Ministry of Higher Education and Scientific Research
University of Mila
Faculty of Mathematics and Computer Science
Department of Computer Science
Master 2 I2A – Big Data
2026/2027

Chapter 2
Hadoop Systems

Presented by: Dr. Brahim Benabderrahmane

Chapter Learning Objectives

01 **Explain** the scaling limits of traditional single-server database systems and justify the Hadoop architecture based on the data locality principle.

02 **Analyze** the five sequential stages of the MapReduce processing workflow to trace key-value pair transformations across cluster nodes

03 **Compare** the MapReduce v1 JobTracker architecture against Hadoop v2 YARN.

04 **Develop** and submit distributed batch processing jobs by writing Driver, Mapper, and Reducer components using Python scripts and the Hadoop Streaming utility

05 **Evaluate** the ongoing foundational role of Hadoop as a distributed storage (HDFS) and cluster resource management (YARN) layer within modern data lakehouse and hybrid cloud architectures

Table of Contents

01 Motivation for Hadoop

02 Hadoop Ecosystem – The Big Picture

03 MapReduce Paradigm

04 Hadoop Distributed File System (HDFS)

05 Hadoop Architecture Evolution

06 Hadoop Programming Basics

07 Hadoop in Today’s Big Data World

01

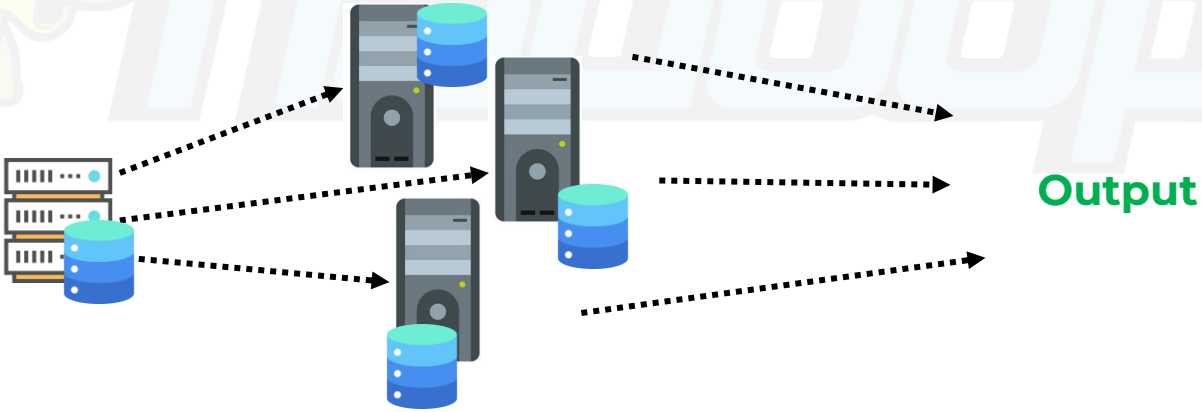
Motivation for Hadoop

Hadoop and Distributed Big Data Processing

Hadoop is an open-source framework designed for **storing and processing very large datasets**.
It runs on **clusters of ordinary (commodity) servers**, making it **scalable and cost-effective**.
Hadoop has **two main components:**

- **HDFS (Hadoop Distributed File System):** stores data by splitting files into blocks and spreading them across many machines.
- **MapReduce:** processes the data in **parallel**, close to where it is stored.

This approach allows **massive data analysis** that would be too slow or expensive on a single powerful server.



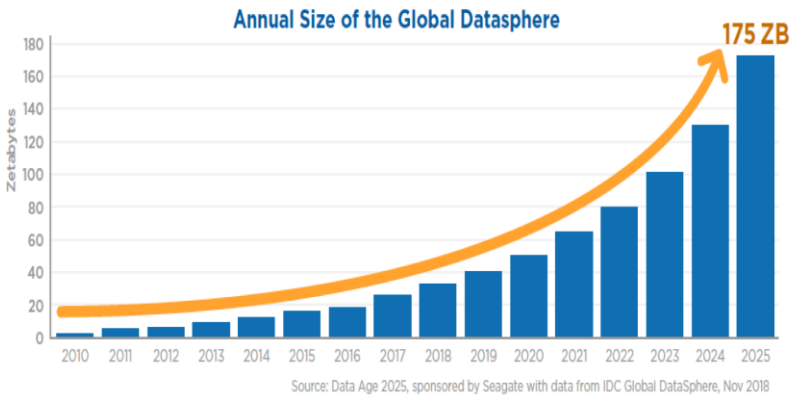
Why We Need Hadoop *(The Challenge of Big Data)*

Explosion of data volumes

- Social media, mobile devices, IoT, online transactions
- From terabytes → petabytes → exabytes

Traditional systems cannot cope

- Limited by single-server storage and CPU power
- Scaling vertically (buying bigger servers) is expensive



Need for distributed solutions

- Store data across many machines
- Process data in parallel for speed and efficiency

Other key requirements

- Fault tolerance: system must survive node failures
- Cost effectiveness: use clusters of commodity hardware

Limitations of Traditional Systems *(Single-Machine Bottlenecks)*

Storage limitations
- A single machine cannot hold today's massive datasets
- Expanding storage on one server is costly and limited

Processing power limits
- One CPU cannot process petabytes efficiently
- Tasks take hours or days to finish

Scalability problem
- Vertical scaling (buying bigger servers) is expensive and reaches physical limits

Reliability risk
- If the server fails, all data and processing stop

Cost issue
- High-end machines and proprietary storage systems are expensive

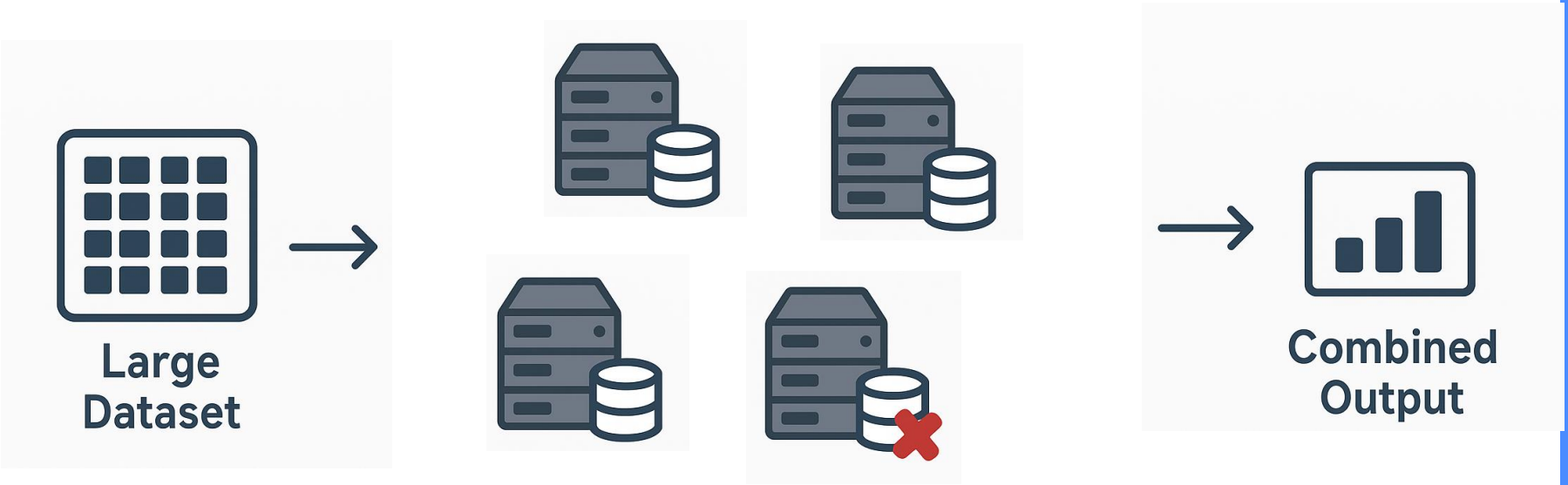
Storage full

Single point of failure

CPU overloaded

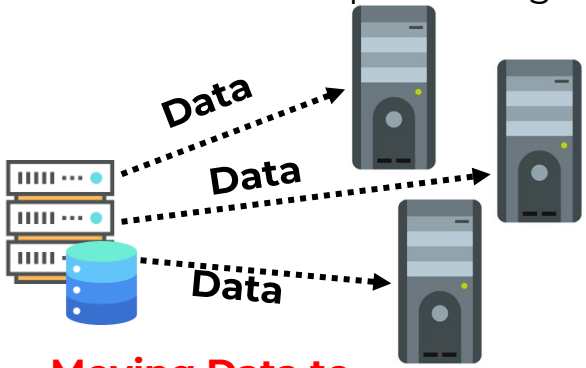
Distributed Systems Approach *(Many Machines Working Together)*

- **Divide the problem:** Break huge datasets into smaller, manageable chunks
- **Distributed storage:** Keep data pieces on different machines (nodes)
- **Parallel processing:** Each node processes its own data chunk at the same time
- **Horizontal scalability:** Add more machines to increase storage and speed
- **Fault tolerance:** Replicate data so the system continues working if a node fails
- **Cost efficiency:** Use clusters of inexpensive, commodity hardware

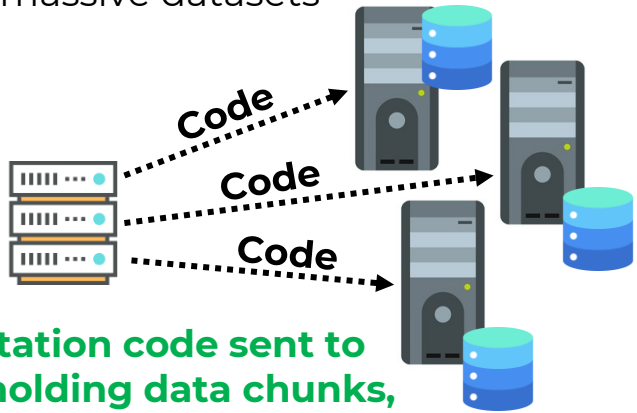


The Locality Principle - *Move Computation to Data*

- **Traditional approach:**
  Move data to the program → requires huge data transfers over the network
- **Problem:**
  Moving terabytes or petabytes across the network is slow and expensive
- **Locality principle**:
  Send the program (code) to the nodes where the data is stored
- **Efficiency**:
  Each node processes its own local data → far less network traffic
- **Result:** Faster processing and better scalability for massive datasets



Moving Data to Computational Units



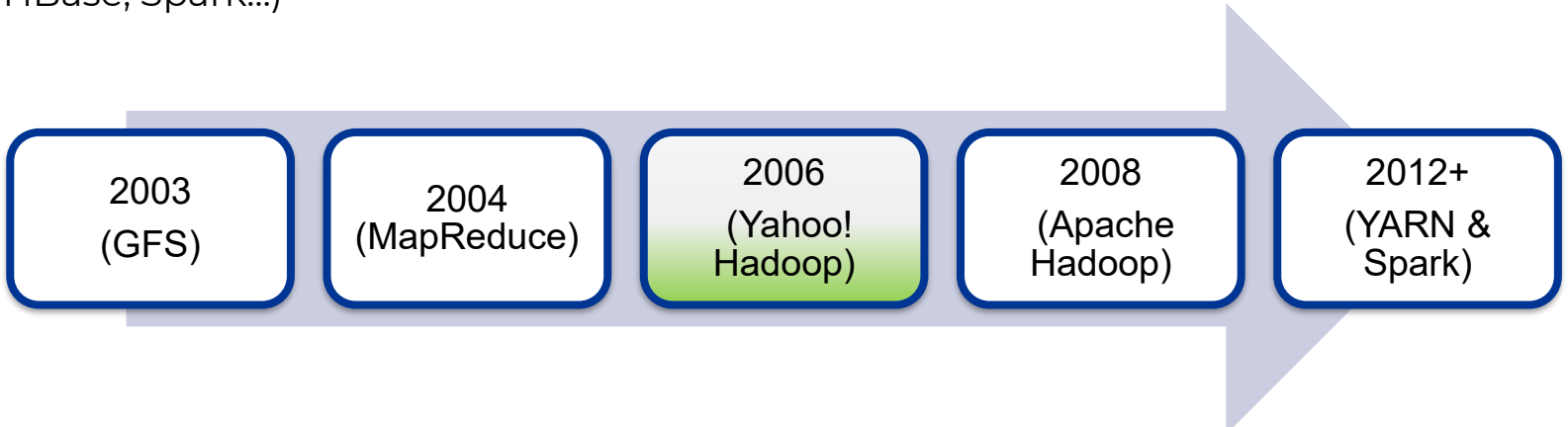
Computation code sent to nodes holding data chunks, then results combined

02

Hadoop Ecosystem
The Big Picture

Introducing Hadoop – Origins and Evolution

Origins:
Created at **Yahoo! in 2006**, inspired by Google's GFS (2003) and MapReduce (2004)
Purpose:
Handle **web-scale data** (storing & processing billions of web pages)
Open source:
Became an **Apache project in 2008** → widely adopted by industry
Core idea:
Use **clusters of commodity servers** to manage massive data
Evolution:
From a simple MapReduce engine → full **big-data ecosystem** with many tools (Hive, Pig, HBase, Spark…)



Hadoop Ecosystem – Core & Supporting Components

Core components:

- **HDFS:** Distributed storage for huge datasets
- **YARN / MapReduce:** Resource management and parallel processing

Data access & processing tools:

- **Hive:** SQL-like interface for querying data in HDFS
- **Pig:** High-level scripting for data transformations

NoSQL storage:

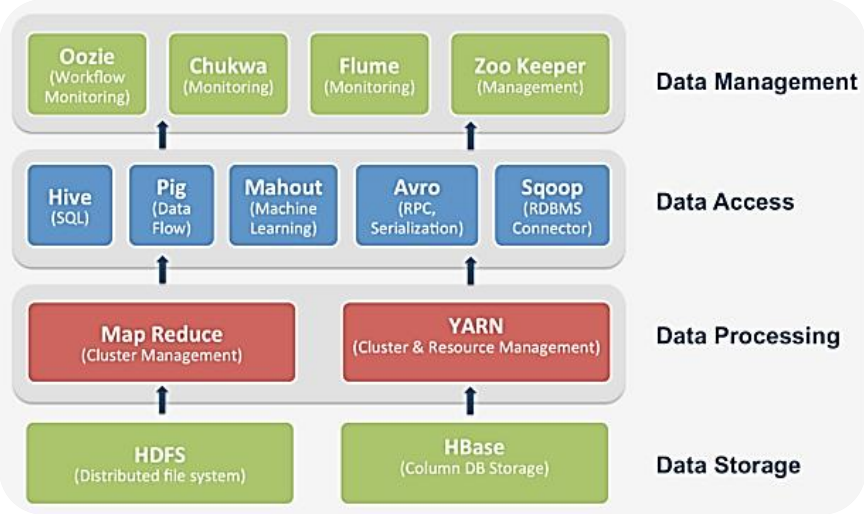
- **HBase:** Column-oriented distributed database

Coordination & scheduling:

- **Zookeeper:** Synchronization and cluster coordination
- **Oozie:** Workflow scheduling for jobs

Data ingestion:

- **Sqoop:** Transfers data between Hadoop and RDBMS
- **Flume:** Collects and streams log or event data

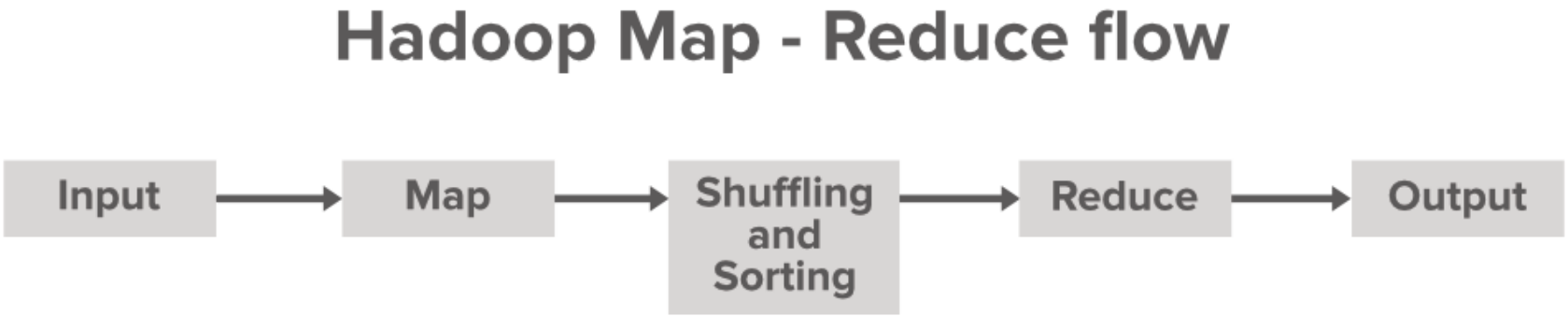


03

MapReduce Paradigm

MapReduce – Core Processing Paradigm

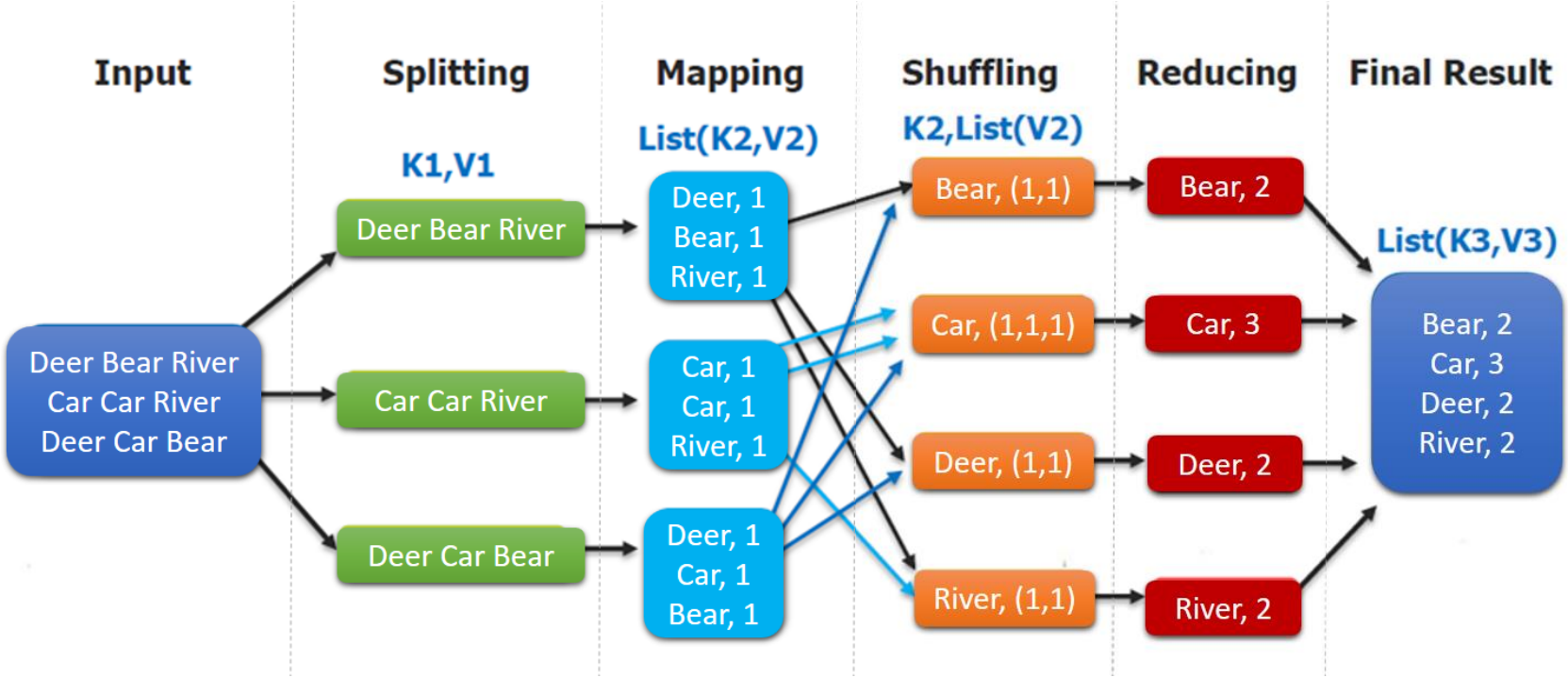
1. **Key idea:** Split large data processing into **Map** and **Reduce** phases
2. **Map phase:** Each node processes its local data and emits **(key, value)** pairs
3. **Shuffle & sort:** Intermediate results are grouped by key across the cluster
4. **Reduce phase:** Each reducer aggregates values for its assigned key and produces final results
5. **Designed for:** Large-scale **batch processing**, parallel, and fault-tolerant



MapReduce Workflow – From Input to Output

- **Hadoop splits the input file into blocks stored on different data nodes.**
- **Each node applies the mapper function to its local split and emits (key, value) pairs.**
- **Intermediate pairs are grouped by key and sent to the correct reducer.**
- **Reducers process each key's list of values to produce final aggregated results.**
- **Final data is written back into HDFS.**

Conceptual Example – Word Count



Strengths of MapReduce

Scalability:	Handles petabytes of data across thousands of commodity machines
Fault Tolerance:	If a node fails, tasks are re-run on another node automatically.
Data Locality:	Processing happens where the data is stored, reducing network bottlenecks.
Simplified Programming:	Developers only write Map and Reduce functions; Hadoop handles distribution.
Cost-Effective:	Runs on commodity hardware instead of expensive high-end servers.
Proven for Batch Processing:	Ideal for large-scale, offline analytics (e.g., log analysis, web indexing).

Limitations of MapReduce

High Latency	Each job reads from disk, processes, and writes back → slow for iterative or interactive tasks.
Batch-Only Model	Designed for **offline, one-pass batch processing**, not for real-time or streaming analytics.
Complex Programming	Requires low-level code; even simple jobs can be verbose and hard to maintain.
Inefficient for Small Jobs	Startup overhead makes it poor for **short, frequent tasks**.
Not Memory-Efficient:	Intermediate data is persisted to disk between stages, wasting time and I/O.
Hard to Optimize Pipelines	Chaining multiple MapReduce jobs leads to **complex workflows** and performance bottlenecks.

04

Hadoop Distributed File System (HDFS)

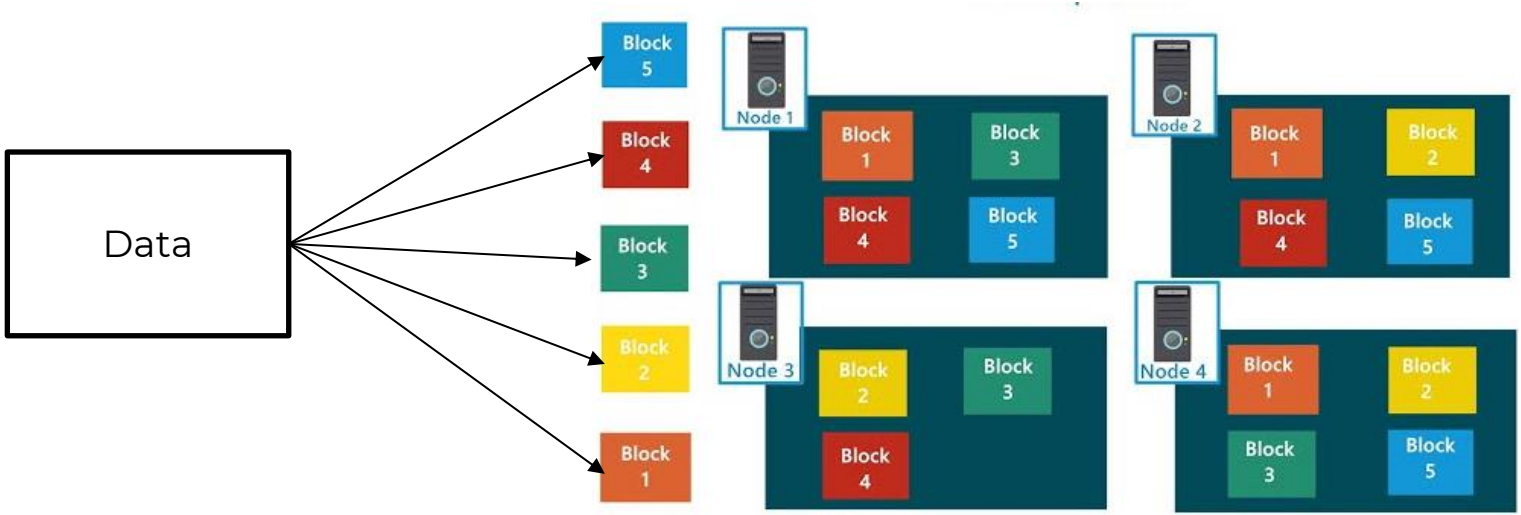
HDFS – Distributed Storage for Big Data

Challenges of Big Data

- **Huge files:** Cannot fit on one machine's disk
- **Need for reliability:** Single machine failure can lose data
- **Scalability issue:** Hard to expand storage on a single server

HDFS Solution

- **Distributed storage:** Store data across many machines in a cluster
- **Blocks:** Split files into fixed-size blocks (e.g., 128 MB)
- **Replication:** Keep multiple copies (default = 3) for fault-tolerance
- **Optimized for throughput:** Best for large sequential reads/writes, not small random updates



HDFS Architecture – NameNode & DataNodes

NameNode (Master)

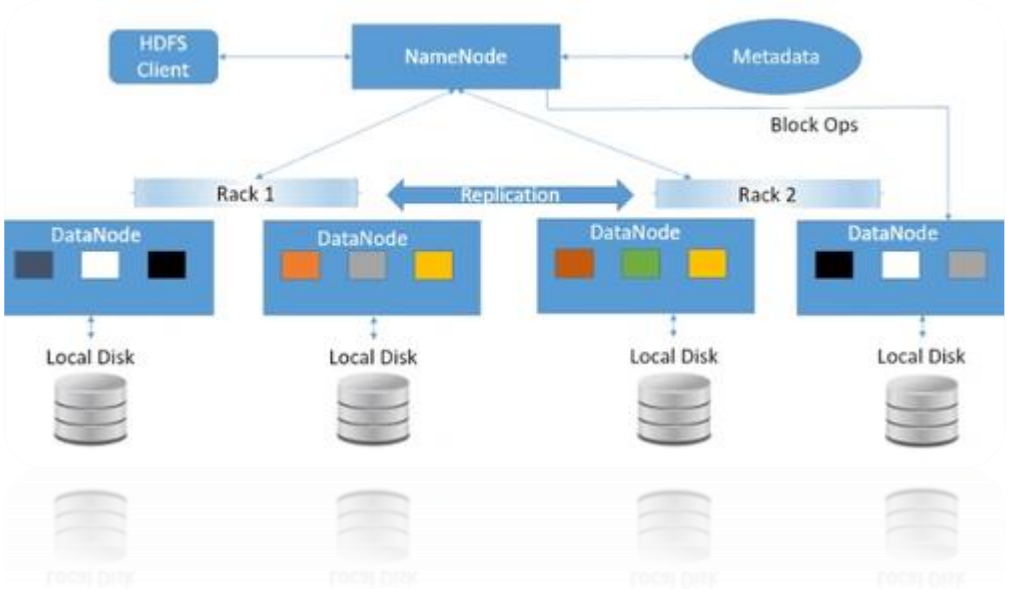
- Central **metadata manager**
- Stores file system namespace (file names, directories)
- Tracks **which DataNodes hold which blocks**
- Directs clients to the appropriate DataNodes for read/write

DataNodes (Workers)

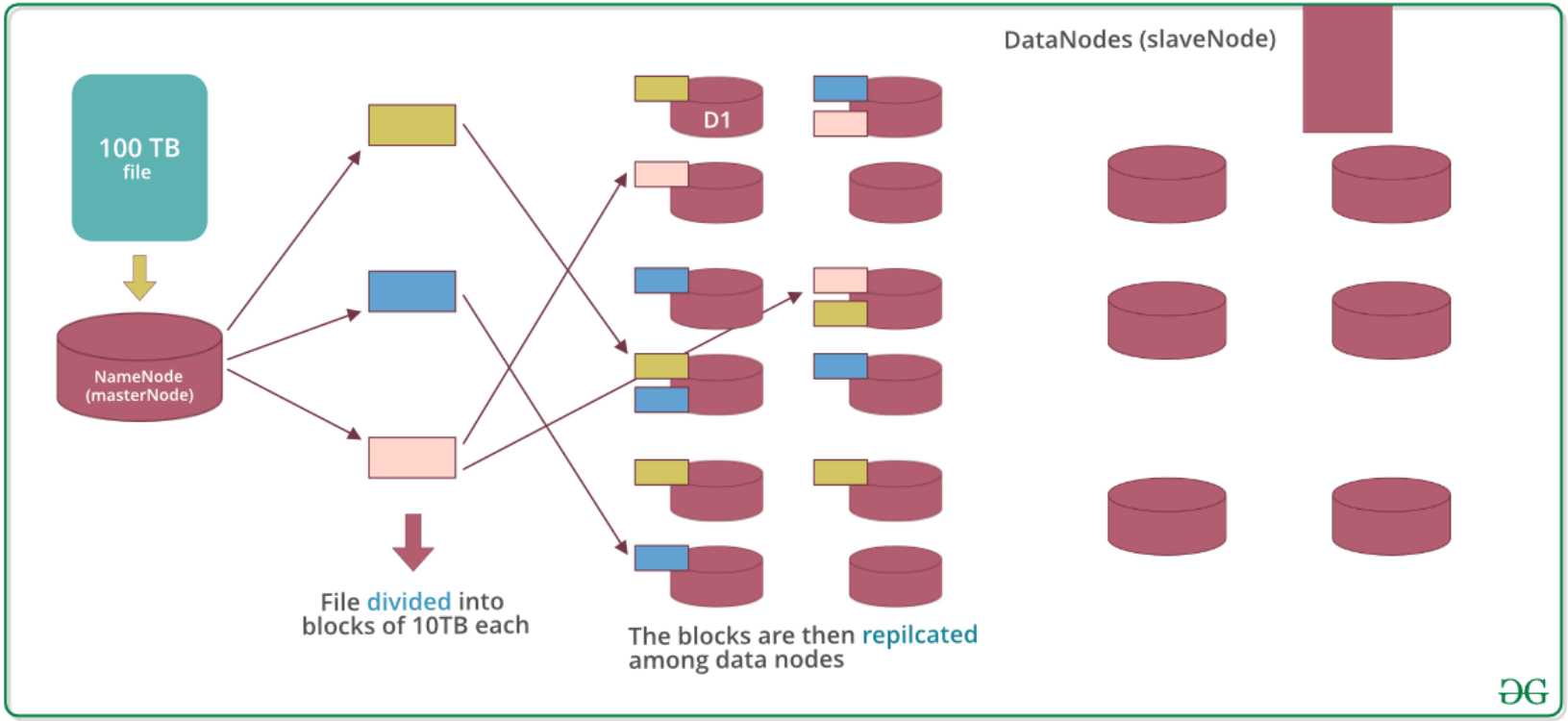
- Store the **actual data blocks**
- Perform **read/write** operations on instruction from NameNode
- Periodically send **heartbeat & block reports** to NameNode

Replication

- Each block is stored on **multiple DataNodes** (default: 3 copies)
- Provides **fault tolerance** — if one node fails, data is still available



HDFS Architecture – Data storage



05

Hadoop Architecture Evolution

Hadoop Cluster Architecture – Nodes and Roles

Cluster Basics

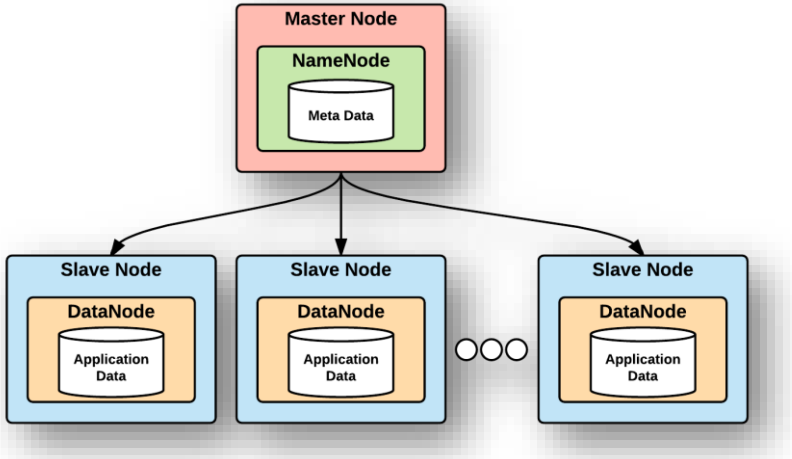
- A **cluster** = many machines working as **one system**
- Designed for **horizontal scalability** → add more nodes to increase capacity

Master Nodes:

- **NameNode** → manages **HDFS metadata** (file names, block locations)
- **ResourceManager / JobTracker** → manages **computation and resource allocation**

Worker Nodes:

- **DataNode** → stores actual **data blocks**
- **TaskTracker / NodeManager** → executes tasks (MapReduce or Spark jobs)



Hadoop v1 – MapReduce-Centric Architecture

Core Components

JobTracker (Master)
- Schedules and monitors all **MapReduce jobs**
- Assigns tasks to workers and tracks their progress

TaskTrackers (Workers)
- Run **map** and **reduce** tasks on each node
- Report status and progress back to JobTracker

Limitations of Hadoop v1
- **Single JobTracker** → bottleneck & **single point of failure**
- **Tightly coupled with MapReduce** → cannot easily run other processing frameworks
- **Limited scalability** for very large clusters (typically a few thousand nodes)

Hadoop v2 – YARN (Yet Another Resource Negotiator)

Key Idea

- **Separates resource management from computation**
- Allows many processing frameworks to share the cluster

Components:

1. ResourceManager (Master)

- Allocates **cluster resources** to applications

2. NodeManagers (Workers)

- Manage **resources on each node**
- Launch containers to run tasks

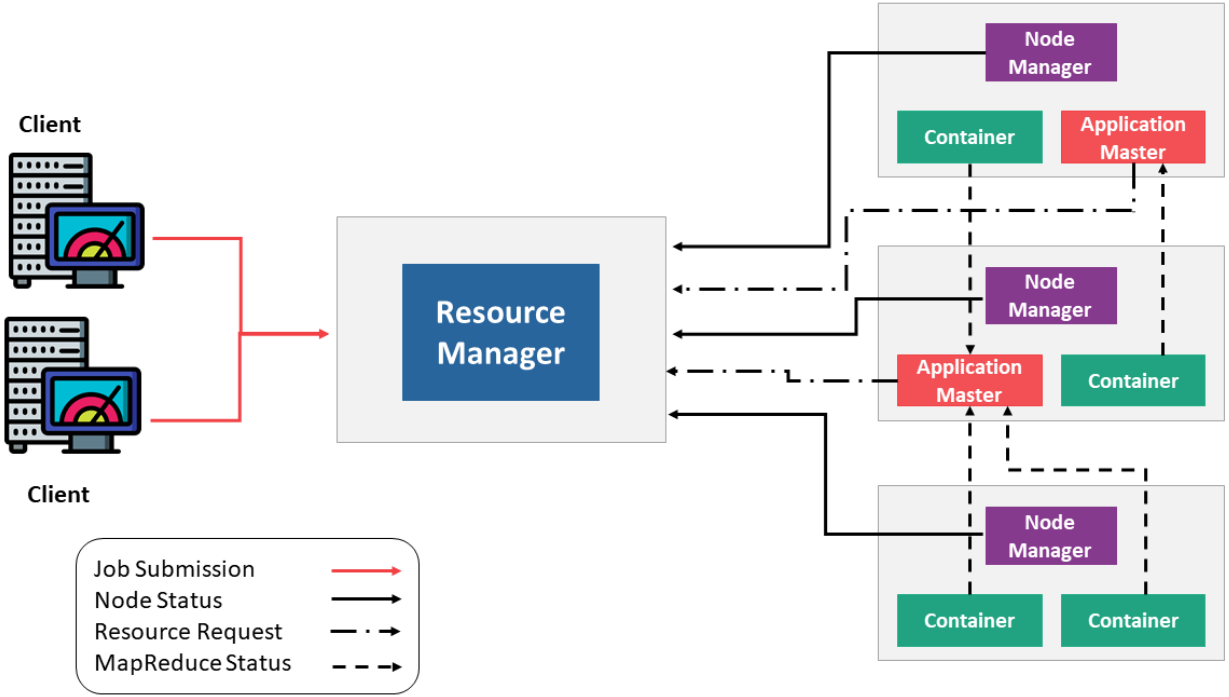
3. ApplicationMaster (per job)

- Manages the **execution of its own application**
- Requests resources from the ResourceManager

Benefits

- Runs **multiple frameworks** (MapReduce, Spark, Tez, etc.) on the same cluster
- Provides **better scalability and higher cluster utilization**
- Offers **improved fault tolerance** by avoiding a single JobTracker bottleneck

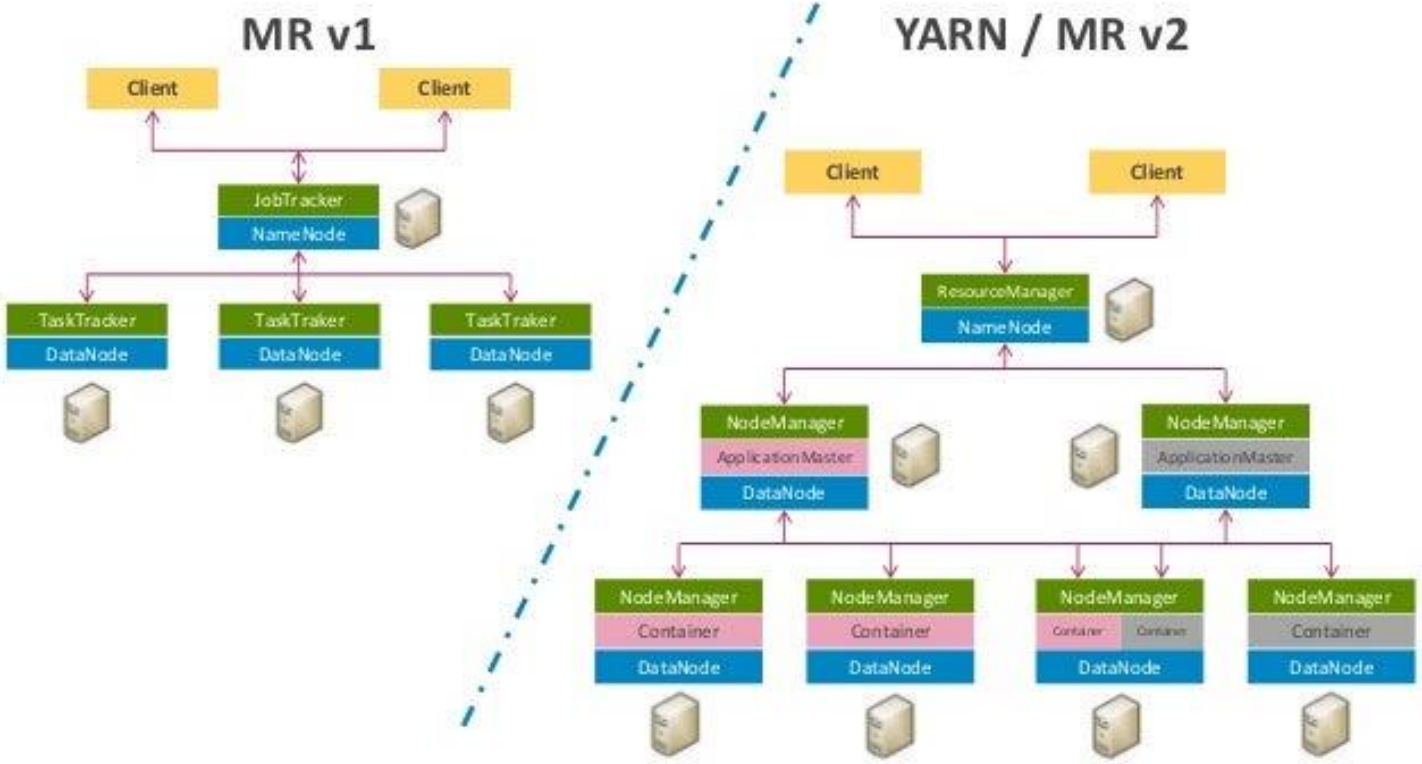
Hadoop - YARN


Client
Client
Resource Manager
Node Manager
Container
Application Master
Node Manager
Application Master
Container
Node Manager
Container
Container
Job Submission
Node Status
Resource Request
MapReduce Status

Hadoop MRv1 vs. YARN – Key Differences

Feature	Hadoop MRv1	Hadoop v2 – YARN
Resource Manager	**JobTracker** manages both resource allocation **and** job scheduling	**ResourceManager** handles resources; **ApplicationMaster** manages each job
Application Support	Supports **only MapReduce**	Supports **multiple frameworks** (MapReduce, Spark, Tez, etc.)
Scalability	Limited due to single JobTracker bottleneck	Scales to **much larger clusters** (10,000+ nodes)
Fault Tolerance	**Single point of failure** (JobTracker)	More resilient: if an ApplicationMaster fails, others continue
Flexibility	Tightly coupled with MapReduce	Flexible: can run **batch, interactive, streaming** workloads

MapReduce 1 vs YARN



- **YARN** : **Y**et **A**nother **R**esource **N**egotiator
- **MR** : MapReduce

06

Hadoop Programming Basics

Anatomy of a Hadoop Job – Driver, Mapper, Reducer

Driver Program

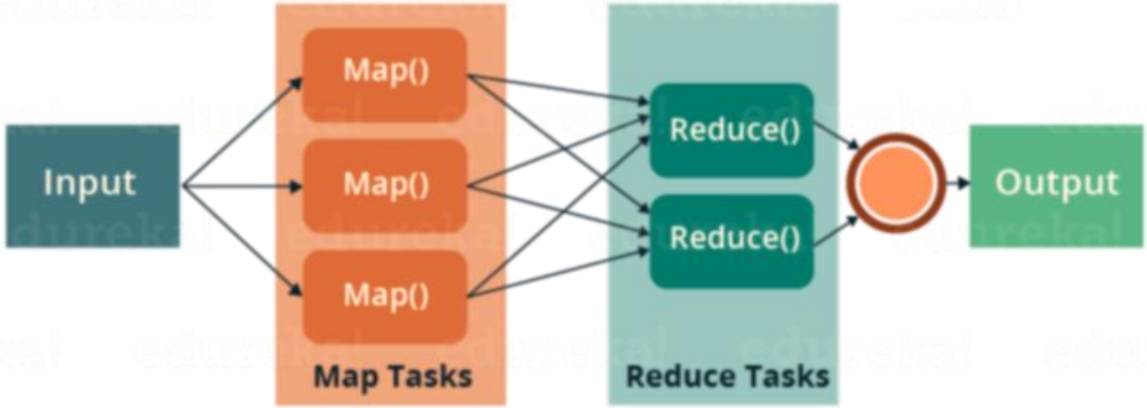
- Configures the **job** (input/output paths, mapper/reducer classes, settings)
- Submits the job to the **cluster** via YARN
- Monitors progress until completion

Mapper

- Processes **input data record by record**
- Transforms data and emits **intermediate (key, value) pairs**
- Example: (word → 1) in a word count job

Reducer

- Receives all intermediate values grouped by **key**
- **Aggregates or combines** results
- Produces **final output** (e.g., total count per word)



Hadoop Job Execution Overview

1. Job Submission

- User runs a program (Driver) that defines **Mapper**, **Reducer**, input/output paths
- The **Driver** submits the job to the **YARN ResourceManager**

2. Job Initialization

- **ResourceManager** creates an **ApplicationMaster** for the job
- The **ApplicationMaster** requests containers on **NodeManagers**

3. Task Execution

- Containers launch **Map tasks** and **Reduce tasks** on cluster nodes
- Map outputs are **shuffled and sorted**, then sent to reducers

4. Job Completion

- Reducers produce the **final output** written back to **HDFS**
- The **Driver** receives job completion status and logs results

Map Function Logic

Mapper processes each line independently.

```
import sys

for line in sys.stdin:
    for word in line.strip().split():
        print(f"{word}\t1")
```

Reduce Function Logic

```
import sys
current_word = None
current_count = 0

for line in sys.stdin:
    word, count = line.strip().split("\t")
    count = int(count)

    if current_word == word:
        current_count += count
    else:
        if current_word:
            print(f"{current_word}\t{current_count}")
        current_word = word
        current_count = count

if current_word:
    print(f"{current_word}\t{current_count}")
```

The Driver Class

Driver configures job → submits to YARN → monitors progress.

```
hadoop jar /usr/lib/hadoop-mapreduce/hadoop-streaming.jar \
  -input /data/input \
  -output /data/output \
  -mapper mapper.py \
  -reducer reducer.py \
  -file mapper.py \
  -file reducer.py
```

Hadoop Job Execution in the Cluster

What Happens When You Run a Job

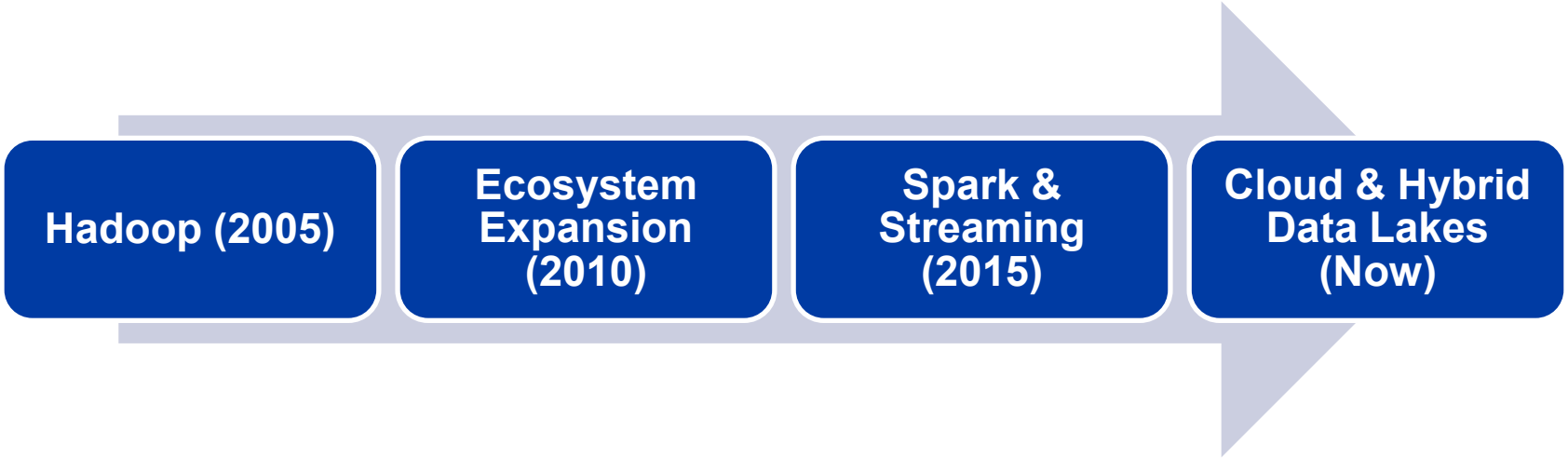
1. **Driver submits** the job to the **YARN ResourceManager**
2. **ApplicationMaster** starts and requests resources from the cluster
3. **Map tasks** are launched on **DataNodes** holding the input blocks (data locality)
4. **Shuffle & Sort** phase moves intermediate data to reducers
5. **Reduce tasks** run, results written to **HDFS output directory**
6. **ApplicationMaster** reports job completion to the Driver

07

Hadoop in Today’s Big Data World

Hadoop's Legacy and Current Role

- Hadoop revolutionized big data storage and processing in the 2010s.
- Today, **HDFS** is still used for large-scale, distributed storage.
- **YARN** continues to manage resources in many enterprise clusters.
- **MapReduce** has declined, replaced by **Apache Spark**, **Flink**, and **Presto**.
- Hadoop now serves as a **data lake foundation**, as many systems still rely on it underneath.



Hadoop vs. Modern Data Architectures

Feature	Traditional Hadoop Cluster	Modern Cloud/Hybrid Systems
Deployment	On-premise servers	Cloud-based (AWS EMR, Dataproc, HDInsight)
Storage	HDFS	Cloud storage (S3, GCS, ADLS)
Compute Engine	MapReduce, Spark-on-YARN	Serverless Spark, Flink, Beam
Scalability	Hardware scaling	Elastic scaling
Maintenance	Manual cluster management	Managed services

Hadoop in Real-World Use Today

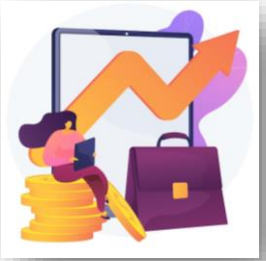
Still used by:

- **Enterprises** managing petabytes of data on-premise (banks, telecom, government).
- **Cloud-managed Hadoop clusters** (AWS EMR, Azure HDInsight, Google Dataproc).
- **Hybrid architectures** mixing HDFS + cloud object storage.

Acts as backend storage for:

- **Spark**, **Hive**, **Presto**, **Impala**, **Kafka pipelines**.

Hadoop remains strong where **data locality** and **cost control** matter

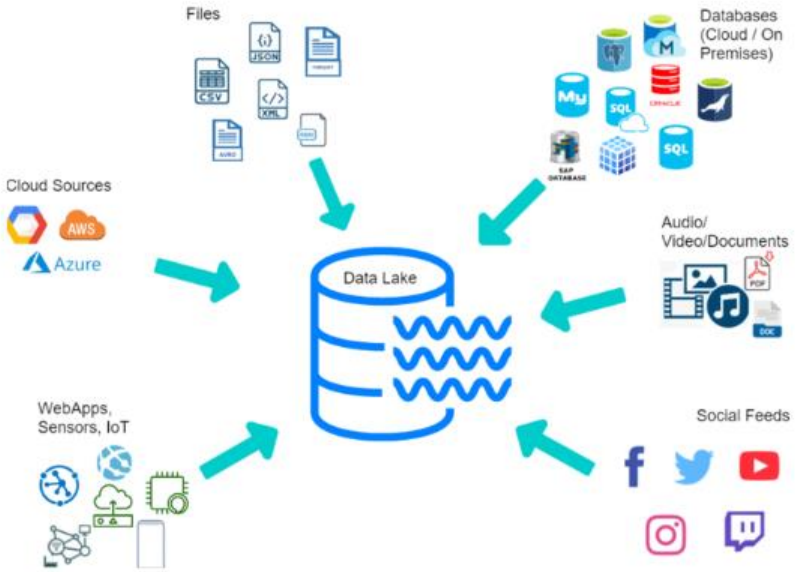


The Future of Hadoop

Shift from batch → real-time analytics.
Hadoop evolving toward:

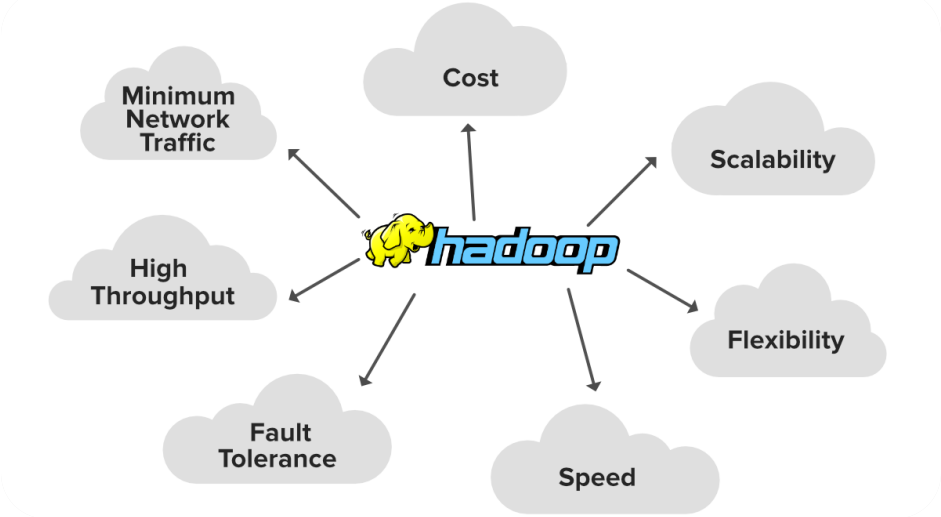
- Integration with **cloud-native** tools (Kubernetes, object storage).
- Use as part of **data lakehouse architectures** (HDFS + Delta Lake / Iceberg).

Open-source community still active (Apache Hadoop 3.x+).
Future: Hadoop as a **stable storage + resource layer**, not the “whole stack.”



Key Takeaways

- Hadoop laid the **foundation** for distributed data processing.
- **HDFS and YARN** remain critical in many infrastructures.
- The ecosystem evolved : **Spark, Flink, and cloud platforms** now lead in compute.
- Hadoop's **principles (scalability, fault tolerance, data locality)** live on in all modern systems.
- The future is **hybrid and cloud-integrated**, but Hadoop's DNA remains everywhere.



End of Chapter 2