have limited **scalability**
- Cannot process **real-time data streams**

🏁 Summary

> If the data is huge, fast-moving, and comes in multiple formats, it is considered Big Data.

Such workloads require **distributed** platforms like **Hadoop** and **Apache Spark** for processing.

📦 Monolithic vs Distributed Systems

🏗️ Monolithic Systems

- **Single data storage**
- **Single processing layer**
- **Vertical scaling**
- **Simple consistency but limited capacity**

A monolithic system keeps all its data in one place, usually a single database running on one machine. Because everything is centralized, managing data consistency, backups, and access patterns is straightforward.

However, as the amount of data grows, the system becomes slower since one machine handles all reads, writes, and processing. Scaling is done by upgrading the same server with more RAM, CPU, or storage, but this approach has a natural physical limit.

Monolithic systems work well when data volumes are small or moderate, and workloads are mostly transactional rather than large-scale analytics or real-time processing.

```
┌─────────────────────────┐
│      Application        │
└───────────┬─────────────┘
            │
            │
┌───────────▼─────────────┐
│ Single Database/Storage │
└─────────────────────────┘

➤ One machine handles storage + processing
➤ Scaling = Add more RAM/CPU (Vertical Scaling)
```

🌍 Distributed Systems

- **Data is partitioned across multiple machines**

- **Parallel data processing**
- **Horizontal scaling**
- **High availability and fault tolerance through replication**

In a distributed system, data is stored across many machines (nodes) instead of one. This allows the system to scale as data grows because new machines can be added without replacing existing ones.

Processing is also distributed, meaning multiple nodes can work on different parts of the data at the same time, leading to faster performance for large workloads. Replication ensures that the system continues to operate even if one node fails, providing reliability and fault tolerance.

Distributed systems are ideal for Big Data environments where data volume, speed, and variety exceed what a single machine can handle.

A distributed system consists of

1- Distributed storage

2- Distributed / Parallel Processing

3- Scalable

```
            ┌─────────────────────────────┐
            │   Application   │
            └───────────────┬─────────────┘
                  │
      ┌───────────────────────┴─────────────────────────┐
      │                  │
  ┌─────────────▼─────────────┐      ┌─────────────▼─────────
──┐
  │  Node 1      │        │  Node 2      │
  │ (Data + Compute)  │        │ (Data + Compute)  │
```

```
   └────────────────┬────────────────┘     └────────────────┬──────────
──┘
              │                 │
              │                 │
   ┌──────────────▼──────────────┐     ┌──────────────▼──────────
───┐
   │  Node 3        │    ...    │  Node N        │
   │ (Data + Compute)  │         │  (Data + Compute)  │
   └─────────────────────────────┘    └──────────────────────────
─────┘

   ➤ Data is partitioned + replicated
   ➤ Processing happens in parallel
   ➤ Scaling = Add more machines (Horizontal Scaling)
```

🧠 Summary

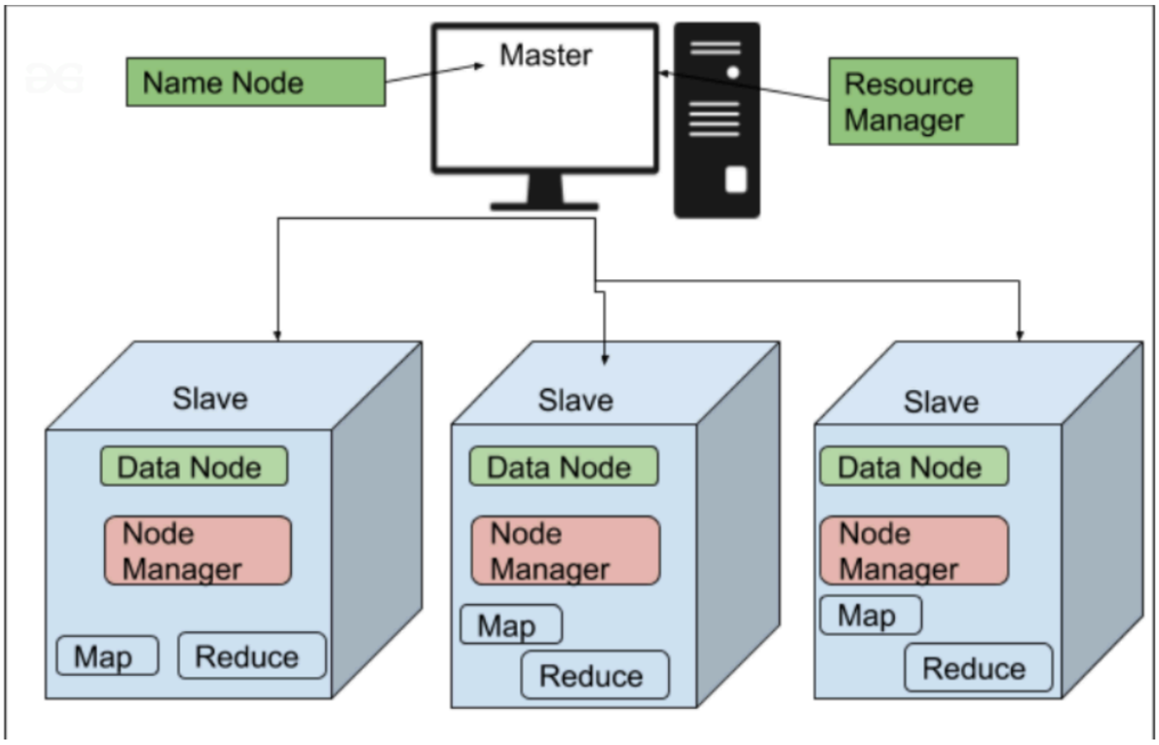
- **Monolithic systems:** one machine, simpler, but limited for large-scale data.
- **Distributed systems:** many machines, scalable, fault tolerant, and suitable for Big Data.

When I say resources it means : Storage , CPU (no of cores) , RAM → Compute is CPU + RAM

🔷 What is Hadoop

Hadoop is an open source framework used to store and process large-scale data across a cluster of machines. It was designed to solve the limitations of traditional systems that could not handle growing data volumes, especially when data became too large for a single server.

Hadoop provides distributed storage, distributed processing, and fault tolerance using commodity hardware, which makes it cost-effective and scalable. It became one of the foundational technologies in the Big Data ecosystem.



🔷 Core Components of Hadoop

Hadoop is built around three main components that work together to store and process data efficiently.

1) HDFS (Hadoop Distributed File System)

- Distributed file storage layer
- Breaks large files into blocks and stores across multiple machines
- Provides replication for fault tolerance

HDFS is the storage layer of Hadoop. Instead of storing a file on one machine, HDFS splits the file into blocks and distributes them across a cluster. Each block is also replicated (usually 3 copies) so the system continues to work even if one machine fails. This approach allows Hadoop to store massive datasets and ensures durability and high availability.

2) YARN (Yet Another Resource Negotiator)

- Manages cluster resources like CPU and memory
- Schedules and monitors jobs running across the cluster
- Enables multiple processing frameworks (MapReduce, Spark, Hive, etc.)

YARN is the resource management layer. It decides which application runs where and how much compute it gets. Without YARN, a cluster would be difficult to coordinate.

3) MapReduce

- Distributed processing engine
- Processes data in parallel using two phases: Map and Reduce
- Best suited for large batch processing

MapReduce is Hadoop's original computation framework. It works by splitting a job into smaller tasks that run in parallel across machines. The **Map** phase processes the input and extracts key-value pairs, while the **Reduce** phase aggregates or summarizes results. Although MapReduce is slower compared to newer engines like Spark, it was the first framework to enable large-scale distributed data processing.

```
          ┌──────────────────────────────────┐
          │      Hadoop         │
          └──────────────────┬───────────────┘
                  │
  ┌────────────────────────────────────────┼────────────────
──────────────────────┐
  │                │                │
  │                │                │
  ┌─────▼─────┐         ┌─────▼─────┐          ┌─────▼─────
─┐
  │ HDFS  │         │ YARN  │        │ MapReduce │
  └─────┬─────┘         └─────┬─────┘         └─────┬─────┘
    │              │              │
    │              │              │
  ┌─────▼──────────────────────────┐ ┌─────▼────────────────
```

```
────┐ ┌────▼───────────────────────┐
│ Distributed Storage  │ │ Resource Management   │ │ Parallel Process
ing   │
│ (stores data across  │ │ (allocates CPU & memory) │ │ (executes map
→reduce jobs)│
│  cluster nodes)    │ │           │ │         │
└──────────────────────┘ └──────────────────────
────┘ └───────────────────────┘
```

Challenges with Hadoop

Hadoop solved early Big Data problems, but it has several limitations that make it less suitable for modern workloads.

- **Slow Processing:** Hadoop uses disk-based processing, making jobs much slower than modern in-memory engines like Spark.
- **Not Real-Time:** It's designed for batch processing and struggles with real-time or streaming use cases.
- **Difficult to Maintain:** Running and managing Hadoop clusters requires significant operational effort and tuning.
- **Complex Ecosystem:** The Hadoop stack is large and can be difficult to integrate, manage, and learn. (Pig, Hive, Oozie, HBase, Sqoop)
- **Hard to Develop:** Writing MapReduce code requires low-level programming, making development slower and less flexible for data engineers.
- **Infrastructure Overhead:** Hadoop clusters require storage replication and large on-premises hardware, increasing cost and operational responsibility.

🔷 Overview of Apache Spark

Apache Spark is a modern distributed data processing engine designed to process large-scale data quickly and efficiently.

Unlike Hadoop MapReduce, Spark performs most operations in **memory**, which makes it significantly faster for batch processing, machine learning, SQL analytics, and streaming workloads.

Spark can run on multiple cluster managers like YARN, Kubernetes, or standalone mode and can read data from HDFS, S3, ADLS, Kafka, and many other systems.

🔷 Key Highlights

- **In-memory processing → much faster than MapReduce**
- **Unified engine for batch, streaming, ML, and graph processing**
- **Easy to use with high-level APIs (Python, SQL, Scala, Java)**
- **Integrates with cloud and modern data platforms**
- **Scales horizontally across large clusters**

🔷 Why Spark Became Popular

Spark addresses many limitations of Hadoop by providing faster execution, simpler development, and flexibility for different workloads.

Instead of writing complex MapReduce code, developers can write concise logic using Spark SQL, DataFrames, PySpark, and built-in libraries.

Spark supports real-time processing, iterative algorithms, and ML workloads, making it suitable for modern data engineering and analytics use cases.

Spark code can be written in python, java, scala , R but Python is most popular and easy to learn.

Spark + Python = PySpark