BUSINESS INTELLIGENCE AND DATA ANALYTICS
– WE PREDICT THE FUTURE –

University of Petra

Institute of Analytics

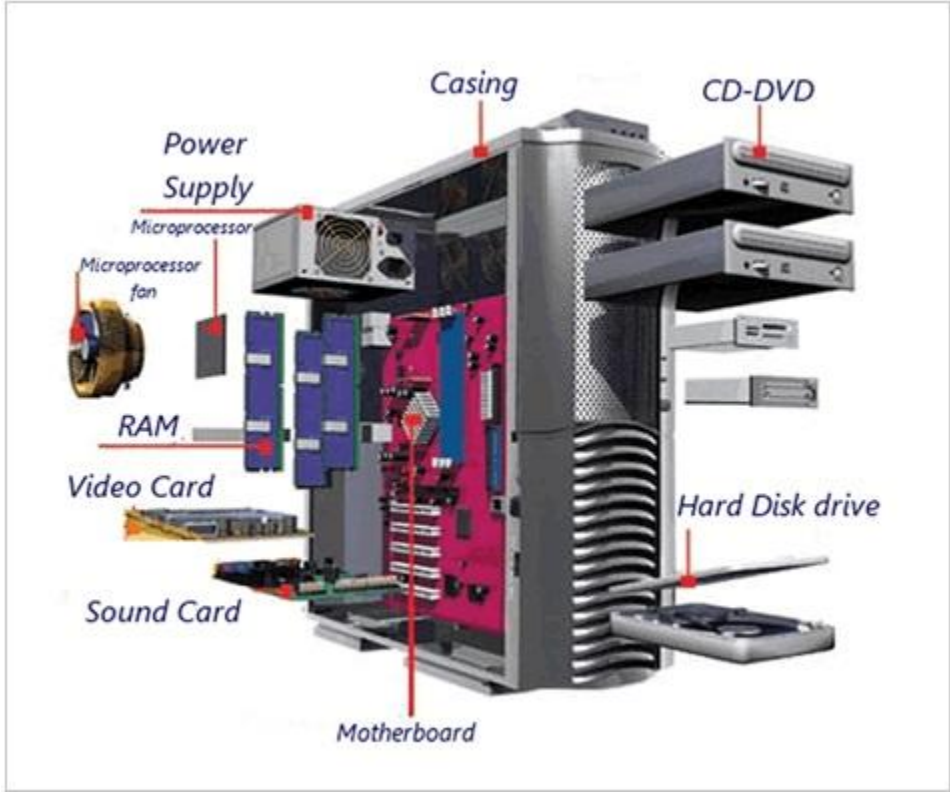
University of Petra

307401
Big Data and Data Warehouses

Introduction to Hadoop

The Limitation of a Single Machine

- A traditional OS only manages **one computer**:
  - One CPU (or a few cores)
  - One memory space
  - One storage subsystem
  - One set of devices
- **When data grows to terabytes or petabytes, or workloads need hundreds of CPUs, a single machine is not enough.**
- Scaling Up (Vertical Scaling) is limited.
- Scaling Out (Horizontal Scaling) is required.

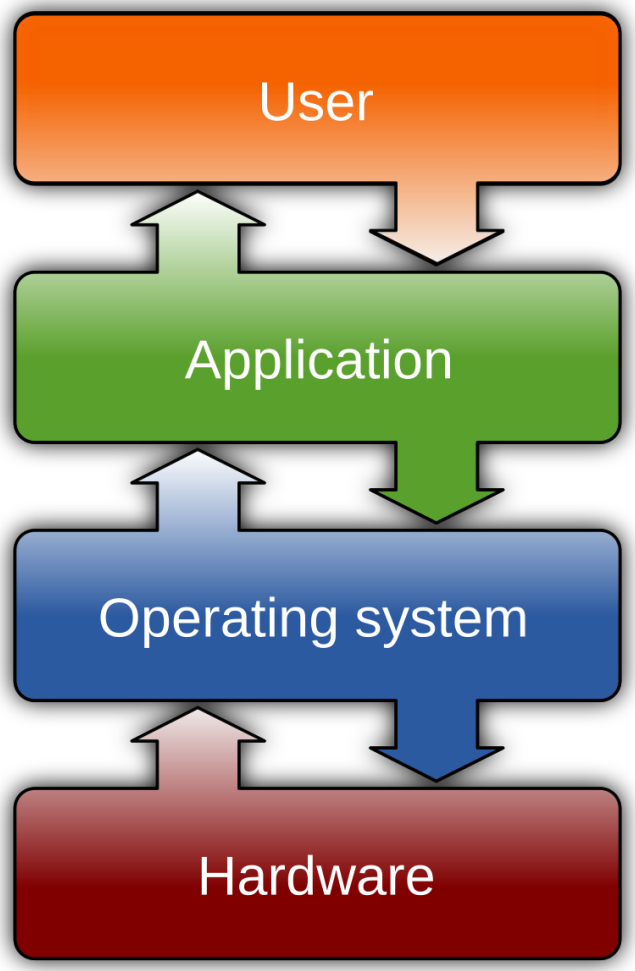


Traditional OS

An OS is the **software layer between hardware and applications**. Its core mission: **manage resources and provide abstraction** so users and programs can use the computer easily and safely.

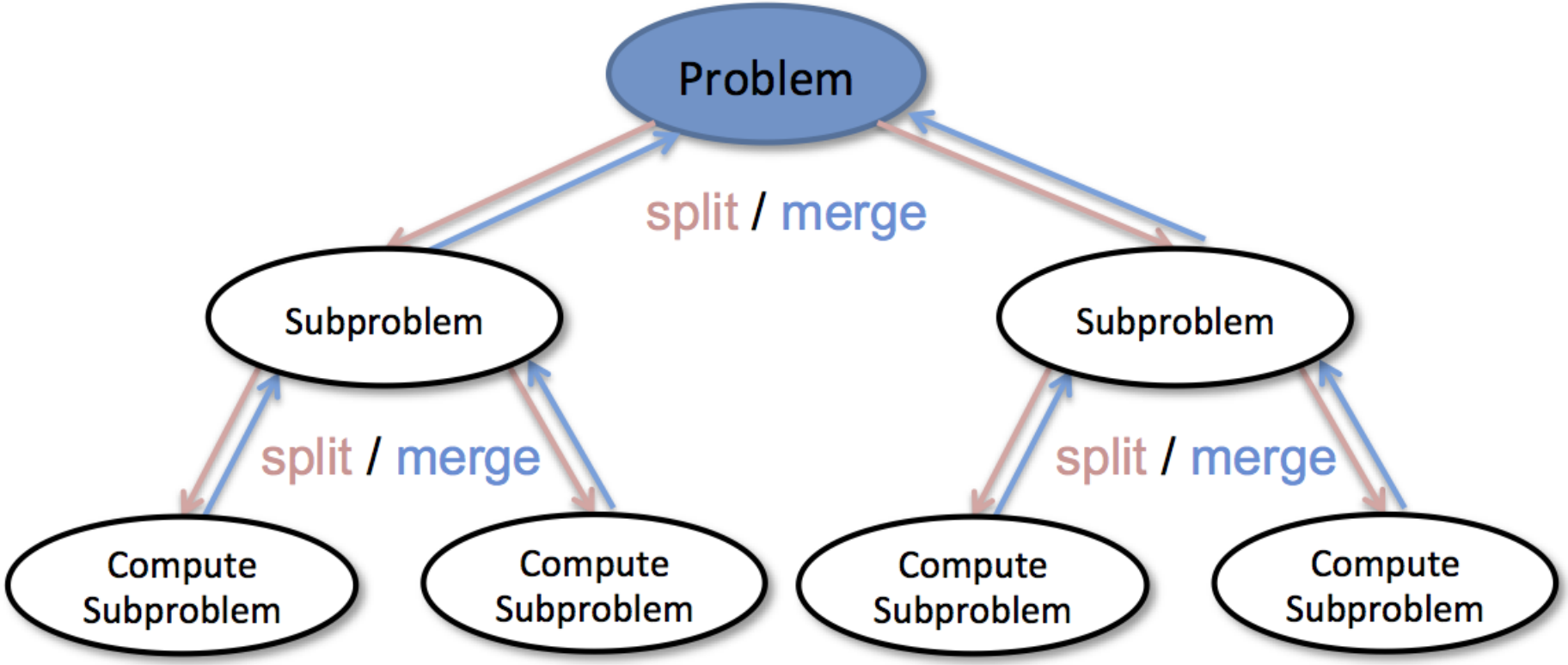
Core OS Responsibilities:

- **File System**
  - Manages how data is stored and retrieved on disk.
  - Provides a *logical abstraction* of files and directories.
  - Handles operations like create, read, write, delete.
  - Example: NTFS (Windows), ext4 (Linux).
- **Process Scheduling**
  - Decides *which program runs next* and *for how long*.
  - Shares CPU time among multiple processes (multitasking).
  - Goal: maximize performance, responsiveness, and fairness.
- **Resource Allocation**
  - Distributes hardware resources (CPU, RAM, I/O, network) among running tasks.
  - Prevents one process from starving others.
  - Handles isolation, sharing, and synchronization.
- **Fault Handling and Protection**
  - Detects errors, recovers from crashes, isolates failing programs.
  - Protects user data and maintains system stability.
- **Abstraction**
  - Hides the messy details of hardware (disk blocks, CPU cycles) behind *simple APIs* like “open a file” or “run a process.”



Solution for Big Data Problems

Divide and Conquer



Types of Processing in Computing

Processing Type	Description	Example
Sequential Processing	Tasks are executed one after another in a linear sequence.	Reading and executing lines of code in a single-threaded program.
Multi-Tasking	Multiple tasks share the same CPU by switching between them rapidly.	Running multiple applications (browser, music player, and document editor) on a computer.
Parallel Processing	Multiple processors or cores execute different parts of a task simultaneously.	Image processing using multiple CPU cores.
Distributed Processing	Tasks are divided across multiple computers working together over a network.	Cloud computing and blockchain networks.

Understanding Speedup in Parallel Computing

What is Speedup?

- A performance metric that compares how fast a task runs using multiple processors vs. a single one.
- It answers the question: *"How much faster is the program when run in parallel?"*

Speedup Formula

$$Speedup(S) = \frac{T_1}{T_p}$$

- T_1: Time taken by a single processor
- T_p: Time taken by p processors

Types of Speedup

- **Linear Speedup**
  - Ideal case: each processor contributes equally.
  - *Example*: 4 processors reduce a 40s task to 10s → $S = 40/10 = 4$
- **Superlinear Speedup**
  - Occasionally, $S > p$ due to improved memory usage or cache efficiency.
  - *Example*: Task finishes in 9s with 4 processors → $S = 40/9 \approx 4.44$
- **Sublinear Speedup**
  - Occurs when communication or synchronization slows down performance.
  - *Example*: Task takes 12s with 4 processors → $S = 40/12 \approx 3.33$

Amdahl's Law: The Limits of Parallel Speedup

Formula:

$$S = \frac{1}{(1-P) + \frac{P}{p}}$$

Where:

- S: Overall speedup of the program
- P: Fraction of the program that can be parallelized (parallel portion)
- p: Number of processors

Core Idea:

Amdahl's Law shows that the speedup of a program is limited by the portion that must be executed sequentially, regardless of how many processors are used.

Why Some Parts Can't Be Parallelized:

- **Initialization:** Data setup must occur before tasks are distributed.
- **Data Dependencies:** Certain tasks depend on the results of previous ones.
- **Input/Output:** Reading/writing operations are often inherently sequential.
- **Critical Sections:** Only one processor can safely execute some operations (e.g., updating shared data).

Impact of Sequential Limits:

Even with an unlimited number of processors, the speedup is ultimately constrained by the sequential portion. If 10% of the task is sequential, the theoretical maximum speedup is:

$$S_{\max} = \frac{1}{1-P} = \frac{1}{0.1} = 10$$

Example:

If 75% of a task is parallelizable ($P = 0.75$) and we use 4 processors ($p = 4$):

$$S = \frac{1}{(1-0.75) + \frac{0.75}{4}} = \frac{1}{0.25 + 0.1875} = \frac{1}{0.4375} \approx 2.29$$

This shows limited speedup due to the 25% sequential portion.

Challenges in Distributed Resource Management (Addressed by YARN Next)

Challenge / Question	What it relates to in YARN
How do we assign work units to workers?	The ResourceManager and ApplicationMaster handle resource allocation and task assignment.
What if we have more work units than workers?	Scheduling and queuing policies (capacity, fairness, FIFO).
What if some workers are slower than others?	Speculative execution and dynamic scheduling handle stragglers.
What if workers need to share partial results?	HDFS or shared storage, not directly YARN's job, but coordinated via YARN-managed containers.
How do we aggregate partial results?	The ApplicationMaster (e.g., in MapReduce, the JobTracker) coordinates result collection.
How do we know all the workers have finished?	The ApplicationMaster tracks container/task status via NodeManagers.
What if workers die?	Failure detection and recovery — NodeManagers report heartbeats to ResourceManager; tasks are rescheduled.

Introduction to Hadoop

- Hadoop is an open-source framework overseen by Apache Software Foundation which is written in Java for storing and processing of huge datasets with the cluster of commodity hardware.
- There are mainly two problems with the big data that Hadoop addresses:
  1. The need to store huge amount of data.
  2. The need to process that stored data.
- The traditional approach like RDBMS is not sufficient due to the 5-Vs related to data.
- Hadoop serve as the solution to the problem of big data i.e. storing and processing big data.

The Origins of Big Data Processing Frameworks

In the early 2000s, Google faced **massive data-processing challenges** that traditional file systems couldn't handle. They were indexing the entire web — billions of pages — and needed to store and process enormous amounts of data efficiently and reliably.

The core problems:

- **Huge datasets** — petabytes of data spread across thousands of inexpensive (and failure-prone) machines.
- **Frequent hardware failures** — commodity servers would fail regularly; the system had to tolerate this automatically.
- **High throughput, not low latency** — Google's batch workloads (like indexing) needed fast data streaming and parallel reads/writes.
- **Large sequential reads/writes** — most jobs involved processing large files, not small random reads.
- **Append-heavy workloads** — most files were written once and then read many times.

So, Google built GFS (published in 2003 by Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung):

- A **distributed file system** for large-scale, fault-tolerant data storage.
- Files are split into **64 MB chunks**, replicated (default 3 copies), and distributed across servers.
- A single **master node** keeps metadata (namespace, chunk locations).
- Clients read/write directly from **chunk servers**, not through the master — reducing bottlenecks.
- Designed for **batch processing**, not interactive queries.
- GFS was the foundation for storing and feeding data to Google's large-scale processing systems.

The Birth of Hadoop - Doug Cutting & Mike Cafarella Work

2002 – Nutch begins

- Doug Cutting and Mike Cafarella start the **Nutch** project, an open-source web search engine.
- Goal: build a free, large-scale crawler and search system like Google.
- This is **before** Google's GFS and MapReduce papers.

2003 – Google publishes the GFS paper

- Google releases its paper on the **Google File System (GFS)**.
- Doug reads it and realizes Nutch could use similar distributed storage ideas.

2004 – Google publishes the MapReduce paper

- Google describes its **MapReduce** programming model for distributed computation.
- Doug and Mike see how these two ideas (GFS + MapReduce) could solve Nutch's scalability limits.

Doug Cutting

Mike Cafarella

The Birth of Hadoop - Doug Cutting & Mike Cafarella Work

2004–2005 – Nutch adopts Google's ideas

- Doug and Mike implement their own versions:
  - **Nutch Distributed File System (NDFS)** – inspired by GFS.
  - **MapReduce implementation** – inspired by Google's paper.
- These additions allow Nutch to handle very large web crawls and computations.

2005–2006 – Hadoop is born

- The storage and processing layers (NDFS + MapReduce) are separated from Nutch to form a new project: **Hadoop**.
- Around this time, **Yahoo hires Doug Cutting** to work full-time on Hadoop and scale it up for real production use.

2008 – Hadoop becomes an Apache project

- **Apache Hadoop** is officially established as a top-level open-source project.
- It becomes the foundation of the modern **Big Data ecosystem**.

In Brief - باختصار

- Google **invented the ideas** behind Big Data — the architecture, the theory, the papers.
- But **Hadoop implemented and open-sourced** those ideas — making Big Data available to everyone else.

So:

- **Google** = the *research and internal innovation phase*.
- **Hadoop** = the *open-source and industrial revolution phase*.

Concept	Google's Work	Doug Cutting's Contribution
Distributed Storage	GFS (Google File System)	HDFS (Hadoop Distributed File System)
Distributed Processing	MapReduce (Google's model)	Hadoop MapReduce
Purpose	Power Google Search and data analytics	Open-source big data framework for everyone
Outcome	Internal system at Google	Sparked the entire Big Data ecosystem (Hadoop, Hive, Spark, etc.)

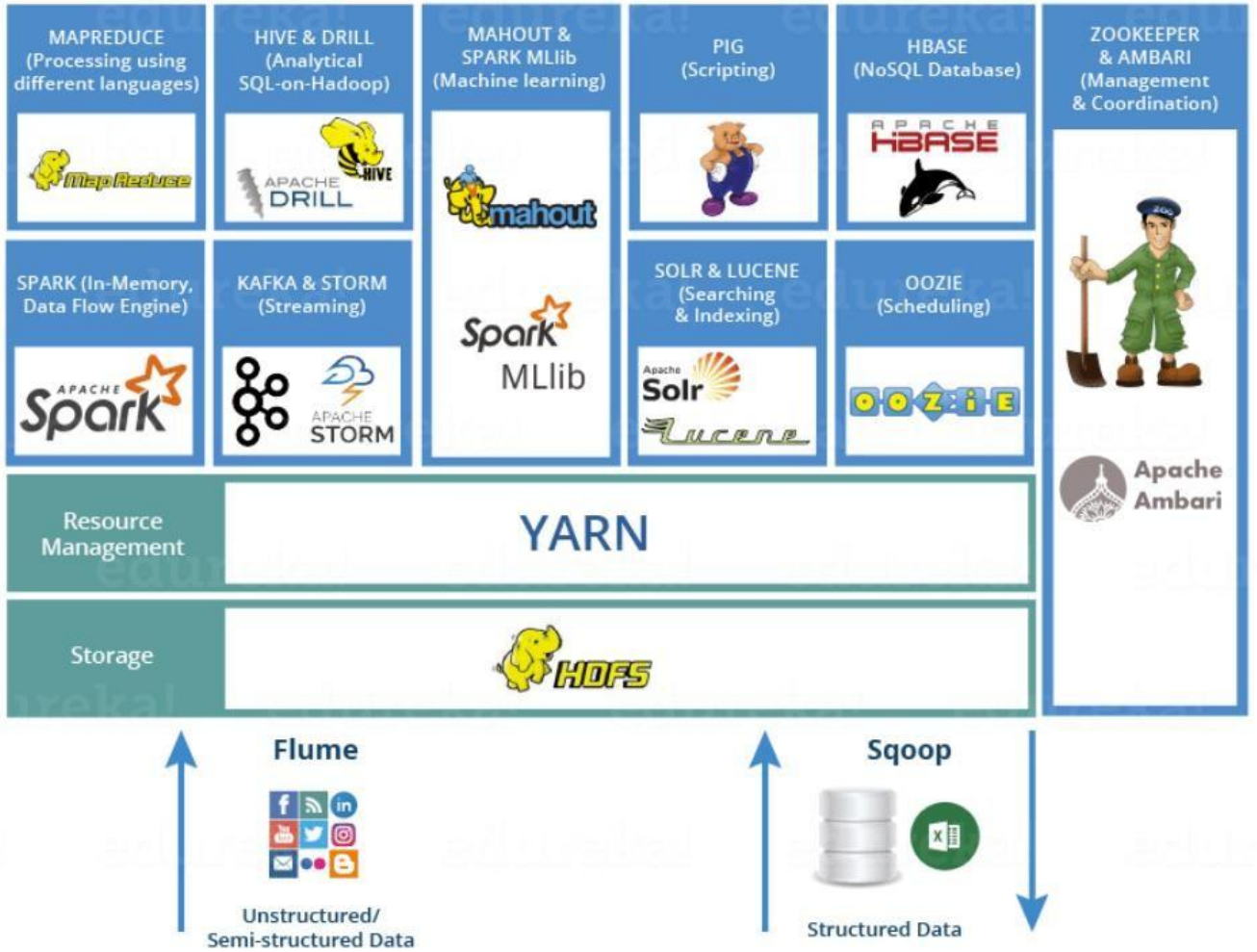
Hadoop Echo System

Hadoop ecosystem is a suite of tools and frameworks that enables Hadoop to handle large amounts of data.

Basically, Hadoop has the following three components:

1. HDFS (Hadoop Distributed File System): A distributed file system that provides high-throughput access to application data.
2. MapReduce: A framework for parallel processing of large data sets.
3. YARN, Resource Manager.

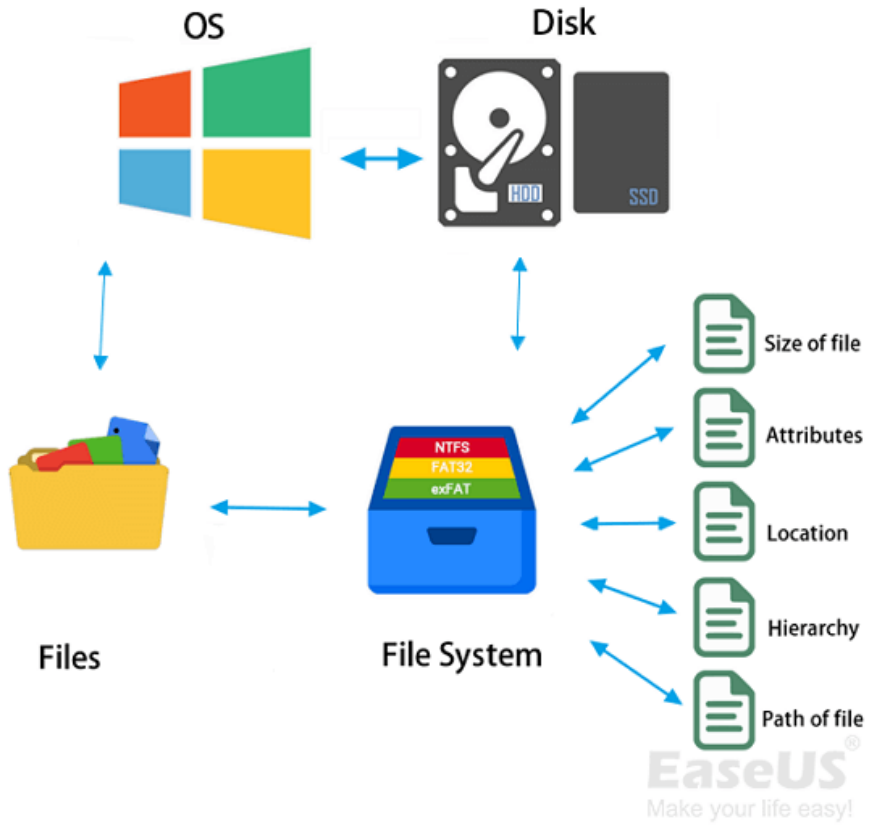
Other ecosystem components: Hive, Pig, HBase, Storm, Spark, Kafka, Flume, Sqoop, Oozie, Zookeeper…etc. These components **work together** to provide a comprehensive solution for managing and processing large data sets in a distributed computing environment.



HDFS – Hadoop Distributed File System

What is HDFS?

- **Hadoop Distributed File System (HDFS)** is the storage layer in Hadoop.
- It enables **distributed storage** and **fault tolerance** by replicating data across multiple nodes.
- A **file system** is the method and structure that an operating system uses to **store, organize, and manage data** on storage devices like hard drives, SSDs, or USB drives.
- It defines **how files are named, stored, and accessed**. The file system keeps track of where each file's data is physically located on the disk and manages metadata such as file size, permissions, and timestamps.
- Common examples include **NTFS** (Windows), **ext4** (Linux), and **APFS** (macOS).
- In distributed systems like Hadoop, a **distributed file system** (such as HDFS) spreads data across many machines, allowing large-scale storage and parallel processing.



HDFS – Hadoop Distributed File System

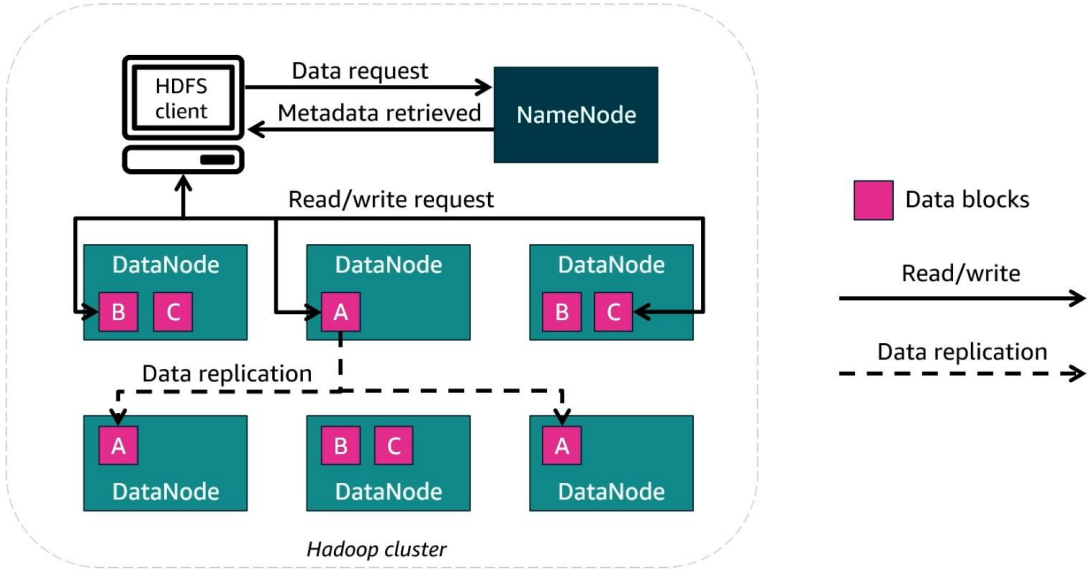
Key Components of HDFS:

1. NameNode

- Acts as the **master node**.
- Manages **metadata** (file locations, permissions, and structure).
- Does not store actual data.

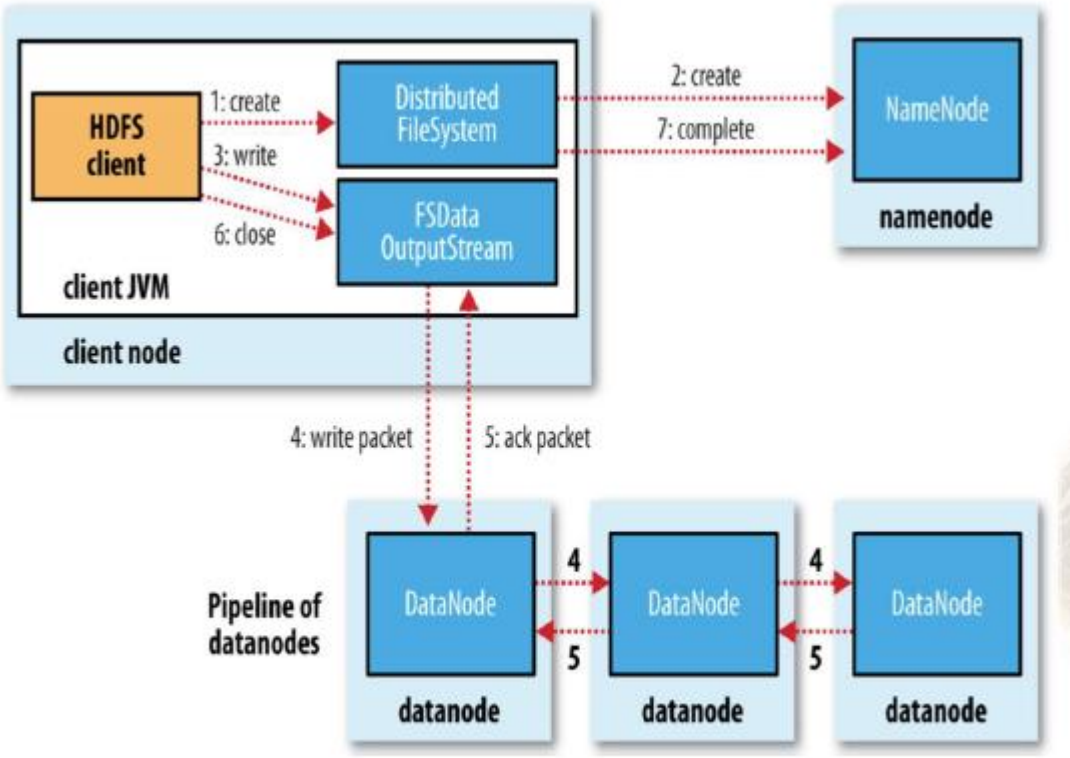
2. DataNodes

- Store actual data in **blocks**.
- Handle read/write requests from clients.
- Perform **data replication** to ensure fault tolerance.



Writing File to HDFS

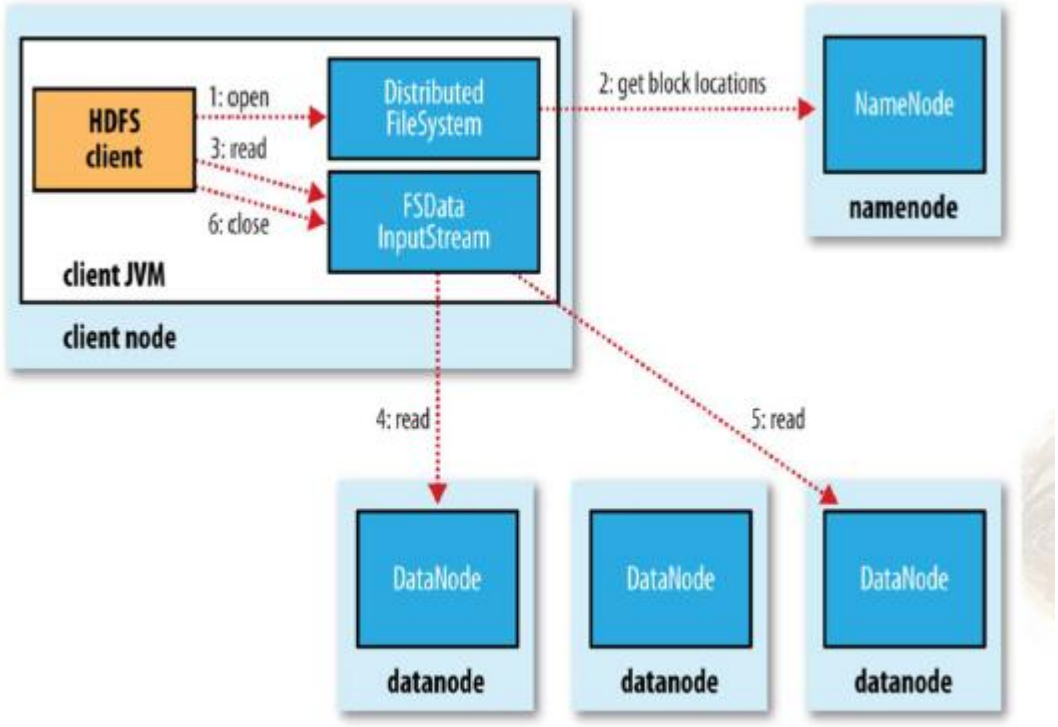
1. **Client Application Initiates Upload:** A client application (such as a command-line tool or custom software using the HDFS API) initiates a file upload by contacting the **NameNode**.
2. **NameNode Responds with Block Allocation:** The NameNode calculates how many blocks will be needed based on the file size and the configured block size (e.g., 128 MB) and then assigns which DataNodes should store each block and its replicas.
3. **Client Writes Data Blocks to DataNodes:** The client begins writing each block to the first assigned DataNode. That DataNode then forwards the block to the second, and then the third (in a pipeline fashion) to meet the replication requirement.
4. **Data Replication Happens Automatically:** Replicas of each block are written to multiple DataNodes, typically three, to ensure fault tolerance.
5. **DataNodes Send Block Reports:** After storing the blocks, the DataNodes send reports back to the NameNode, confirming the block storage and replication status.
6. **Upload Completion:** Once all blocks are successfully written and acknowledged, the client receives confirmation. The file is now stored and available in HDFS.



A client writing data to HDFS (From Hadoop Definite Guide)

Reading File from HDFS

1. **Client Application Sends Read Request to NameNode**
   A client application (e.g., a user command or a program using the HDFS API) requests to read a file by contacting the NameNode.
2. **NameNode Returns Block Metadata**
   The NameNode checks its metadata and returns the list of block locations for the requested file, including the addresses of the DataNodes that store each block.
3. **Client Reads Data Blocks from DataNodes**
   The client directly contacts the appropriate DataNodes to read the actual data blocks. It can choose the closest replica to reduce latency.
4. **Client Reassembles the File**
   The client assembles the blocks in the correct order to reconstruct the full file in memory or on local storage.



A client reading data to HDFS (From Hadoop Definite Guide)

YARN, Yet Another Resource Negotiator

What is YARN?

- YARN (Yet Another Resource Negotiator) is the resource management layer in Hadoop.
- It efficiently allocates resources and schedules tasks for processing big data jobs.

Key Components of YARN

1. Resource Manager (RM) - Central component managing resource allocation.

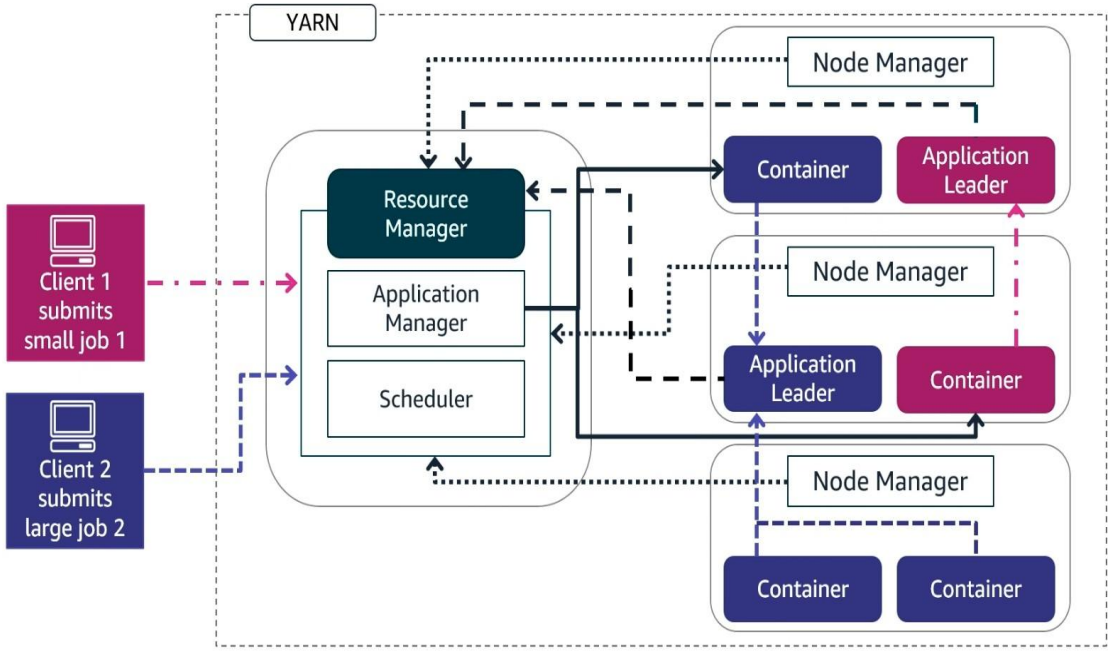
- It has two main parts:
- **Application Manager** – Manages job submission and execution.
- **Scheduler** – Allocates resources to tasks based on policies.

2. Node Managers (NM)

- Runs on each DataNode to execute tasks.
- Manages **Containers**, which are allocated resources for executing tasks.

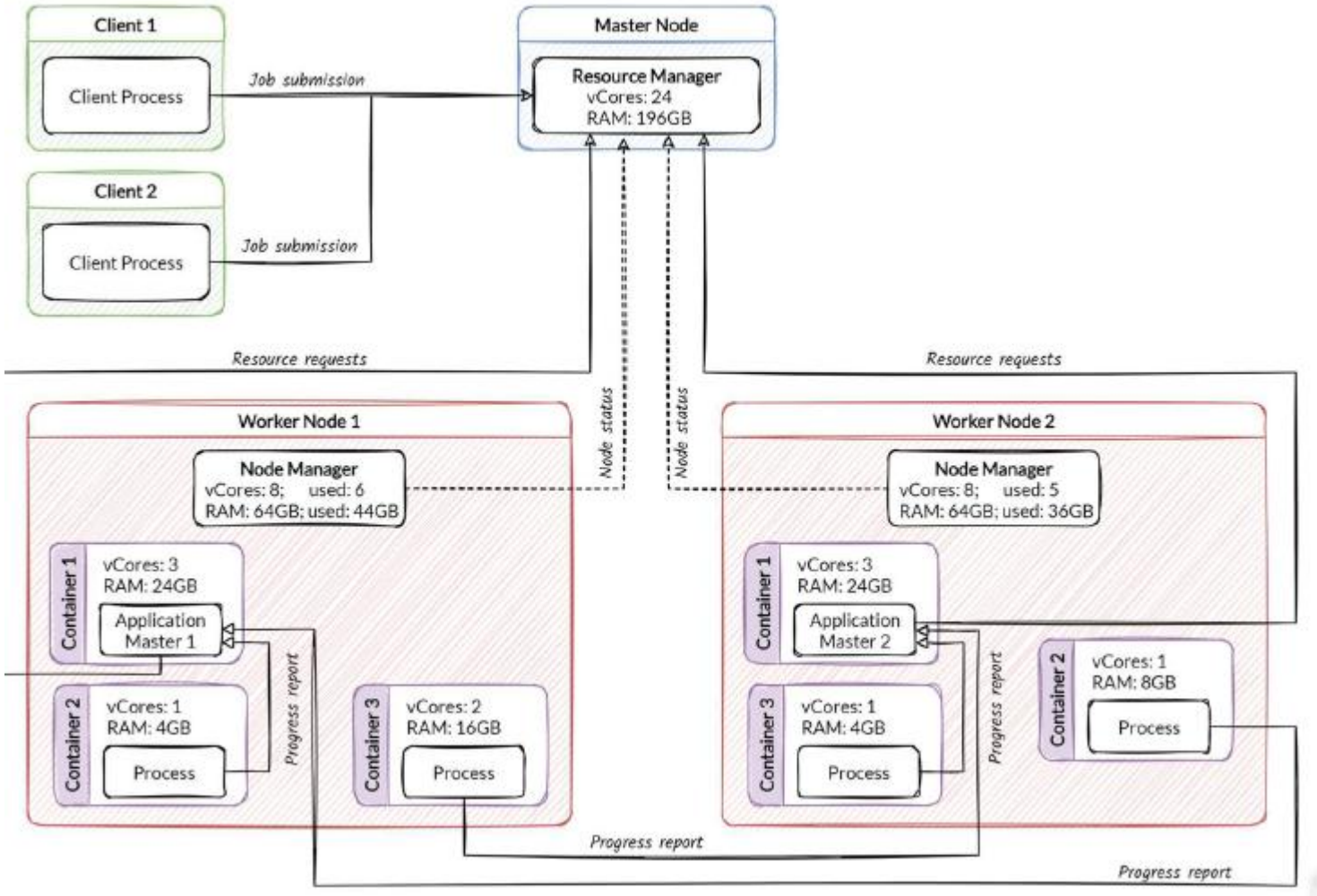
3. Application Master (AM)

- Runs within a container and coordinates job execution.
- Communicates with the Resource Manager and Node Managers.



Resource Allocation in YARN

1. **A client application** submits a job (application) to the **Resource Manager (RM)**. The job includes details like the number of required resources and the type of task.
2. **The Application Manager** component of the RM registers the new application and keeps track of its status.
3. **The Scheduler** (within the RM) evaluates current cluster conditions and assigns a container on a suitable **Node Manager (NM)** to launch the **Application Master (AM)**.



Resource Allocation in YARN

4. **The Application Master** running inside the allocated container is initialized. It is responsible for managing the entire job lifecycle, including requesting additional resources for task execution.
5. **Based on the job's needs, the Application Master requests more containers from the Resource Manager** to execute individual tasks (e.g., map or reduce tasks).
6. **The Scheduler** again evaluates availability and policy constraints and allocates containers on various **Node Managers**.
7. **Each Node Manager** launches containers and runs the assigned tasks (as directed by the Application Master). Tasks process data and perform computations.
8. **The Application Master** monitors task completion, handles retries if needed, and reports status back to the Resource Manager.
9. **Once all tasks finish, the final output is either returned to the client** or stored in HDFS, and the Application Master deregisters from the RM.

MapReduce: Distributed Big Data Processing Framework

MapReduce is a programming model and framework for processing massive datas clusters using parallel computation.

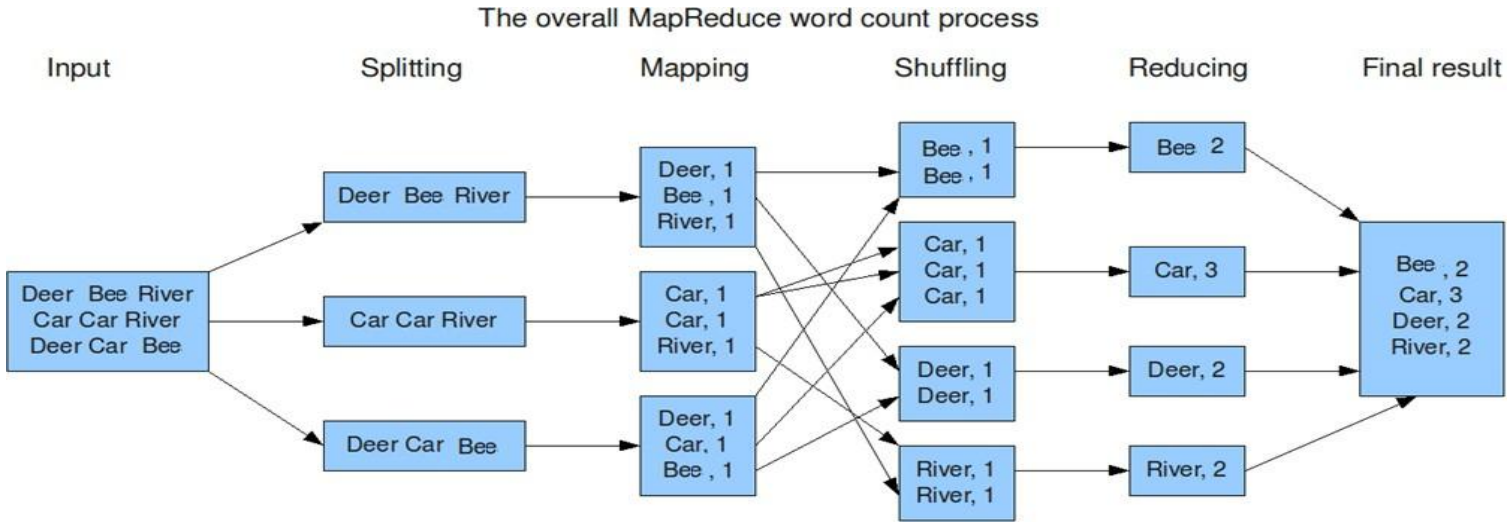
Key Components:

1. **Mapper**: Processes input data chunks and transforms them into intermediate key-value pairs
2. **Reducer**: Aggregates and combines intermediate data by key to produce final results
3. **YARN ApplicationMaster**: Coordinates job execution and resource management
4. **HDFS Integration**: Provides distributed storage for input/output data

MapReduce: Distributed Big Data Processing Framework

Technical Workflow:

- **Input Splitting**: Data divided into blocks (typically 128MB) across HDFS nodes
- **Map Phase**: Parallel Mapper tasks process splits → intermediate key-value pairs
- **Shuffle & Partition**: Framework redistributes data by key across cluster nodes
- **Sort & Group**: Intermediate data sorted and grouped by key for each Reducer
- **Reduce Phase**: Reducer tasks aggregate values per key → final output to HDFS



MapReduce: Distributed Big Data Processing Framework

Advantages:

- **Fault Tolerance**: Automatic task retry and data replication
- **Scalability**: Handles petabyte-scale datasets across thousands of nodes
- **Framework Integration**: Works with Hadoop ecosystem (Hive, Pig, Spark)

Real-World Applications:

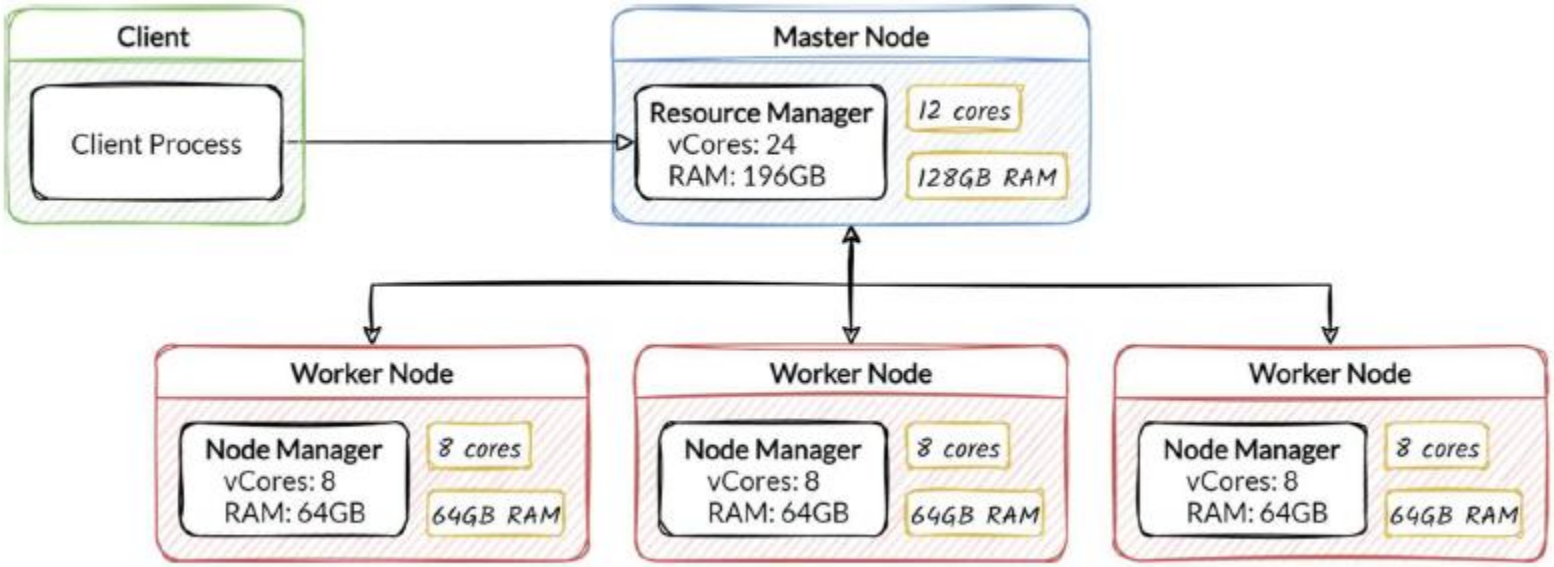
- Log analysis for fraud detection (Facebook, Google)
- Search index generation and web crawling
- Financial risk analysis and recommendation systems

Modern Context: While Spark has largely replaced MapReduce for many use cases, MapReduce remains fundamental for batch processing in enterprise Hadoop environments.

Map Reduce Scenario

1.Client Submission:

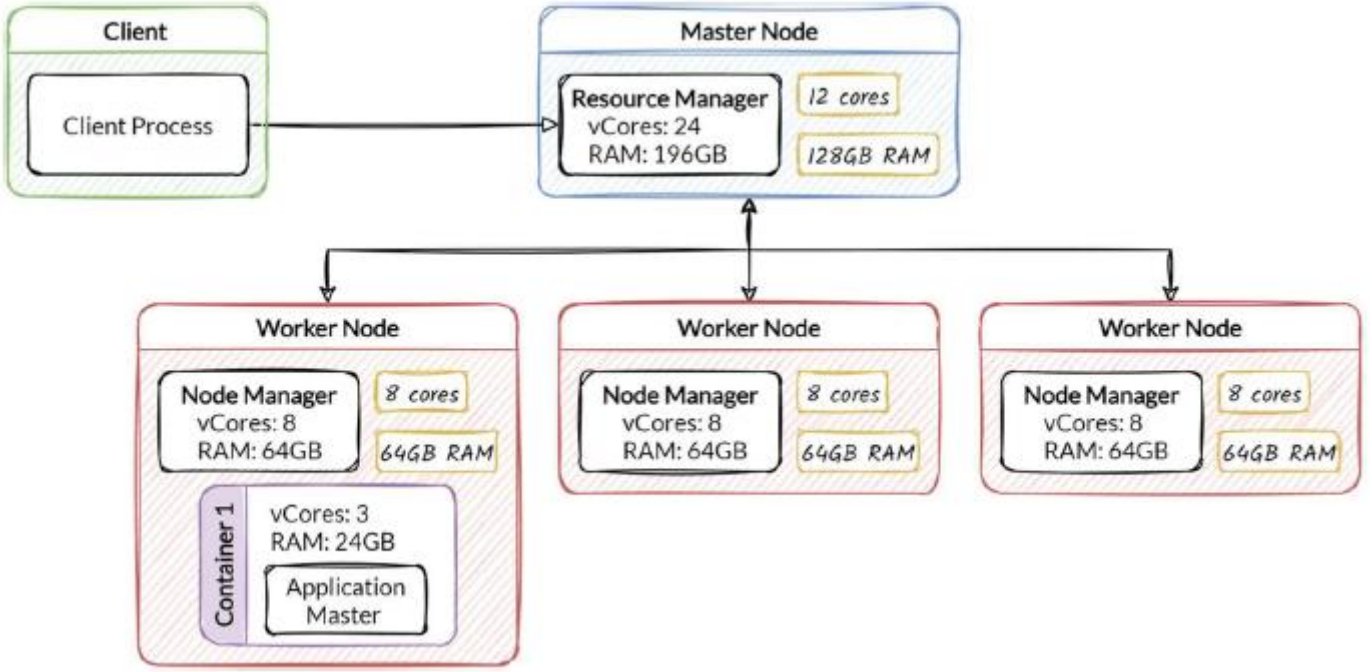
The client submits a MapReduce job to the Hadoop cluster. The job includes the Mapper and Reducer code, and configuration parameters. The data is already stored in the Hadoop Distributed File System (HDFS)



Map Reduce Scenario

2.ResourceManager and Job Scheduling:

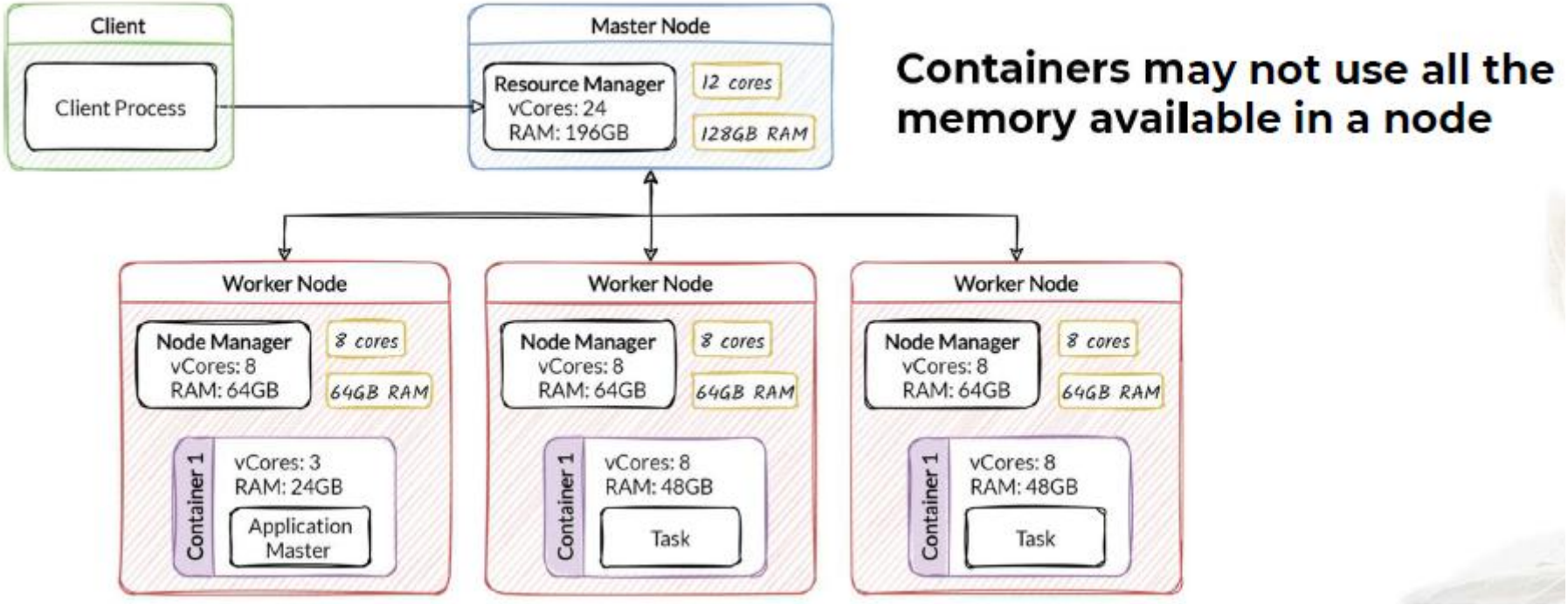
The ResourceManager receives the job submission and delegates the job to the ApplicationsManager.
The ApplicationsManager allocates a new ApplicationMaster (AppMaster) instance for the job.



Map Reduce Scenario

3.ApplicationMaster Initialization:

The ApplicationMaster negotiates resources from the ResourceManager and requests containers (execution environments) on various NodeManagers (Data Nodes).



Map Reduce Scenario

4.Data Splitting:

- The ApplicationMaster communicates with the NameNode to locate the input data blocks stored across the DataNodes.
- The input data is split into chunks, and the splits are assigned to the containers on the DataNodes where the data is located.

5.Mapper Execution:

- The NodeManagers launch Mapper tasks in the allocated containers.
- Each Mapper processes its assigned data chunk, producing intermediate key-value pairs.

Map Reduce Scenario

6.Shuffling and Sorting:

- The intermediate key-value pairs are shuffled and sorted by key. This involves transferring data across the network to group all values for the same key together.

7.Reducer Execution:

- The NodeManagers launch Reducer tasks in containers.
- Each Reducer processes its assigned key group, aggregating the values to produce the final output.

8.Writing the Output:

- The Reducers write the final output to HDFS, which is managed by the NameNode.
- The ApplicationMaster monitors the job's progress and updates
- the ResourceManager.

9.Job Completion:

- Once all Mapper and Reducer tasks are complete, the ApplicationMaster informs the ResourceManager.
- The client is notified of the job's completion, and the final
- results are available in HDFS

Summary

1. The client launches the process (connection with the Resource Manager)
2. The Resource Manager requests a container where the Application Master is executed
3. The Application Master requests the containers to execute all the tasks (in different nodes)
4. All the tasks are executed in the containers. Containers are released once its tasks are finished
5. The Application Master finishes when all tasks have been completed and release the container

Map and Reduce functions in Python

Map Function

```
lst = [1, 2, 3, 4]
list(map(lambda x: x*x, lst))
```

[1, 4, 9, 16]

```
def square(x):
    return x*x
list(map(square, lst))
```

[1, 4, 9, 16]

Map and Reduce functions in Python

Reduce Function

```
from functools import reduce
reduce(lambda x, y: x + y, lst)
10

def add_reduce(x, y):
out = x + y
print(f"{x}+{y}-->{out}")
return out
reduce(add_reduce,lst)

3<--2+1
6<--3+3
10<--4+6
10
```

Map Reduce Using Hadoop and MRJob Python Library

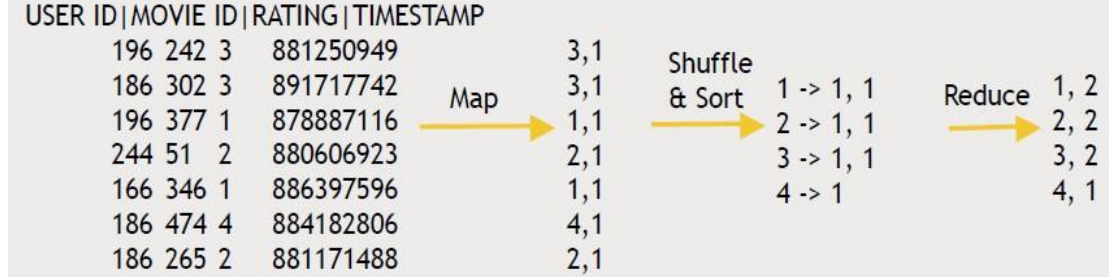
Map Reduce Movie Ratings Count Example

User ID| Movie ID| Rating | Time Stamp
0 50 5 881250949
0 172 5 881250949
0 133 1 881250949
196 242 3 881250949
186 302 3 891717742
22 377 1 878887116
244 51 2 880606923
166 346 1 886397596
298 474 4 884182806
115 265 2 881171488
253 465 5 891628467
305 451 3 886324817

Map Reduce Movie Ratings Count Example

```python
from mrjob.job import MRJob
from mrjob.step import MRStep

class RatingsBreakdown(MRJob):

    def steps(self):
        return [
            MRStep(mapper=self.mapper_get_ratings,
                   reducer=self.reducer_count_ratings)
        ]

    def mapper_get_ratings(self, _, line):
        (userID, movieID, rating, timestamp) = line.split('\t')
        yield rating, 1

    def reducer_count_ratings(self, key, values):
        yield key, sum(values)

if __name__ == '__main__':
    RatingsBreakdown.run()
```

Writing the Mapper



Map Reduce Example – Two Steps (Sorting after Counting)

```
from mrjob.job import MRJob
from mrjob.step import MRStep

class RatingsBreakdown(MRJob):
    def steps(self):
        return [
            MRStep(mapper=self.mapper_get_ratings,
                   reducer=self.reducer_count_ratings),
            MRStep(reducer=self.reducer_sorted_output)
        ]

    def mapper_get_ratings(self, _, line):
        (userID, movieID, rating, timestamp) = line.split('\t')
        yield movieID, 1

    def reducer_count_ratings(self, key, values):
        yield str(sum(values)).zfill(5), key

    def reducer_sorted_output(self, count, movies):
        for movie in movies:
            yield movie, count
if __name__ == '__main__':
    RatingsBreakdown.run()
```

Real-World Use Case: Log Analysis for Ad Click Fraud Detection

Company Example: Facebook

Objective: Detect fraudulent ad clicks by analyzing massive volumes of server logs.
Data Source: Web server logs from Facebook's ad delivery infrastructure.
Scale: Billions of clicks daily across millions of IP addresses

Data Structure (Sample Log Entry):

{ "timestamp": "2025-03-20T14:33:21Z",
"user_id": "u98374",
"ad_id": "ad4562",
"ip_address": "192.168.1.5",
"click": true,
"device_type": "mobile",
"location": "New York, USA"}

Processing with MapReduce – Facebook Example

Map Phase (What We're Mapping):

Input: Individual log entries

Mapper Logic:

- **Extract key fields:** ip_address, timestamp, user_id, ad_id
- **Emit key-value pairs:** (ip_address, "timestamp|user_id|ad_id")
- **Example output:** ("192.168.1.5", "14:33:21|u98374|ad4562")

Shuffle & Group:

Framework groups all clicks by IP address:

("192.168.1.5", ["14:33:21|u98374|ad4562", "14:33:45|u98374|ad4562", "14:34:12|u98374|ad4562"])

Processing with MapReduce – Facebook Example

Reduce Phase (What We're Reducing):

Input: All clicks grouped by IP address

Reducer Logic:

- **Count total clicks per IP:** `click_count = list.length`
- **Calculate click velocity:** `clicks_per_minute = count / time_window`
- **Detect patterns:** Same user_id + rapid clicking + same ad_id
- **Output:** `("192.168.1.5", "count=47, velocity=15.7/min, fraud_score=0.89")`

Outcome:

- **Fraud Detection:** IPs with > 10 clicks/minute + same user patterns = **HIGH FRAUD RISK**
- **Action:** Block suspicious IPs and preserve legitimate ad revenue
- **Business Value:** Protects millions in advertising spend daily
- **Key Insight:** We're **mapping each click event to its IP address** and **reducing by aggregating all clicks per IP** to detect abnormal patterns that indicate bot activity.

Hadoop as a Distributed Operating System

Hadoop effectively acts like an **Operating System for a cluster of computers**. It makes **many machines behave like one** — transparently to the user.

Traditional OS Function	Hadoop Equivalent	Explanation
File System	**HDFS (Hadoop Distributed File System)**	Stores data across multiple machines, replicates for fault tolerance, presents one logical filesystem view.
Process Scheduling	**YARN (Yet Another Resource Negotiator)**	Decides where and when to run distributed tasks across the cluster — just like a CPU scheduler, but for hundreds of nodes.
Resource Allocation	**YARN ResourceManager & NodeManagers**	Manages memory, CPU, and containers across nodes to ensure balanced usage and fairness.
Abstraction Layer	**Hadoop APIs (MapReduce, Spark, Hive)**	Let users submit jobs as if running on one large system, hiding the complexity of network, failures, and data locality.
Fault Handling	**HDFS replication, YARN task retry**	Detects node failures, re-replicates data, and re-runs failed tasks automatically.

How Hadoop “Fuses” Machines Together

- **Storage Unification (HDFS)**
  - Splits large files into blocks (e.g., 128 MB each).
  - Distributes and replicates those blocks across many machines.
  - Provides one unified namespace like /user/data/....
- **Computation Unification (MapReduce/YARN)**
  - Treats the cluster’s CPUs and RAM as a shared pool of compute resources.
  - Schedules tasks *where the data is located* (data locality).
  - Retries failed tasks automatically — users don’t see machine boundaries.
- **Transparency**
  - User submits a job to Hadoop, not to a specific server.
  - The system automatically finds resources, moves data, and handles recovery.
  - From the user’s perspective, it’s like running on one giant computer.

Single Machine OS vs Hadoop

Concept	Single-Machine OS	Hadoop (Cluster OS)
Hardware	CPU, RAM, Disk	Multiple servers (nodes)
File System	ext4, NTFS	HDFS
Scheduler	CPU scheduler	YARN ResourceManager
Process	Local process	Distributed container
Memory management	Per process	Per container across nodes
Fault recovery	Process restart	Node & task re-execution
User view	“My PC”	“My cluster”