

Introduction to the World of Big Data

1. What Is Big Data?

Big Data means:

- Very **large amounts of data**
- Data that is **complex**
- Data that comes **very fast**
- Data that can be **structured, semi-structured, or unstructured**

Example:

Every day the world generates **2.5 quintillion bytes** of data from WhatsApp, Facebook, YouTube, sensors, and more.

2. Why Traditional Databases Failed

Traditional databases (RDBMS) failed because they:

- Cannot handle **huge data volumes** (terabytes → petabytes)
- Become **very expensive** when scaled
- Cannot store **unstructured** or **semi-structured** data
- Cannot handle **high-speed streaming data**

Big Data technologies were created to solve these problems.

3. The 3 Vs of Big Data

Big Data has 3 special features:

1) Volume

- Refers to the **size of the data** generated and stored.

- Big Data reaches **terabytes, petabytes, and even zettabytes**.
- Growth is driven by social media, online transactions, sensors, GPS, etc.
- **Example:**
Facebook generates **about 500 terabytes of data per day**.

2) Velocity

- Refers to how **fast data is produced and needs to be processed**.
- Every 60 seconds:
 - 3.3 million Facebook posts
 - 450,000 tweets
 - 400 hours of YouTube videos

3) Variety

- Different types of data:
 - **Structured:** tables (E.g. employee database, bank records)
 - **Semi-structured:** JSON, XML, HTML (Has some structure but does not fit traditional tables.)
 - **Unstructured:** photos, videos, emails, social media posts, sensor logs



Structured Data



Unstructured Data



Semi-Structured Data

4. Sources of Big Data

Big Data comes from many places:

1) Sensors

- Accelerometers in phones
- Proximity sensors
- Vehicle sensors

- Medical devices

2) Healthcare

- Electronic Health Records (EHRs)
- Patient portals
- Clinical repositories

3) Machines

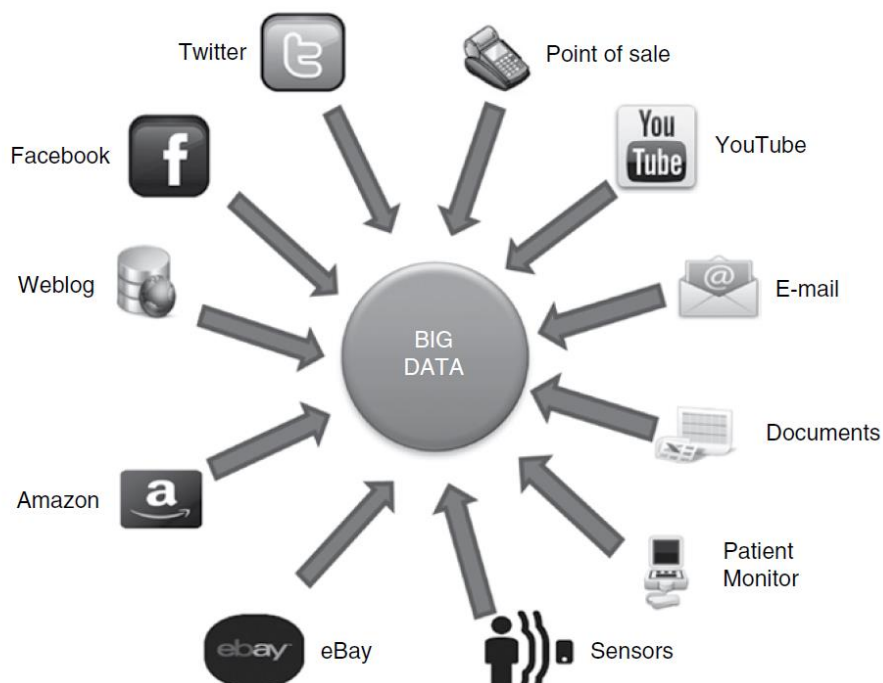
- Airplane black boxes
- Satellites

4) Websites

- Click-stream data (what users click)

5) Organizations

- Emails
- Documents
- Internal transactions



5. Types of Data

There are two broad categories :

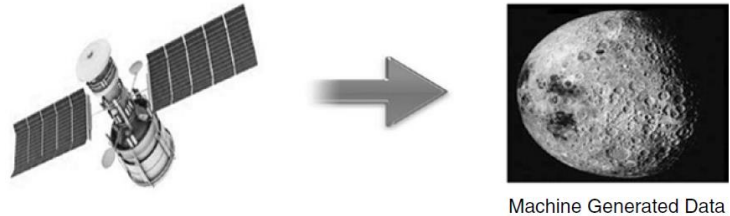
A) Human-Generated Data

- Social media posts
- Emails
- Documents



B) Machine-Generated Data

- Sensors
- Weather systems
- Satellites



Data formats:

1. **Structured** → tables, rows, columns
2. **Unstructured** → images, videos, PDFs
3. **Semi-structured** → XML, JSON

6. Big Data Infrastructure

Big Data uses special technologies:

Hadoop

- An open-source framework written in Java.
- Stores and processes huge amounts of structured, semi-structured, and unstructured data.
- Uses a cluster of many inexpensive machines (commodity hardware).
- Runs on **cheap hardware** (commodity servers)
- Highly scalable: performance increases by adding more nodes.

- Files are written once and read many times.
- Designed to be cost-effective and fault-tolerant.

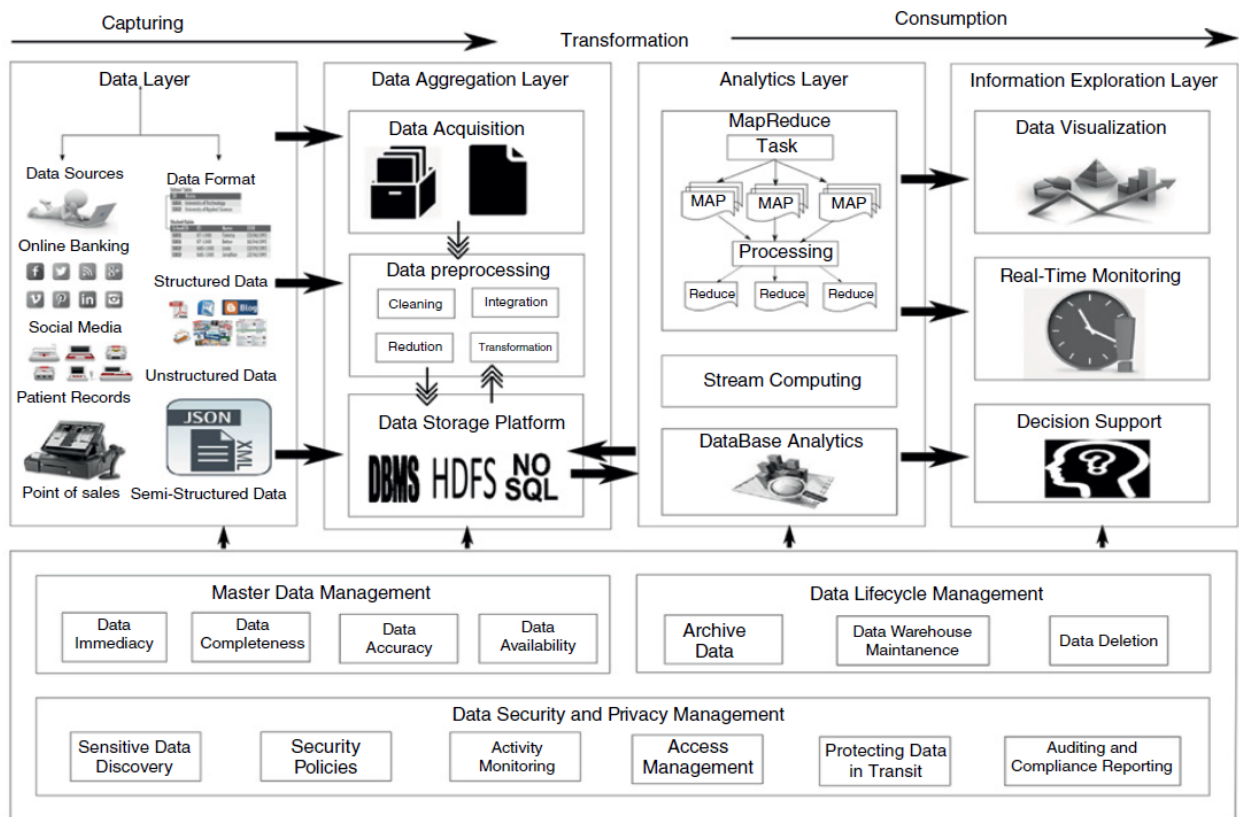
HDFS (Hadoop Distributed File System)

- A distributed file system for storing massive datasets across many machines.
- Works well with low-cost hardware.
- Stores data in a **write-once, read-many** formats.
- Handles data from multiple sources.
- Very fault-tolerant: data is replicated across several machines.

MapReduce

- A batch-processing programming model inside Hadoop.
- Uses a **divide-and-conquer** approach:
 - **Map** phase → splits the task into smaller parts
 - **Reduce** phase → combines the results
- Processes data **in parallel** across many machines.
- Highly scalable and fault-tolerant.
- Faster than traditional systems because processing happens **where the data is stored**, not moved elsewhere.

7. Big Data Life Cycle



The Big Data journey has several steps:

1. Data Generation

- Data is created continuously from many sources:
 - Social media
 - Online banking
 - Sensors / IoT
 - Smartphones
 - Transactions
 - Emails, images, videos
- This is where the life cycle starts.

2. Data Aggregation (Collection + Transfer)

- Raw data from different sources is:
 - **Collected**
 - **Transferred** to storage systems (HDFS, NoSQL, DBMS)
- Goals:
 - Bring all data into one place
 - Handle data arriving at high velocity
 - Compress or organize data if needed

3. Data Preprocessing

Raw data is usually **dirty, incomplete, inconsistent**, so it must be cleaned and prepared.

The four main stages:

a) Data Cleaning

- Fix missing values
- Correct errors and inconsistencies
- Remove duplicates
- Reduce noise

b) Data Integration

- Combine data from many sources into one unified dataset
- Example sources:
 - Smartphones
 - Online transactions
 - Organization files
 - Social media posts
 - Sensors, satellite images

c) Data Reduction

- Reduce data size while keeping important information
- Techniques:
 - Compression
 - Dimensionality reduction
 - Attribute selection
 - Sampling

d) Data Transformation

- Convert data into a useful, consistent format
- Techniques:
 - Smoothing (remove noise)
 - Aggregation (combine values)
 - Generalization (higher-level categories)
 - Discretization (convert numeric → categories)

4. Analytics

Data is analyzed to extract insights and reveal patterns.

Three main types:

a) Descriptive analytics

- What happened?
- Summaries, statistics, trends

b) Predictive analytics

- What will happen?
- Machine learning, forecasting

c) Prescriptive analytics

- What should we do?

- Recommendations, optimization

Analytics uses tools such as MapReduce, YARN, NoSQL engines, and streaming systems.

5. Visualization

- Results are presented visually so humans can understand them easily.
- Tools: dashboards, Tableau, Power BI, charts.
- Common charts:
 - Line, bar, scatter, bubble, pie charts

6. Decision Making

- The final stage.
- Organizations use the insights to make:
 - Business decisions
 - Customer analysis
 - Operational improvements
 - Strategic planning

8. Big Data Applications

Big Data is used in almost every industry to improve efficiency, reduce costs, and make smarter decisions.

1. Healthcare

Big Data helps improve patient care and medical research.

Uses:

- **Early disease detection**
(Analyzing patient history, lab results, genomics to identify diseases early.)
- **Remote patient monitoring**
(Wearable devices send real-time health data to doctors.)

- **Medical image analysis**
(MRI, CT scans analyzed using AI.)
- **Personalized treatment**
(Treatment plans based on patient data.)
- **Public health monitoring**
(Tracking epidemics using large data sets.)

2. Banking & Financial Services

Big Data increases security and improves customer service.

Uses:

- **Fraud detection**
(Analyzing transaction patterns to catch fraudulent credit card activity.)
- **Credit risk scoring**
(Predicting if a customer is likely to repay loans.)
- **Wealth management**
(Personalized investment recommendations.)
- **Customer retention analysis**
(Identifying customers who may switch banks.)
- **Real-time analytics**
(Detecting fraud within milliseconds.)

3. Telecom

Big Data improves network quality and customer experience.

Uses:

- **Network optimization**
(Analyzing traffic patterns to prevent slowdowns and outages.)
- **Personalized data plans**
(Offering plans based on customer usage.)

- **Reducing churn**
(Identifying customers likely to leave the company.)
- **Call center analytics**
(Detecting common issues, improving resolution time.)
- **Predicting service demand**
(Using historical usage data.)

4. Marketing & Retail

Big Data helps companies understand and predict customer behavior.

Uses:

- **Customer behavior analysis**
(Purchase history, website clicks, social media activity.)
- **Recommendation systems**
(Products shown on Amazon, Netflix, YouTube.)
- **Targeted advertising**
(Ads tailored to user interests.)
- **Sales forecasting**
(Predicting demand for products.)

5. Finance (General Business Analytics)

Beyond traditional banking, Big Data supports financial operations.

Uses:

- **Real-time fraud detection**
(Streaming data analysis to detect suspicious activity instantly.)
- **Customer retention analysis**
(Monitoring customer actions across platforms.)
- **Risk management**
(Predicting market risks, interest rate changes.)

- **Investment insights**
(Analyzing market trends and portfolios.)

6. Additional Application Areas

Web Analysis

- Uses search engine data and social media activity to show relevant ads.

Call Centers

- Identifies common problems and improves staff performance.

Agriculture

- Uses sensors to monitor crops and optimize growth.

Smartphones

- Facial recognition
- Security features
- Personalized assistant tools

Lecture Summary

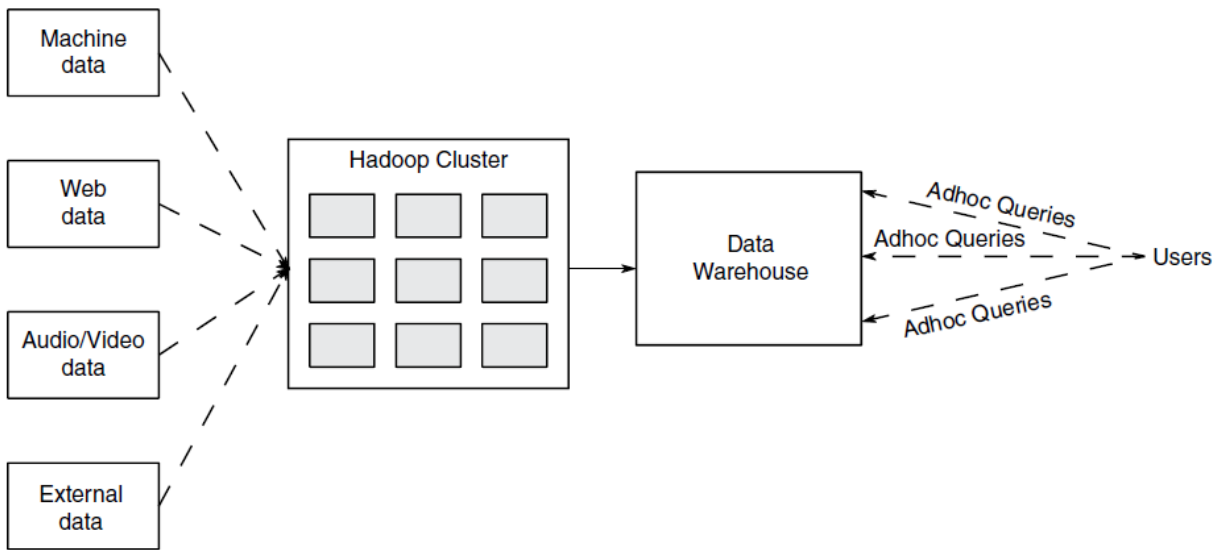
In this lecture you learned:

- What Big Data is
- Why traditional systems failed
- The 3 Vs (Volume, Velocity, Variety)
- Where Big Data comes from
- Types of data
- What Hadoop, HDFS, and MapReduce do
- The Big Data life cycle
- Real applications

Lecture 2

Big Data Storage & Architecture

Big Data needs **special storage methods** because normal computers cannot store or process huge amounts of data efficiently:



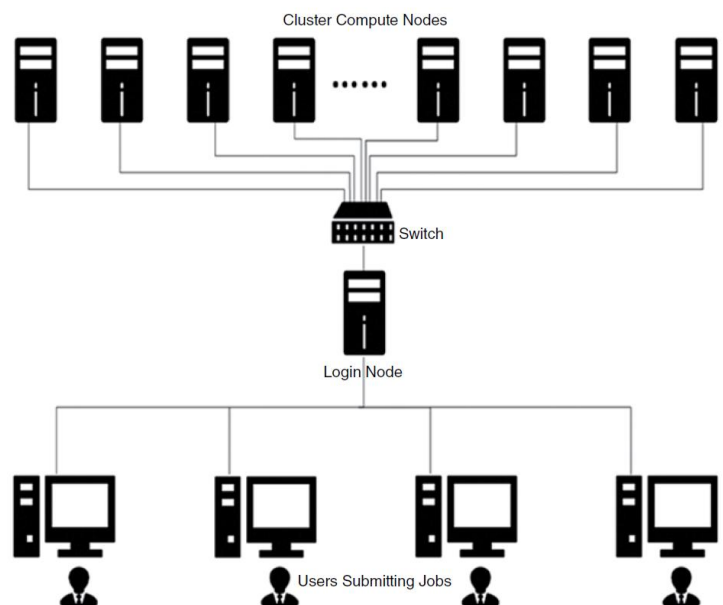
In this lecture, you will learn how Big Data storage works—**clusters, distributed systems, sharding, replication, and HDFS**.

1. What Is Cluster Computing?

Cluster computing means:

- Many **separate computers** (nodes)
- Connected together through a **local area network (LAN)**.
- Working as **one powerful system**

Clusters provide:



- **High availability** (system keeps working even if one node fails)
- **Fault tolerance:**
Redundant nodes ensure continuous service.
- **Scalability:**
You can add more nodes whenever needed.
- **High performance at low cost:**
Uses **commodity hardware** (cheap machines).

2. Types of Clusters

Clusters are designed differently depending on what the organization needs.

A. High-Availability Cluster

- Designed to avoid downtime
- If one node fails → another takes over automatically (failover)
- Suitable for systems that must run 24/7:
 - **Banking**
 - **Billing systems**
 - **E-commerce**
- Has redundant nodes for safety.
- Ensures data integrity even during failures.

B. Load-Balancing Cluster

- Distributes workload across many nodes
- Prevents overload on a single computer
- Improves performance (Improves response time.)
- Great for web servers, cloud services, online applications.

Common load balancing algorithms:

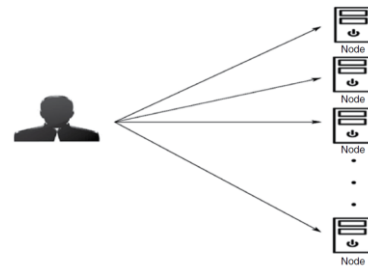
- **Round Robin:** Each request goes to the next server in a circle.
- **Weight-Based:** Nodes with higher capacity get more requests.
- **Random:** Requests sent to random nodes.
- **Server Affinity:** Same client is routed to the same server.

3. Cluster Structures

Clusters can be connected internally in **two main architectural ways**:

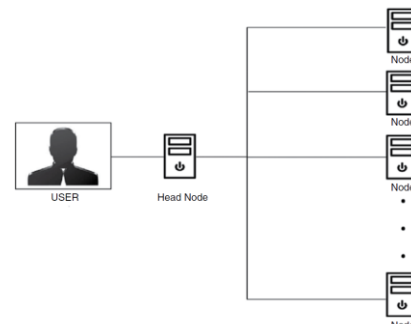
A. Symmetric Cluster

- All nodes work equally
- Every node can run applications.
- No special “master” node.
- Easy to scale: add or remove nodes anytime.



B. Asymmetric Cluster

- One **head node** controls everything.
- Other nodes mainly perform processing tasks.
- Users connect only to the head node.



4. Data Distribution Models

When Big Data grows too large, we must distribute data across many machines.

There are **three main techniques**:

1. **Sharding (Data Partitioning)**
2. **Replication (Copying data)**
3. **Sharding + Replication together (Combined approach)**

Let's understand each one.

5. Sharding (Data Partitioning)

Sharding means:

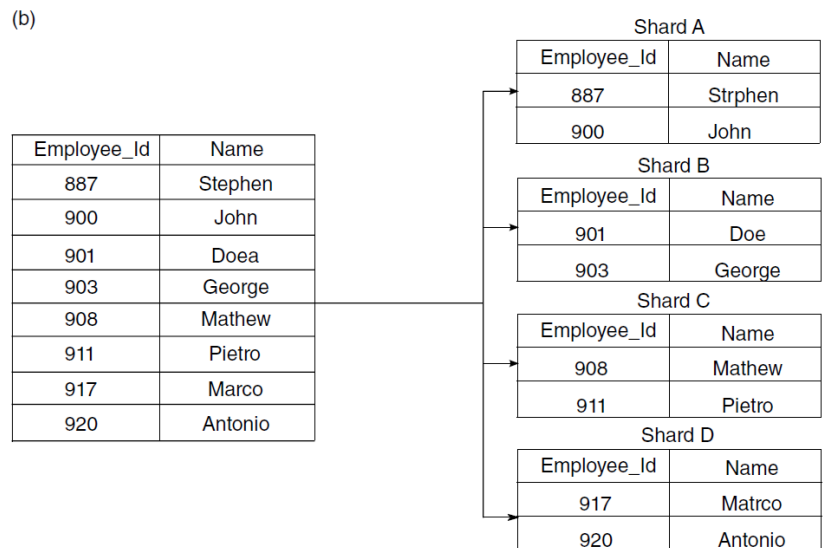
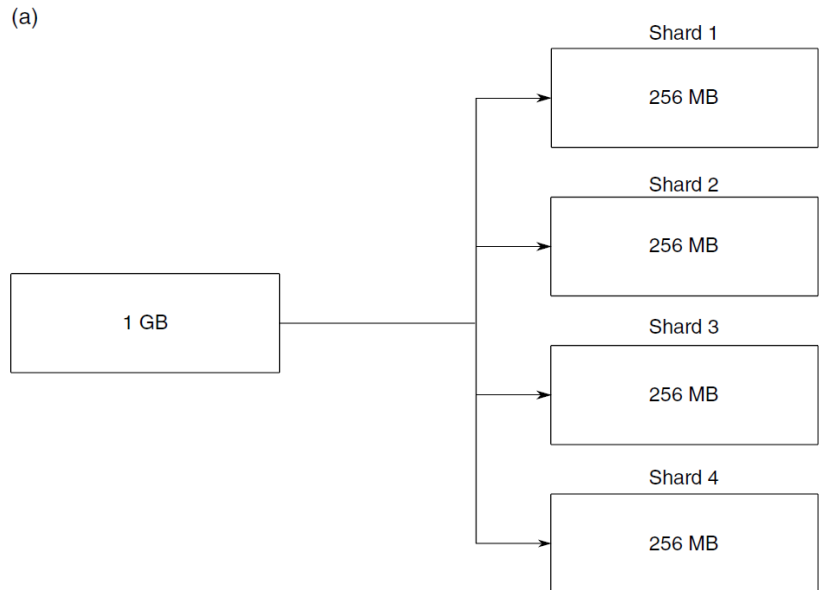
- Splitting large data into **small parts** called *shards*
- Storing each shard on a **different node**

Example:

A 1 GB file → split into 4×256 MB shards → each stored on a different server.

Benefits of Sharding:

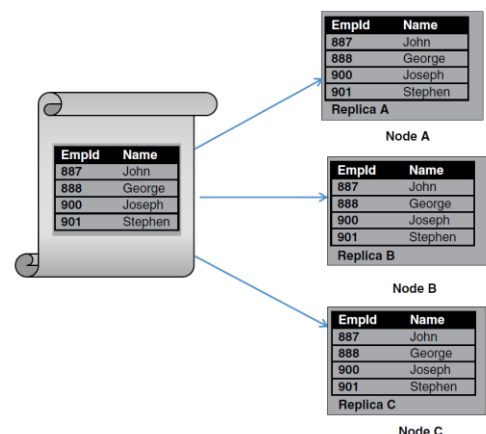
- Reduces load on each node
- Improves query performance
- Greatly increases scalability
- Limits failure impact (If one node fails, **only its shard is lost**)



6. Replication (Copying Data)

Replication means:

- Making **multiple copies** of the same data
- Storing copies on different nodes
- Prevents data loss

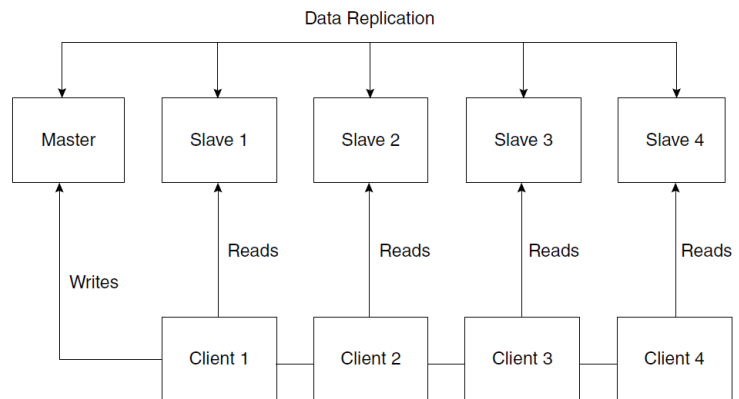


If Node A fails → Node B still has the same data.

Two replication models exist:

A. Master-Slave Model

- One node = Master
- Other nodes = Slaves
- Master node handles all **writes**
- Slaves handle **reads**

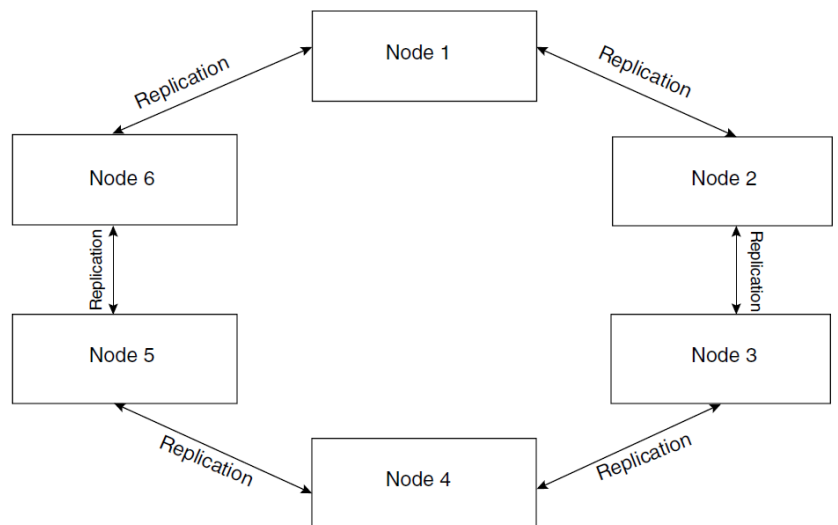


Disadvantages

- If master fails → no writes possible → **single point of failure**
- Scalability limited for writes

B. Peer-to-Peer Model

- **No master node**
- All nodes are equal
- Each node can handle read + write (Better for reliability.)
- No single point of failure
- Good for distributed global systems



Advantages

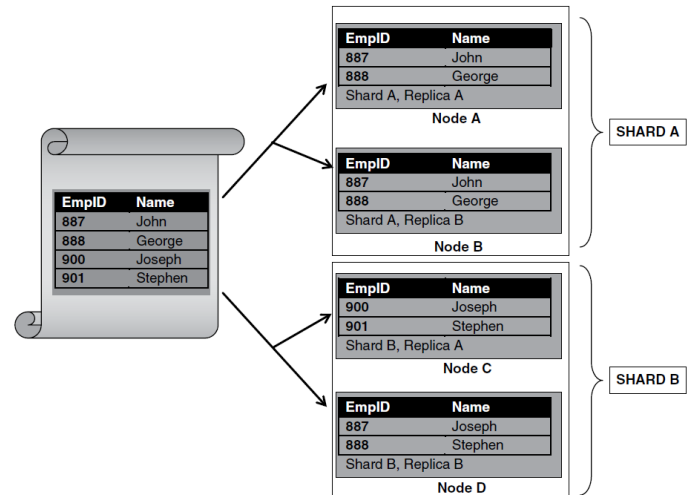
- More reliable
- Better scalability
- More balanced resource usage

7. Combining Sharding & Replication

This combined model is used in **real-world Big Data systems** like Cassandra, MongoDB, HDFS.

How it works

1. **First shard** the dataset;
2. **Then replicate each shard** to multiple nodes.



Example

- **Shard A** → stored on **Node 1** and **Node 2**
- **Shard B** → stored on **Node 3** and **Node 4**

Benefits

- **High scalability** (due to sharding)
- **High reliability** (due to replication)
- Ensures both efficiency and fault tolerance
- Reduces risk of data loss

8. Distributed File Systems (DFS)

A Distributed File System:

- Stores files across many machines (different cluster nodes)
- Appears to users like **one single system** → Users don't need to know where the file is physically stored.
- Supports multiple users reading/writing at the same time
- Uses replication for reliability → Copies of data (replicas) are stored on several nodes.

- Improves fault tolerance → If one node fails, data is still available from other nodes.

Most famous DFS: **Hadoop Distributed File System (HDFS)**.

9. Why HDFS Is Needed

Traditional storage cannot handle:

- Multi-terabyte data
- High-speed data
- Low-cost scaling

HDFS solves this by:

- Splitting files into large blocks (64 MB default)
- Storing blocks across DataNodes
- Replicating blocks (default: 3 copies)
- Running on cheap commodity hardware

10. Scaling Big Data Storage

Two ways to scale:

A. Scaling Up (Vertical)

Increase the power of one single server by adding:

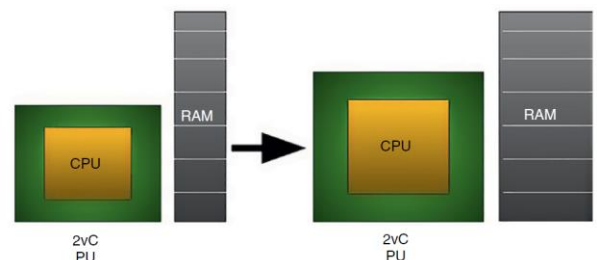
- more RAM, CPU cores, Faster hardware

Advantages:

- Simple to upgrade
- No need to change system architecture

Disadvantages:

- **Very expensive**



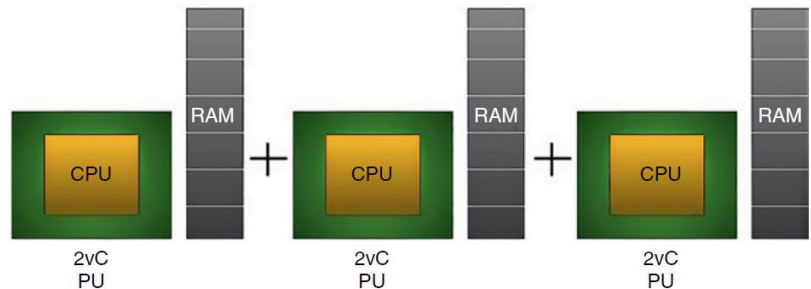
- **Has a hard limit** (you cannot upgrade forever)
- If the single server fails → everything stops
- Not suitable for Big Data

B. Scaling Out (Horizontal)

Add **more servers (nodes)** instead of upgrading a single one.

Advantages:

- Easy to expand anytime
- Much lower cost
- No hardware limits
- Highly fault-tolerant
- Perfect for Big Data systems



Used by:

- **Hadoop / HDFS**
- **NoSQL databases** like MongoDB, Cassandra, HBase

11. Key Big Data Storage Concepts Summary

Concept	Meaning	Why Important
Cluster	Many machines work as one	More power, more reliability
Sharding	Split data into pieces	Scalability
Replication	Duplicate data	Fault tolerance
DFS	Files stored across many nodes	Large, reliable storage
HDFS	Hadoop's storage system	Handles big data easily
Scaling Out	Add machines	Cheapest + most scalable

Lecture 2 Summary

You learned:

- How cluster computing works
- Types of clusters
- Sharding and replication
- Why distributed file systems are needed
- How HDFS stores and protects data
- Difference between scaling up and scaling out

NoSQL Databases

1. Why Do We Need NoSQL?

NoSQL = Not Only SQL

Traditional relational databases (RDBMS/SQL) were built for **structured data**, **fixed schemas**, and **vertical scaling** (adding power to a single machine). However, modern systems generate **massive, fast, and diverse data**—often unstructured or semi-structured—which exceeds the limits of traditional databases.

- Modern applications—especially Big Data and real-time systems—demand storage that handles massive, varied data formats.
- NoSQL databases address the shortcomings of RDBMS by offering:
 - **Horizontal scalability** using commodity hardware
 - **Flexible schemas** (schema less design)
 - **Support for structured, semi-structured, and unstructured data**
 - **High performance** for large-scale distributed systems
 - **Fault tolerance**
- NoSQL systems make it easy to integrate huge data sources and adapt to changing data formats.

Types of NoSQL databases include:

- Key-value stores
- Document-oriented databases
- Column-oriented databases
- Graph databases

NoSQL is ideal for:

- Real-time web apps
- IoT
- Social media
- Large e-commerce systems
- Sensor data

Key Features of NoSQL

NoSQL databases have the following key features:

1. Schema-less (Flexible Schema)

- No need to design a table structure first
- Every record can have **different fields**
- Easy to store messy or evolving data

2. Horizontal Scalability

- Add more machines (nodes) instead of upgrading one
- Supported using sharding + replication

3. Distributed Computing

- Data spread across many nodes
- Supports parallel processing

4. Low Cost

- Uses commodity hardware
- Extremely cost-effective

5. Handles Massive Volumes

- Designed to support billions of records
- Works well with unstructured data

2. CAP Theorem

The **CAP Theorem**, proposed by Eric Brewer, states that in a distributed database system, it is **impossible** to simultaneously guarantee all three of the following properties. A system can provide **only two at the same time**:

1. Consistency (C)

- Every user sees the **same data** at the same time.
- After a write, all reads return the latest value.
- Example: If multiple nodes serve the same read request, all must return **identical results**.

2. Availability (A)

- The system **always responds** to requests.
- Every read/write operation gets a reply (even if it indicates failure), and the system remains operational.

3. Partition Tolerance (P)

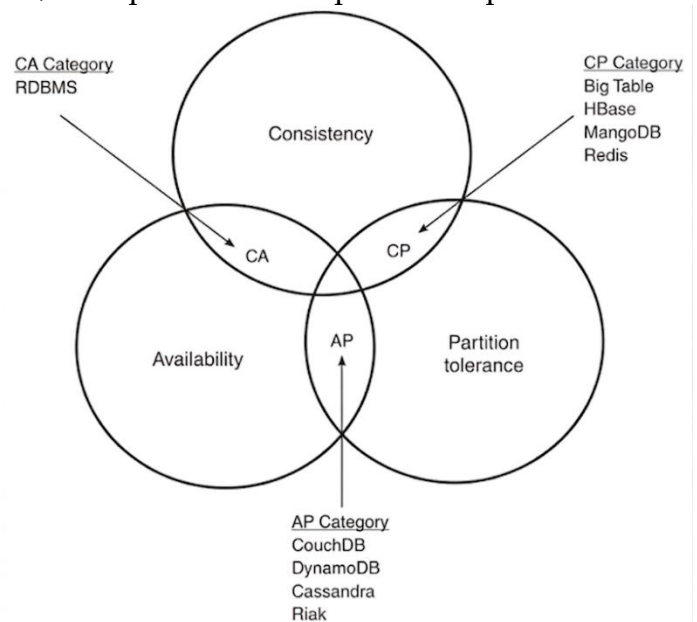
- The system keeps working **even when communication between nodes fails**.
- Even if the network is split into partitions, each part must still process requests.

The CAP theorem says a database can guarantee only **two** of the following three (never all three) at once:

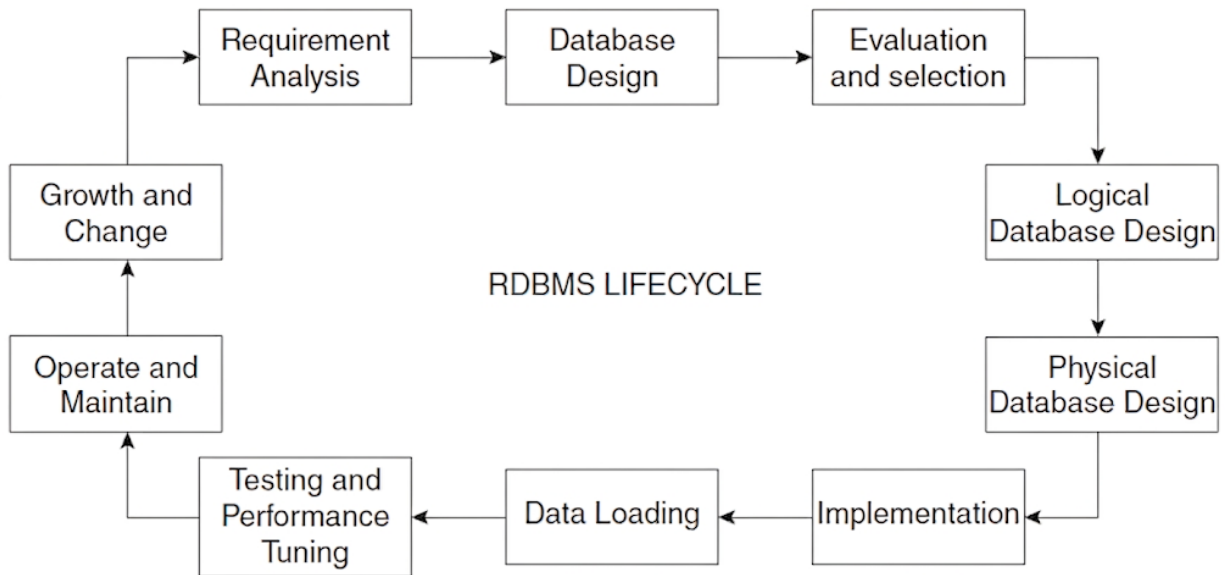
NoSQL databases are designed to achieve either:

- **AP (Availability + Partition Tolerance)**
- **CP (Consistency + Partition Tolerance)**

Traditional RDBMS = **CA** (Consistency + Availability)



Properties of a system following CAP theorem.



3. ACID vs BASE

Databases follow different models depending on their purpose.

SQL databases follow **ACID**, while **NoSQL databases** typically follow **BASE**.

These two models reflect two different philosophies: **strict reliability** vs **high scalability and flexibility**.

A. ACID (Used in SQL Databases)

ACID stands for **Atomicity, Consistency, Isolation, Durability**, and is designed for systems that require strict, reliable, error-free transactions—common in banking or financial systems.

1. Atomicity

- A transaction is **all or nothing**.
- If any part fails, the whole transaction is rolled back.
- No partial updates are ever saved.

2. Consistency

- The database must move from one **valid state** to another.
- After a transaction, all data must follow rules, constraints, and formats.
- Example:
Updating a float column (column of a table of type float) with a value of type varchar is rejected to keep consistency.

3. Isolation

- Transactions do **not interfere** with each other.
- Multiple users can modify data at the same time without conflicts.
- One transaction cannot affect another until it completes.

4. Durability

- Once a transaction is committed, it becomes **permanent**.
- Even if the system crashes immediately after committing, the data must still be preserved.
- Example:
If the system crashes before writing to disk, data is restored when the system comes back.

Best For

- Banking systems
- Financial transactions
- Inventory and billing systems
- Any environment where **accuracy is critical**

B. BASE (Used in NoSQL Databases)

NoSQL databases operate on distributed clusters, so they follow **BASE**, a more flexible model aligned with the **CAP theorem**.

BASE stands for **Basically Available, Soft State, Eventually Consistent**.

1. Basically Available

- The system is designed to **always be available**, even during failures.
- Responses may not always contain the latest data, but availability is guaranteed.

2. Soft State

- Data may be **temporarily inconsistent** because updates propagate across nodes slowly.
- Example:
If Node A is updated before Node B and a user reads from Node B, they get stale data.

3. Eventually Consistent

- After some time, all nodes will contain the **same** and **correct** data.
- Temporary inconsistencies are acceptable as long as they resolve later.
- Example:
Reading too soon after a write may return outdated results, but eventually the correct value appears.

Best For

- Social media applications
- Streaming platforms
- E-commerce analytics
- IoT and sensor networks
- Large-scale distributed systems

ACID vs BASE – Comparison Table

Aspect	ACID (SQL)	BASE (NoSQL)
Reliability	Very high	Moderate
Consistency	Immediate	Eventual
Availability	Not always	Always
Scalability	Vertical	Horizontal
Use Cases	Banking, finance	Big Data, social media

Hence,

- **ACID = Safety first**
Perfect accuracy, strict rules, no room for error.
- **BASE = Scale first**
Always available, flexible, tolerates temporary inconsistency.

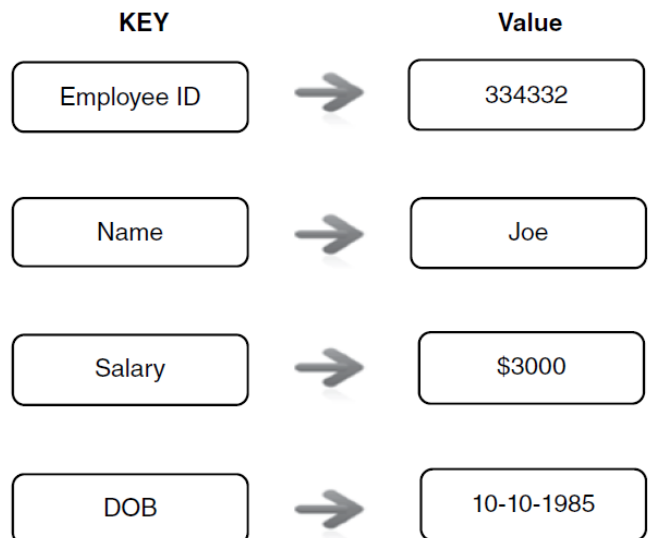
4. Types of NoSQL Databases

NoSQL databases come in four main categories. Each type is optimized for specific data models and use cases.

1. Key-Value Databases

Concept

- Data is stored as **key** → **value** pairs.
- Works like a **dictionary** / **hashmap**.
- **Key** = unique identifier
- **Value** = any data (string, object, number, etc.)
- No schema, extremely fast lookups.



Examples (of key-value databases)

- Redis
- Amazon DynamoDB
- Riak
- Memcached

Use Cases

- Shopping cart data
- User session storage
- Cache systems for performance
- Online gaming session data

Notes

- Fastest and simplest NoSQL model
- Ideal when data is accessed using a unique key

- No foreign keys / no complex relationships

2. Document-Oriented Databases

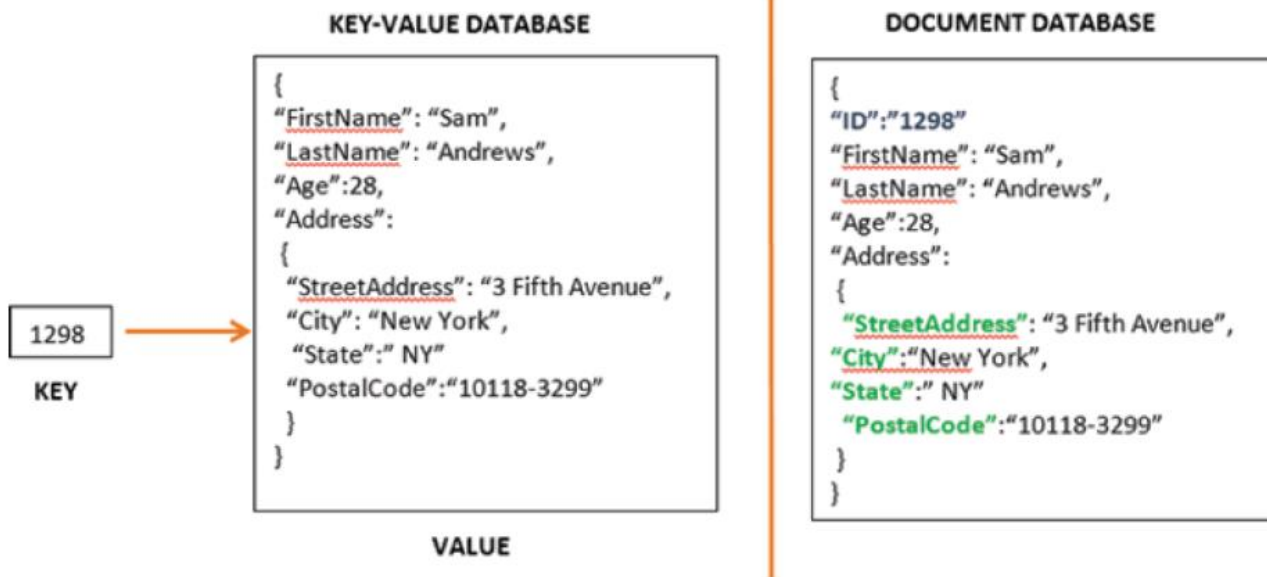
Concept

Store and retrieve data as **documents** (self-contained objects) using formats like:

- JSON
- BSON
- XML
- YAML

Each document may contain:

- Nested fields
- Arrays
- Complex structures



Examples

- MongoDB
- CouchDB

- Microsoft DocumentDB (now Azure Cosmos DB)

Use Cases

- User profiles
- Blogs / CMS
- Product catalogs
- E-commerce applications

Notes

- Horizontally scalable
- Documents are treated as complete units
- Easy to query by any field, not only primary key
- CouchDB supports ACID at the document level
- Data is semi-structured and flexible

3. Column-Store Databases

Concept

- Store data **column-by-column**, not row-by-row.
- Ideal for analytical queries where only selected columns are needed.
- Extremely efficient for large datasets.

Employee_Id	Name	Salary	City	Pin_code
3623	Tony	\$6000	Huntsville	35801
3636	Sam	\$5000	Anchorage	99501
3967	Williams	\$3000	Phoenix	85001
3987	Andrews	\$2000	Little Rock	72201

Row store database

Employee_Id	Name	Salary	City	Pincode
3623	Tony	\$6000	Huntsville	35801
3636	Sam	\$5000	Anchorage	99501

Column store database

Examples

- Apache Cassandra
- Apache HBase

Table: Order

Product_ID	Total_Amount	Product_desc
1000	\$250	AA
1023	\$800	BB
1900	\$365	CC

Internally row database will be stored like this,

1000 = 1000,\$250,AA; 1023 = 1023,\$800,BB; 1900 = 1900,365,CC;

Column Database will be stored like this,

Product_ID = 1000:1000, 1023:1023, 1900:1900; Total_Amount = 1000:\$250, 1023:\$800, 1900:\$365;

Product_desc = 1000:AA, 1023:BB, 1900:CC;

Use Cases

- Time-series data
- IoT sensor data
- Large-scale messaging systems
- Real-time analytics

Notes

- Excellent for OLAP (analytical queries), since fewer columns are read
- Built for massive distribution and fault tolerance
- Cassandra is highly available, fault-tolerant, and supports linear scalability
- Data is stored in **column families**

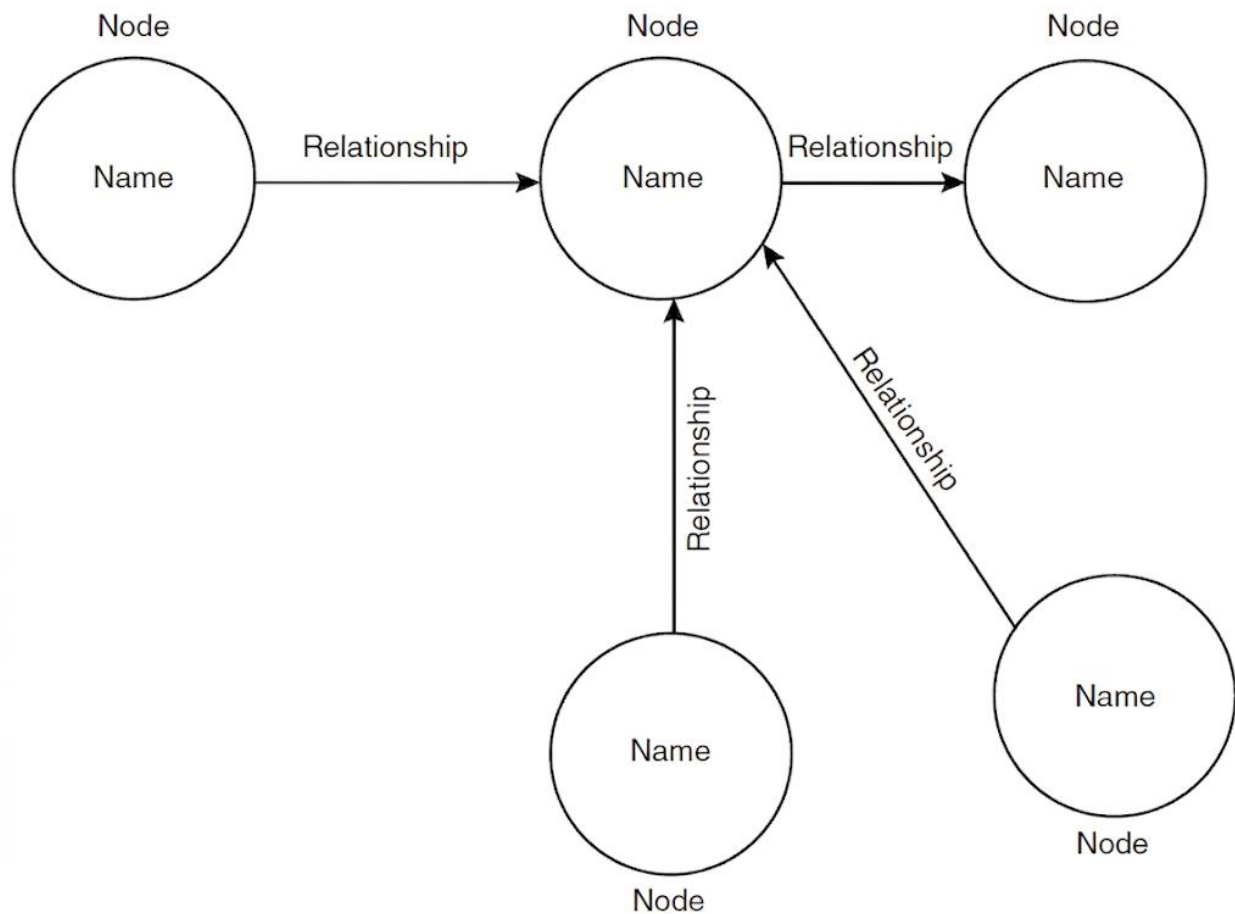
4. Graph Databases

Concept

Focus on **relationships**, not just data.

They consist of:

- **Nodes** → entities (e.g., people, products, accounts)
- **Edges** → relationships (e.g., friend-of, purchased, belongs-to)
- **Properties** → attributes stored on nodes or edges



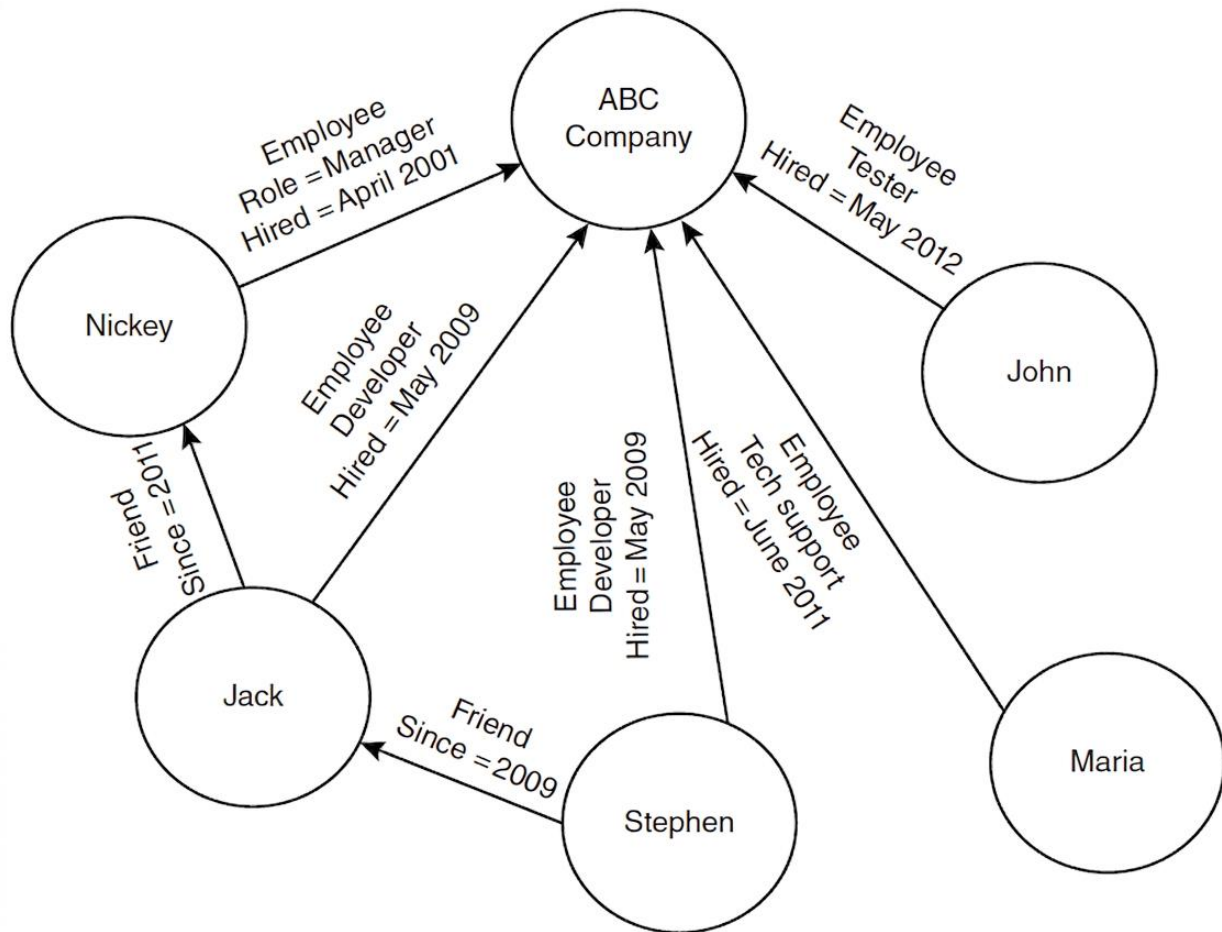
General representation of graph database.

Examples

- Neo4j
- OrientDB
- AllegroGraph

Use Cases

- Social networks
- Fraud detection
- Recommendation engines
- Knowledge graphs



Neo4J Relationships with properties.

Notes

- Relationships are first-class citizens
- Use **Cypher Query Language (CQL)** for queries (in Neo4j)
- Highly effective for traversing networks (friends-of-friends, etc.)
- Designed for semi-structured or connected data

Comparison Table

Type	Data Model	Best For	Examples
Key-Value	Key → Value pairs	Caching, sessions, carts	Redis, DynamoDB
Document	JSON/XML documents	Profiles, catalogs	MongoDB, CouchDB
Column-Store	Columns & column families	Analytics, time-series	Cassandra, HBase
Graph	Nodes + edges	Social networks, fraud, recommendations	Neo4j, OrientDB

5. NoSQL vs RDBMS

Feature	RDBMS	NoSQL
Schema	Fixed, rigid	Flexible, schema-less
Scalability	Vertical	Horizontal
Data Types	Mostly structured	Structured + semi-structured + unstructured
Query Language	SQL	Many (JSON queries, CQL, etc.)
Consistency Model	ACID	BASE
Cost	Expensive hardware	Low-cost commodity machines

6. CRUD Operations in NoSQL

NoSQL databases support:

- Create

- Read
- Update
- Delete

Example (MongoDB commands):

- use databaseName
- db.collection.insert({...})
- db.collection.find()
- db.collection.update()
- db.collection.remove()

7. When to Use NoSQL?

NoSQL is best when you need:

- Fast read/write
- Flexible data storage
- Large-scale distributed systems
- Real-time analytics
- Ability to store images, videos, logs, JSON, sensor data

Examples:

- Facebook feed
- YouTube recommendations
- Amazon product suggestions

Lecture 3 Summary

You learned:

- Why NoSQL is needed
- Differences between NoSQL & SQL
- CAP theorem
- ACID vs BASE
- Main NoSQL types
- Example technologies
- CRUD basics

This knowledge is essential for understanding Big Data storage systems and modern distributed applications.

Big Data Processing & Cloud Computing

This lecture explains how Big Data is processed, the difference between batch and real-time systems, the role of parallel and distributed computing, virtualization, and how cloud computing supports Big Data.

1. What Is Data Processing?

Data processing is the complete set of activities involved in transforming **raw data** into **meaningful, usable information**.

It includes:

- **Collecting data**
- **Manipulating and organizing data**
- **Processing and transforming data**
- **Storing data**
- **Producing useful information** for end users

Data becomes “information” only **after** it is processed, structured, and given meaning.

Where Data Comes From?

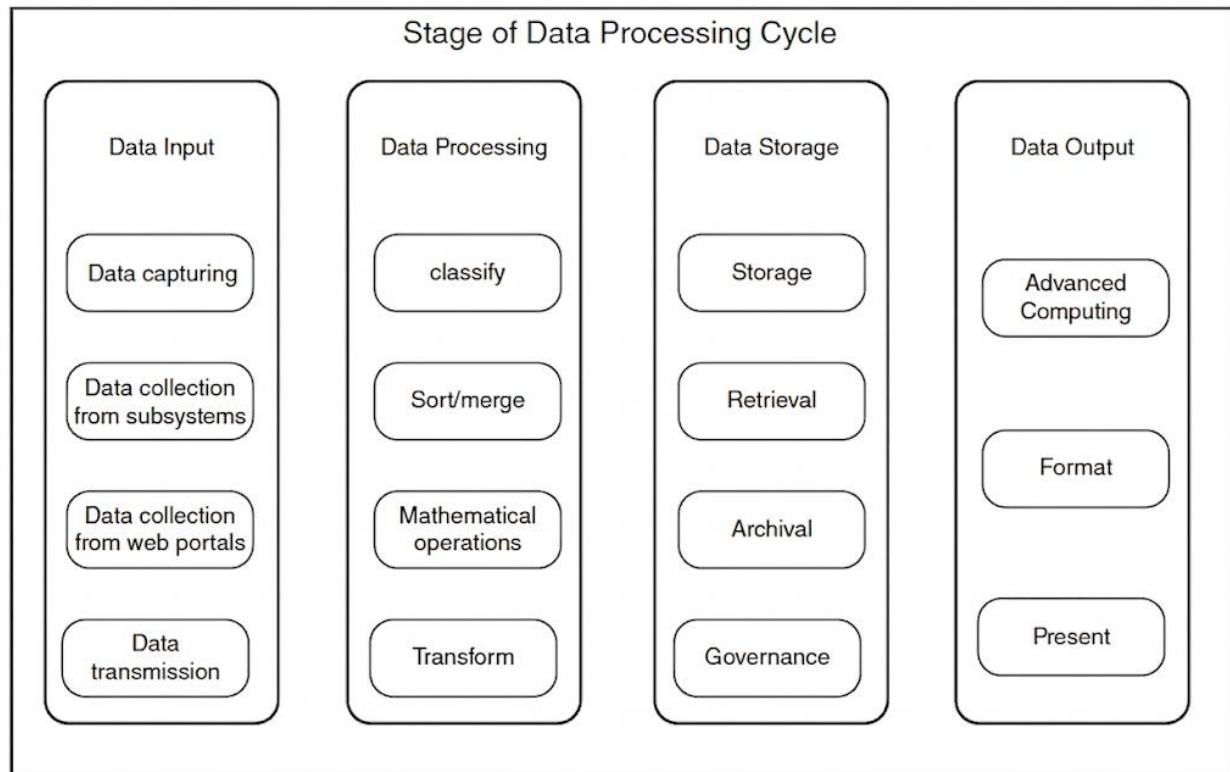
Data may originate from many different sources:

- Transactions
- Sensors
- Websites and web portals
- Observations
- Paper documents converted into digital form
- Machine-generated logs

Collecting raw data from these sources is called **data capture**.

The Data Processing Cycle

The images show the **stages** of the data processing cycle as follows:



Data processing cycle.

1. Data Input

Includes:

- Data capturing (This is the process of obtaining raw data at its original source.)
- Collecting from various sources (After data is captured, it may need to be gathered from different locations or systems.)
- Data transmission (This is the movement of data from one place to another. E.g., Sending data over a network (LAN, WAN, Internet); Uploading files to cloud storage; Transferring data between internal systems; Streaming sensor data to a server.)

2. Data Processing

Activities include:

- **Classifying** (This is the process of grouping data into categories based on shared characteristics. E.g., Categorizing students by grade level; Grouping products by type (electronics, clothing, food); Sorting emails into folders (spam, work, personal).)
- **Sorting/merging**
- **Mathematical operations** (adding, subtracting, multiplying, dividing) (Purpose: Generate meaningful numerical results for decision-making.)
- **Transforming** data into a machine-readable format (This means converting data into a format that computers can understand and process automatically.
 - **Examples:**
 - Encoding text as binary
 - Digitizing handwritten forms
 - Converting images into numerical pixel data
 - Formatting CSV, JSON, or XML files for software processing
 - **Purpose:**
Ensures data can be processed accurately by computer systems.)

3. Data Storage

Data is stored for:

- **Retrieval** (Storing data so it can be quickly accessed and used when needed.)
- **Archival** (Storing data long-term, usually because it's not needed frequently but must be kept.)
- **Governance** (Storing data to meet legal, regulatory, or organizational requirements.)

Storage is needed **only when the future value of the data exceeds its storage cost.**

4. Data Output

Final information is:

- Computed (This means the information has been fully processed by the system — calculations, classifications, summaries, or reports have been completed.)
 - **Examples:**
 - Total sales for the month
 - Average student grades
 - Generated statistical results
 - **Purpose:**
Provide accurate, meaningful results derived from raw data.)
- Formatted (The computed information is then arranged in a clear and organized structure so users can easily understand it.)
 - **Examples:**
 - Tables
 - Charts and graphs
 - Structured reports
 - Dashboards
 - **Purpose:**
Make the information readable, standardized, and visually clear.)
- Presented to users (Finally, the formatted information is delivered through an output medium.)
 - **Examples:**
 - Printed reports
 - On-screen displays
 - Mobile apps
 - Emails

- PDF documents
- **Purpose:**
Ensure users can view and use the information for decisions and actions.)

2. Types of Data Processing

1. Centralized Data Processing

- All data collection and processing occur in **one central location**.
- Requires minimal distributed resources.
- Simpler to maintain and secure.
- Suitable for organizations with a **single centralized service** or limited geographic distribution (example: a single-site company).

2. Distributed Data Processing

- Data collection and processing are spread across **multiple physical locations** or systems.
- Solves the limitations of centralized systems by enabling:
 - **Scalability:** Can add more machines as workload grows
 - **Parallel processing:** Multiple tasks processed simultaneously
 - **Fault tolerance:** If one node fails, others continue working

Architectural Models of Distributed Processing

Distributed processing can be implemented using several architectural models:

1. Client–Server Architecture

- The **client** handles data collection and presentation.
- The **server** performs the data processing and management.
- Drawback: introduces **latency and overhead**, especially when transferring data between client and server.

2. Three-Tier Architecture

- Separates the system into **three independent layers**:
 - **Presentation Tier** – User interface
 - **Application Tier** – Business logic
 - **Database Tier** – Storage and data management
- This separation increases **scalability, security, and performance**.
- Each tier can be scaled independently.

3. N-Tier Architecture

- Extension of three-tier architecture with **more layers** (e.g., middleware, API gateways, Caching layers).
- Further improves **modularity, flexibility, and load distribution**.
- Ideal for large enterprise applications with complex workflows.

4. Cluster Architecture

- Multiple machines (nodes) are connected together to behave like **one system**.
- Enables **parallel computing** to reduce latency.
- Great for large-scale computation, analytics, and big data processing.

5. Peer-to-Peer (P2P) Architecture

- All machines (nodes) have **equal responsibilities** in processing.
- No dedicated server (no central server); every node can act as both client and server.
- Encourages **cooperation among nodes** to distribute workload efficiently.

Examples

- **Cluster architecture** → enables **parallel computing** and **low latency** (e.g., data centers).

- **Peer-to-peer architecture** → Used in file-sharing networks or blockchain, where machines **cooperate equally** in processing tasks.

Why Data Processing Is Important?

- Processed data supports **analysis and decision-making** (raw data alone has little value—processing gives it structure, context, and meaning).
- Businesses use processed data to (extract value):
 - Compare sales figures
 - Detect trends
 - Improve operations
- With big data, moving data between computation and storage layers becomes challenging, making efficient data processing crucial. (Efficient data processing is essential to reduce delays, control costs, and ensure real-time insights.)

Hence,

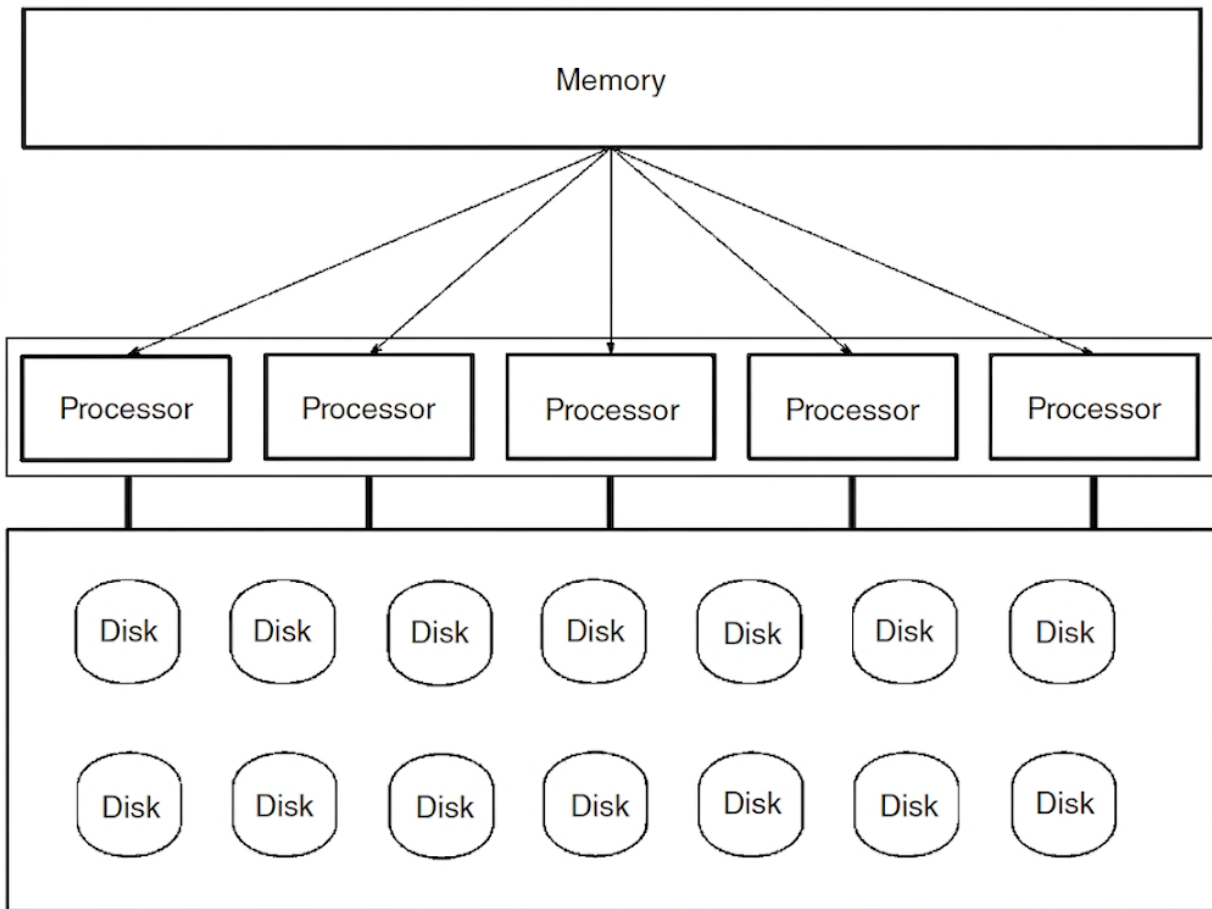
Data processing = turning raw, unorganized data into clear, meaningful information that people or systems can use.

3. Shared-Everything Architecture

Shared Everything Architecture is a system design in which **all processors have equal access to all major resources** in the system. This means every processor can use the same:

- **Shared memory**
- **Shared disk storage**
- **Shared I/O (*input/output*) channels**

This architecture is also known as **UMA (Uniform Memory Access)** because all processors access the same memory pool with **equal access time**.



Shared everything architecture.

Advantages

- Simple to design and implement
- Easy to manage since all data is in one place

Main Limitation

- **Not scalable**

When many processors try to access the same memory and bus, the system faces:

- Contention (multiple processors competing for the same memory and I/O)
- Bandwidth bottlenecks (All processors communicate over the same shared interconnect. As the system grows:
 - The bus becomes congested

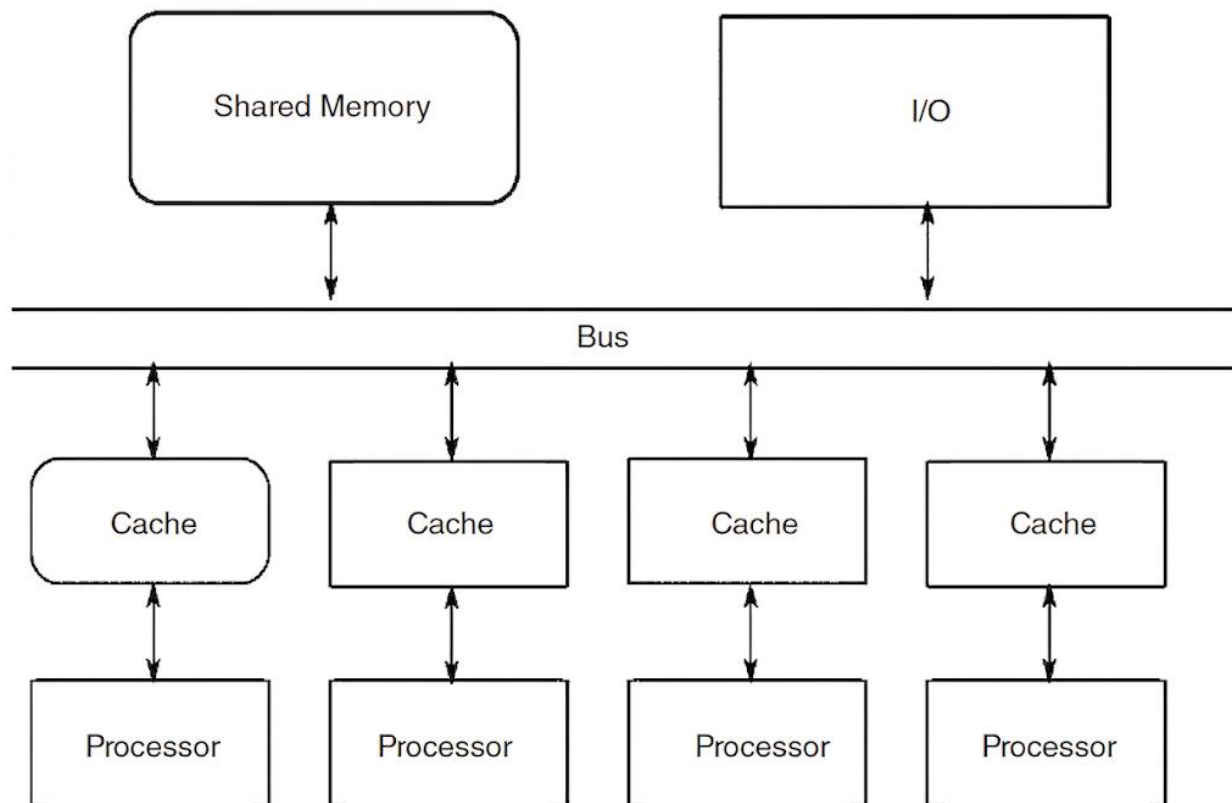
- Data transfer slows down
- Overall responsiveness decreases.)
- Performance degradation (Due to contention and bandwidth limitations, system performance:
 - Stops improving with additional processors
 - Eventually declines in heavy workloads.)

Therefore, this architecture is suitable only for **small-scale systems**.

Types of Shared Everything Architectures

Shared Everything Architectures can be implemented in two main forms:

A. Symmetric Multiprocessing (SMP)



Symmetric multiprocessing memory.

- The figure shows processors connected to shared memory through a bus

- SMP is limited by the number of processors that can efficiently share one bus

Characteristics

- All processors share **one single memory pool**
- Processors communicate via a **shared bus**
- Access time to memory is **uniform** → UMA (All processors experience **equal memory access time**.)
- **Local cache** for each processor to improve performance

Drawbacks

- As processors increase, the shared bus becomes a **bottleneck**
- Leads to bandwidth saturation (**bandwidth choking**).

B. Distributed Shared Memory (NUMA)

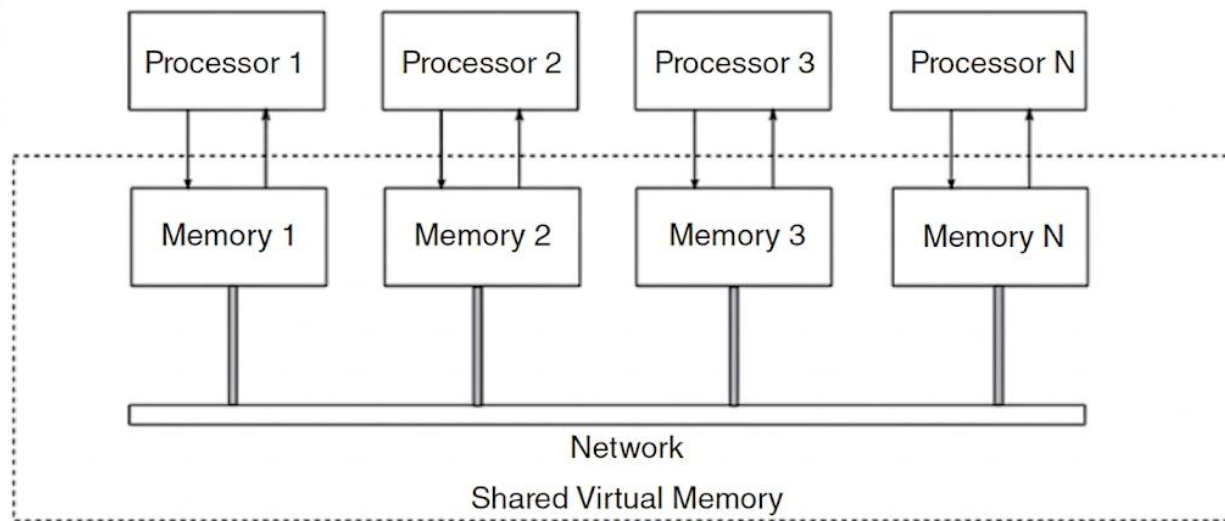
NUMA (Non-Uniform Memory Access) distributes memory across multiple processors, where each processor has its own local memory, but can also access the memory of other processors.

Characteristics

- Multiple processors, each with **its own local memory pool**
- Processors can access both local and remote memory
- Access speed depends on **distance** → known as **NUMA (Non-Uniform Memory Access)**
- Memory access time varies:
 - **Local memory** → fast
 - **Remote memory** → slower
- Eliminates the single-bus bottleneck found in SMP
- Still supports the “shared everything” model at the logical level

Advantages

- Higher scalability than SMP
- Reduces memory access contention
- Better suited for systems with many processors



Distributed shared memory.

The figure shows each processor connected to its own memory but still part of a shared address space

Key Differences: SMP vs. NUMA

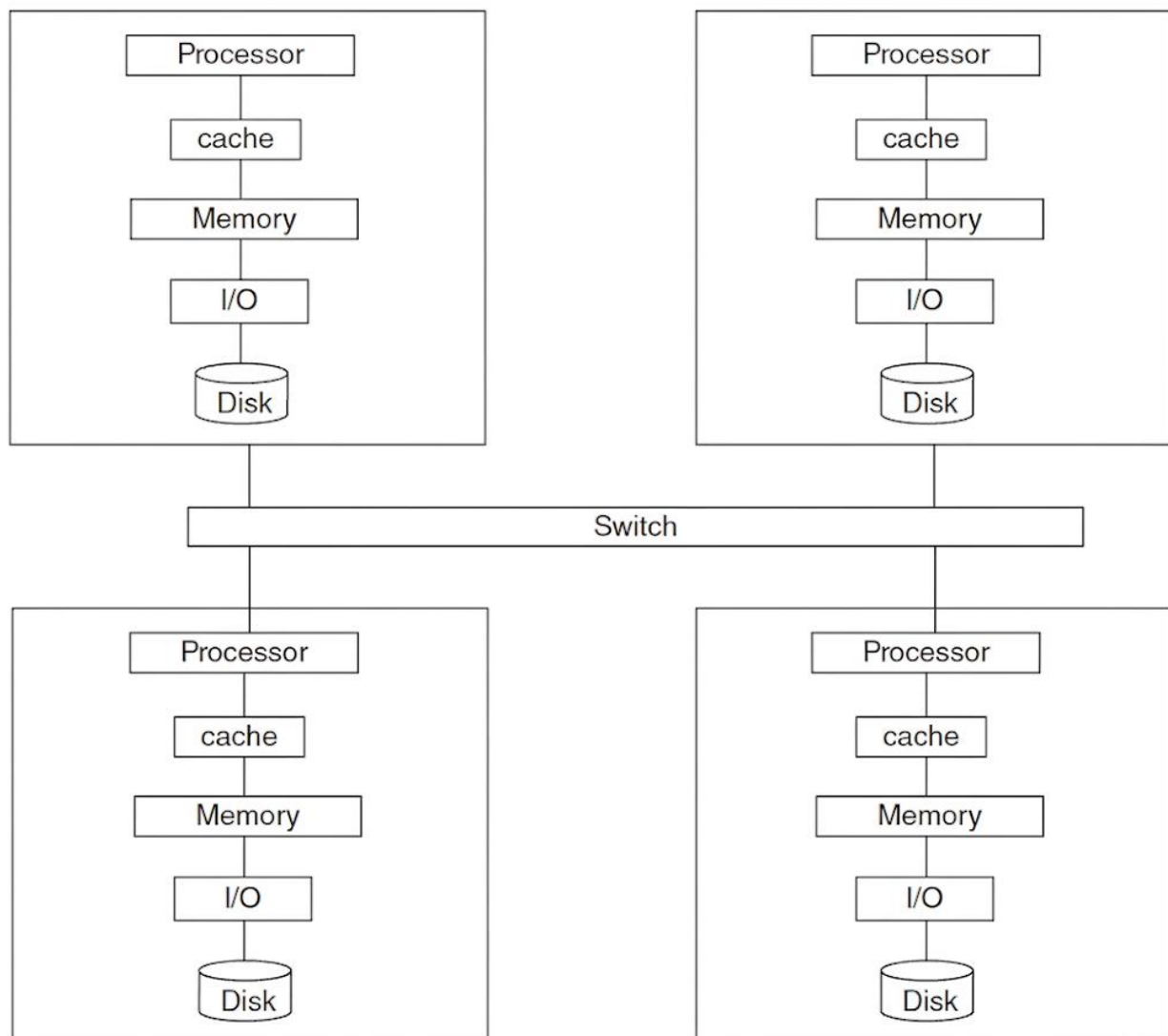
Feature	SMP (UMA)	Distributed Shared Memory (NUMA)
Memory Access	Uniform	Non-uniform
Memory Pools	One	Many
Scalability	Low	Higher
Bottleneck	Shared bus	Reduced contention
Architecture Type	Simple	More complex

Hence,

- **Shared Everything** = All processors share the same memory + same disk
- **SMP (UMA)** = One shared memory for all processors
- **NUMA** = Multiple memory pools; access time varies by distance

Shared Everything is easy to build but struggles with scalability, which is why modern large-scale systems use distributed or shared-nothing architectures instead.

4. Shared-Nothing Architecture



Shared-nothing architecture.

This is the most important architecture for Big Data.

Shared-Nothing Architecture is a type of **distributed system architecture** where multiple independent systems (called **nodes**) work together while **sharing no physical resources**.

Each node has its own:

- **Dedicated memory**
- **Local storage**
- **Independent disks**
- **Own processing power**

No two nodes share memory or storage, which completely eliminates resource contention between them.

Key Characteristics

- Each node is fully independent
- No shared memory, no shared disk, no shared bus
- Nodes communicate over a network
- Failure of one node does not affect others

This independence is what gives the architecture its name:

→ “**Shared-Nothing**” = nothing is shared among nodes.

Why It Scales Extremely Well

Because nodes do not compete for shared resources, the system can grow by simply **adding more nodes**.

This provides:

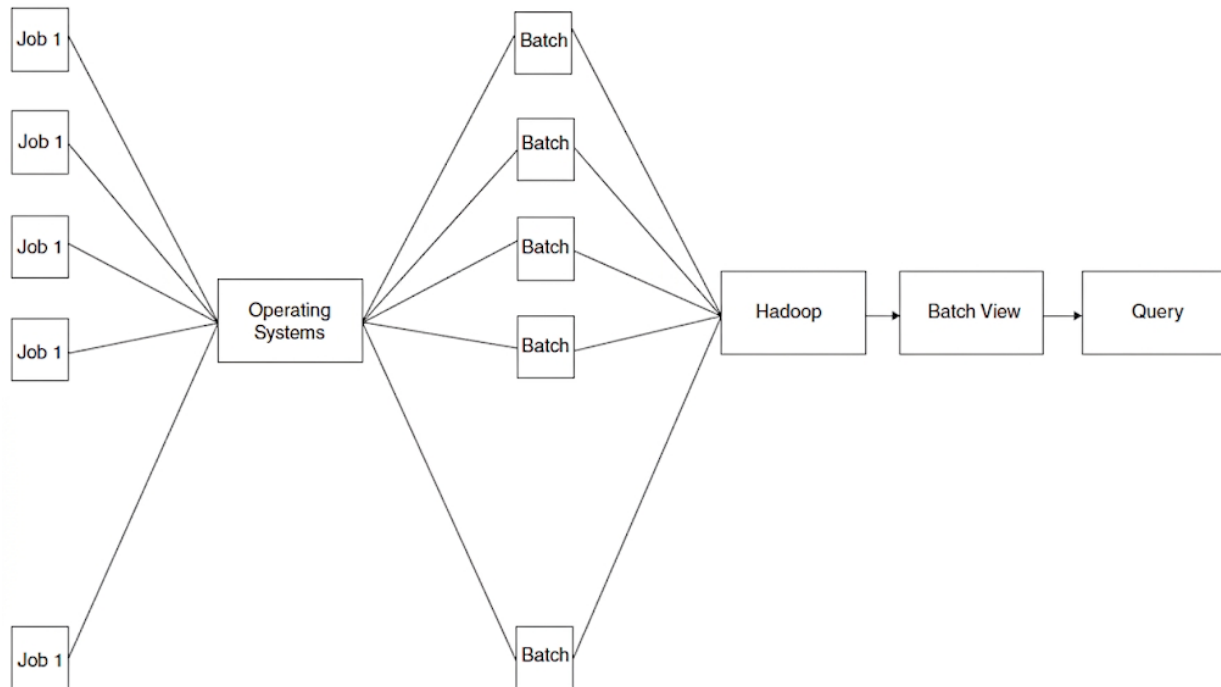
- **Near-infinite scalability**
- **High fault tolerance**
- **Excellent performance for distributed workloads**

Where It Is Used

The high scalability and reliability make Shared-Nothing Architecture ideal for:

- **Internet-scale applications**
- **Web servers and cloud services**
- **Distributed databases** (e.g., Cassandra, Bigtable)
- **Large-scale websites**
- **Modern data processing systems** (e.g., Hadoop, Spark)

5. Batch Processing



Batch processing.

Batch processing is a data processing technique in which a series of related jobs are collected together and executed **sequentially or in parallel**. After all the individual jobs finish, their outputs are combined to produce a **final result**.

This approach is ideal when:

- Large volumes of data must be processed
- Immediate response time is **not** required
- Tasks are repetitive or scheduled

How Batch Processing Works

Batch processing is performed by **collecting data in batches** over a period of time and then executing a processing job on the whole batch.

The output of this batch may also serve as **input** to another job or process.

It is especially suitable for applications that deal with:

- **Terabytes or petabytes** of data
- Situations where real-time processing is unnecessary

Common Use Cases

Batch processing is widely used in many systems, especially where data is accumulated over time and then analyzed or processed together.

Examples:

- **Log analysis:** logs are collected over hours or days, then processed
- **Payroll systems:** salaries are computed monthly in batches
- **Billing systems:** invoices are generated periodically
- **Data warehouses:** bulk data is loaded and transformed for analytics

Batch Processing in Big Data

Technologies like **Hadoop MapReduce** are commonly used to implement batch jobs in big data platforms.

Objectives of batch jobs in big data:

- Aggregate large amounts of data
- Store results for later analysis
- Schedule processing tasks efficiently

Early big data systems focused heavily on batch techniques because they:

- Can extract data at scale
- Run jobs later during off-peak hours
- Are **cost-effective** and **easy to execute** compared to real-time streaming systems

6. Real-Time Data Processing

Real-time data processing refers to processing data continuously **as it arrives**, producing results almost instantly. Instead of waiting to collect data in batches, the system handles the incoming data stream in **real time**, allowing immediate analysis and action.

How Real-Time Processing Works

- Data arrives continuously (as a stream).
- The data is processed **in memory (RAM)** to minimize delay.
- After the real-time computation is completed, the processed data may later be stored on disk.
- This approach is essential whenever decisions must be made **within milliseconds or seconds**.

Because the system analyzes data while it is streaming, storing it first to disk would be too slow—hence, the need for **in-memory processing**.

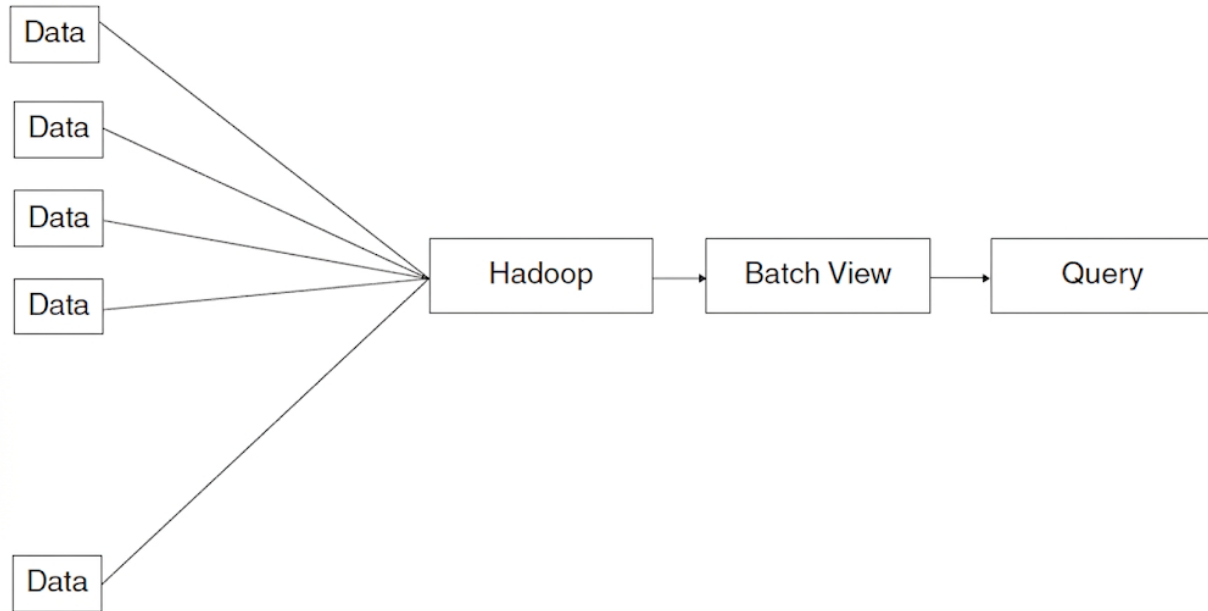
Examples of Real-Time Data Processing

Real-time processing is required in scenarios where immediate responses are critical, such as:

- **Online transactions** (e.g., credit card payments)
- **ATM withdrawals**
- **Point-of-sale (POS) systems**
- **Fraud detection systems** (identify suspicious activity instantly)

- **Monitoring systems** (IoT sensors, stock market feeds, etc.)

These systems must react **as soon as the event occurs**, making real-time processing necessary.



Real-time processing.

Although Hadoop is traditionally batch-based, some architectures combine streaming + batch to provide **near real-time insights**.

This approach is sometimes part of a **Lambda Architecture** (stream layer + batch layer).

Real-Time and Batch Systems

The table shows commonly used processing systems and their types:

Streaming Systems

Solution	Developer	Type	Description
Storm	Twitter	Streaming	Framework for real-time stream processing
S4	Yahoo	Streaming	Distributed stream computing platform
MillWheel	Google	Streaming	Fault-tolerant streaming computation framework

Batch Systems

Solution	Developer	Type	Description
Hadoop	Apache	Batch	First open-source implementation of MapReduce
Disco	Nokia	Batch	MapReduce framework by Nokia

Streaming systems (Storm, S4, MillWheel) are designed for **low-latency, real-time** data processing.

Batch systems (Hadoop, Disco) handle large datasets efficiently but not instantly.

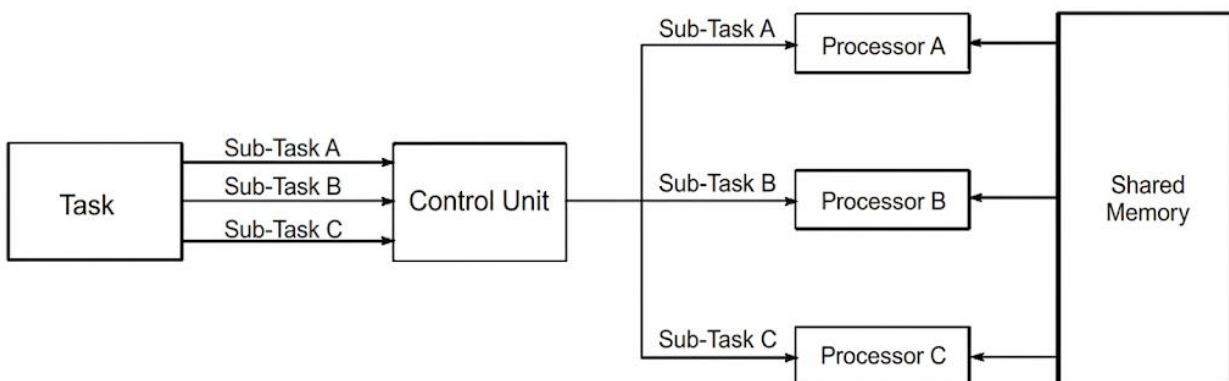
Why Real-Time Processing Matters?

Real-time data processing enables organizations to:

- Respond quickly to events
- Detect fraud immediately
- Monitor transactions and systems live
- Support mission-critical operations where delays are unacceptable

It provides **low latency**, making near-instant decisions possible.

7. Parallel Computing vs Distributed Computing



Parallel computing.

A. Parallel Computing

- One machine
- Multiple processors
- Task is split into sub-tasks
- Sub-tasks run simultaneously

Think of it like a multi-core CPU doing different pieces of work at the same time.

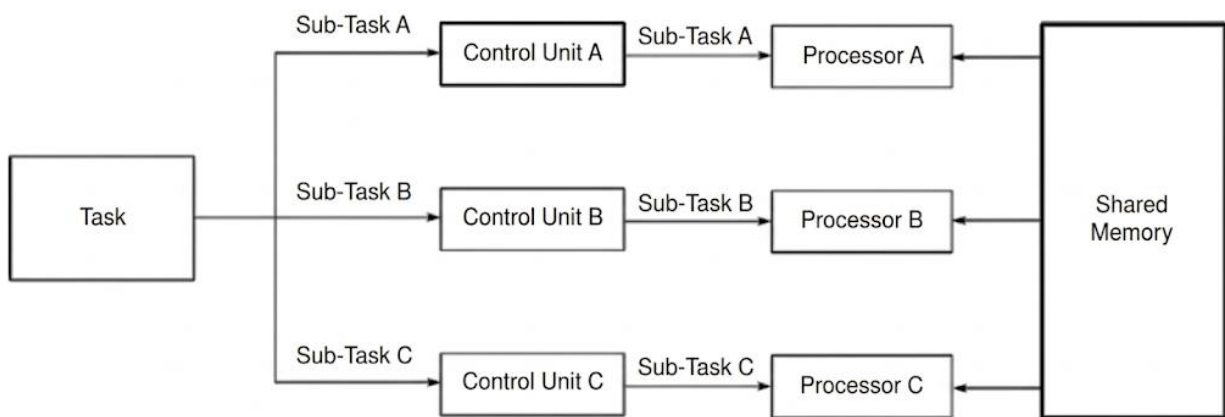
B. Distributed Computing

Distributed computing is similar to parallel computing in that a large task is divided into multiple smaller **subtasks**.

However, the key difference is:

- **In distributed computing, the subtasks are executed on different machines that are networked together.**

These machines form a **cluster**, and each machine has its own processor, memory, and resources.



Distributed computing.

Advantages of Distributed Computing

- **Scalability:** You can add more machines to increase computing power.
- **Fault tolerance:** If one machine fails, others can continue working.
- **Resource sharing:** Each machine contributes its own CPU, memory, and storage.

- **Cost efficiency:** Uses clusters of commodity hardware instead of expensive supercomputers.

Examples of Distributed Computing Platforms

- Hadoop Distributed File System (HDFS)
- Apache Spark
- Kubernetes-based clusters
- Cloud computing platforms (AWS, Azure, Google Cloud)

Key Difference From Parallel Computing

Feature	Parallel Computing	Distributed Computing
Execution environment	Single machine	Multiple machines
Memory	Shared memory	Each machine has its own memory
Communication	Fast (internal bus)	Slower (network-based)
Scalability	Limited	Very high

8. Virtualization in Big Data

What is Data Virtualization?

Data virtualization is a technology that allows you to access data from many different sources **as if it were a single unified source**.

Instead of physically moving or copying data into one place, virtualization **creates a single logical view** of the data.

Key Idea

You can access data from multiple heterogeneous systems (databases, files, cloud storage, etc.) **without moving it**.

Main Purpose

- Provide *one point of access* to data that actually lives in many different sources
- Make data integration easier in big data systems
- Reduce data movement and improve performance

Why Is Virtualization Important in Big Data?

Big data environments contain massive, distributed datasets. Physically copying and integrating all this data is:

- Slow
- Expensive
- Hard to scale
- Redundant

Virtualization solves these issues by giving you **virtual access** instead of requiring physical consolidation.

Virtualization (General Concept)

Virtualization is a technique where physical computer resources (hardware + software) are used to create **virtual versions** of those resources.

Examples of virtual resources:

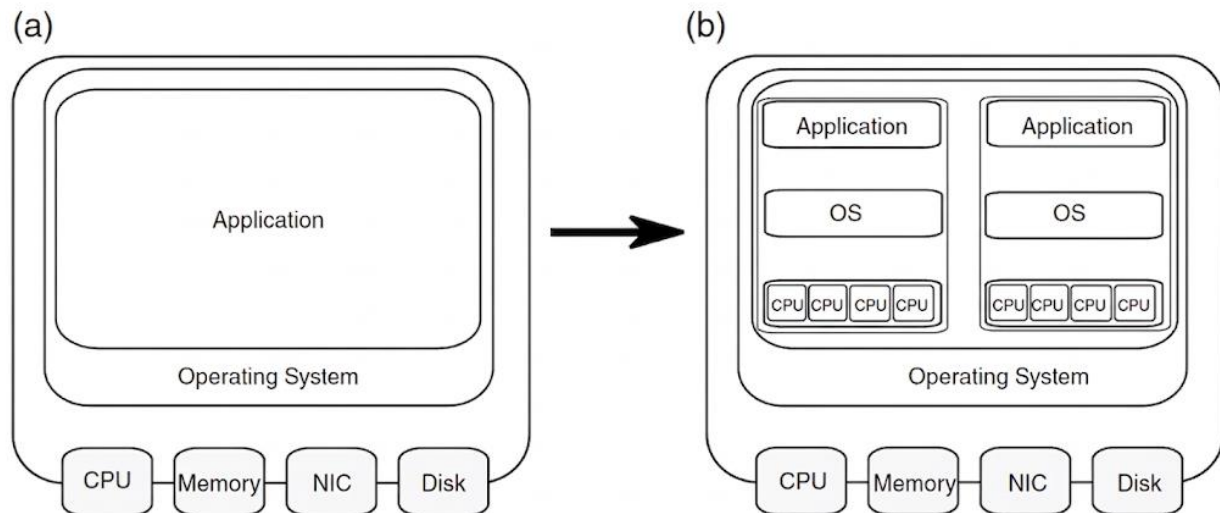
- Virtual CPUs
- Virtual memory
- Virtual network interface cards (NICs)
- Virtual machines (VMs)

The goal is to:

- Use hardware more efficiently
- Reduce cost
- Improve flexibility

- Support cloud computing

System Architecture Before and After Virtualization



System architecture before and after virtualization.

(a) Before Virtualization — Traditional System

- One operating system
- One application running directly on physical hardware

Physical resources (CPU, memory, NIC, disk) are **tied directly** to a single OS and application.

Limitations:

- Poor resource utilization
- Each application needs its own dedicated hardware
- Expensive and not scalable

(b) After Virtualization — Virtualized System

A **virtualization layer** (hypervisor or VMM) sits between the hardware and multiple virtual machines (VMs).

Each VM has:

- Its own OS

- Its own applications
- Its own virtual CPU, memory, NIC, disk

This allows **multiple applications** to run independently on the same physical hardware.

What is a Hypervisor / VMM?

A **Virtual Machine Monitor (VMM)** or **hypervisor** is the software that:

- Creates virtual machines
- Manages virtual CPUs, memory, and I/O
- Allocates physical resources to VMs

VMs run **above** the hypervisor

The hypervisor runs **above** the host hardware.

Attributes of Virtualization

1. **Encapsulation** → VM stored as a single file (easy to move)
2. **Partitioning** → splitting hardware into multiple VMs
3. **Isolation** → problems in one VM don't affect others

These attributes define how virtual machines (VMs) operate and how physical resources are abstracted and shared.

Encapsulation

Encapsulation means that a virtual machine (VM) is packaged as a **single software entity** that behaves like a complete physical machine.

Key Points

- A VM is a **software-based representation** of a physical machine.
- The entire VM—including its OS, applications, and configuration—is stored as a **single file** or a small set of files.
- Because the VM is encapsulated, it can be:
 - Easily identified

- Moved or copied
- Backed up or restored

Why Encapsulation Matters?

- Each application can run inside its **own dedicated VM**.
- Applications are isolated from one another, preventing conflicts.
- The VM appears as a complete standalone machine to the operating system or application.

Partitioning

Partitioning divides the physical hardware of a host machine into **multiple logical units**, allowing multiple VMs to run on the same physical system.

Key Points

- CPU, memory, storage, and network resources can be divided up among several VMs.
- Each VM receives its **own operating system**, independent of others.
- Multiple VMs run simultaneously on single hardware.

Why Partitioning Matters?

- Improves hardware utilization.
- Allows many applications or services to run on one physical machine.
- Enables consolidation of servers (reducing cost).

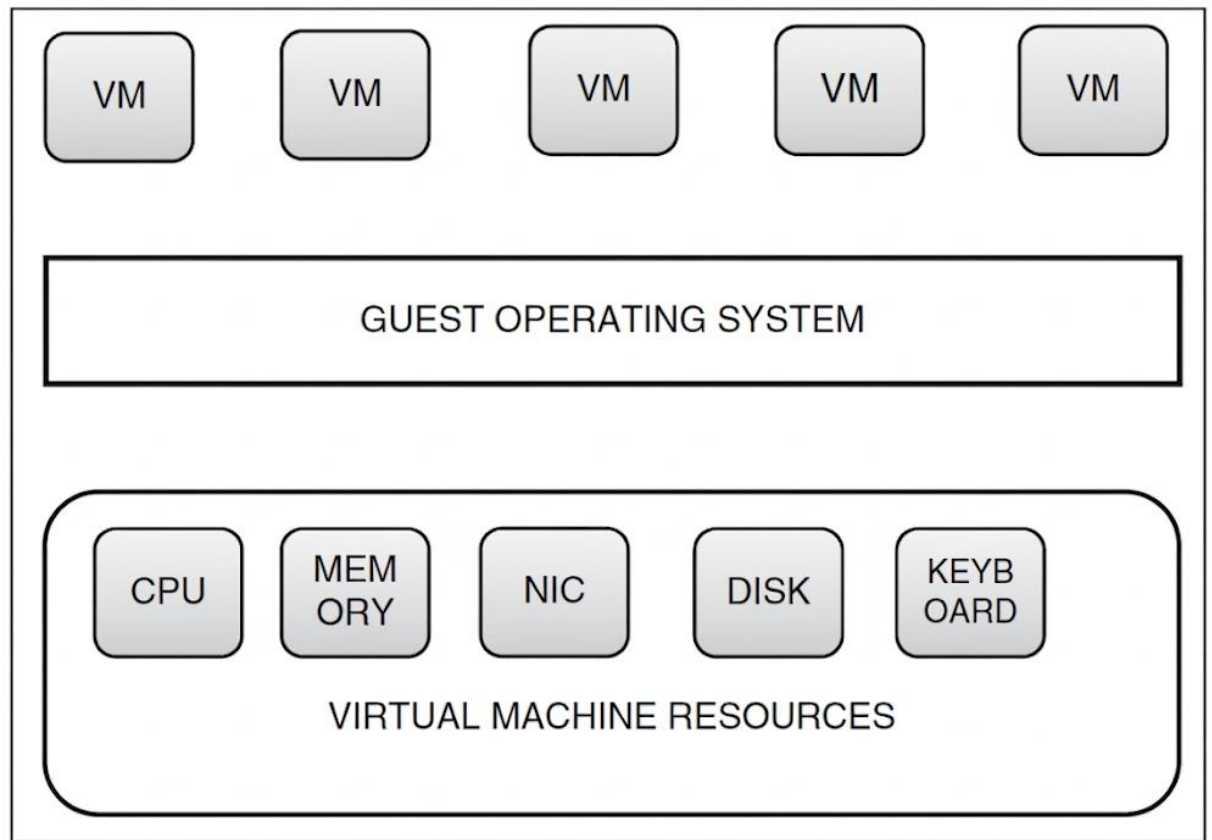
Isolation

Isolation ensures that each VM is fully separated from other VMs and from the host machine.

Key Points

- If one VM crashes, other VMs remain unaffected.
- VM activities cannot interfere with one another.
- Each VM is protected from faults or security risks originating from another VM.

- The **following diagram** shows how each VM is isolated from physical hardware and from other VMs



Isolation.

Why Isolation Matters?

- Strengthens system reliability.
- Essential for multi-tenant environments (cloud systems).
- Enhances security and stability.

Big Data Server Virtualization

Virtualization is widely used in **big data environments** to handle large volumes of data efficiently.

How it Works

- A software layer (hypervisor) sits between hardware and VMs.

- Multiple operating systems can run on the same physical server.
- Each OS functions independently—unaware of other OSs or VMs.

In Server Virtualization

- A physical server is partitioned into **multiple VMs**.
- CPU, memory, I/O devices, and storage are **virtualized**.
- Each VM can run separate applications (e.g., analytics, databases, monitoring services).

Why Virtualization Is Important in Big Data?

- Big data analysis needs flexibility because the **data volume is unpredictable**.
- Virtualization provides:
 - Dynamic resource allocation
 - Scalability
 - Ability to handle sudden increases in workload
- Multiple VMs can work together to process massive datasets.

Real-Time and Big Data Needs

- In real-time processing, data flow is constant and unpredictable.
- Virtualization provides the ability to:
 - Scale resources quickly
 - Handle bursts of data
 - Ensure reliability under heavy workloads

9. Introduction to Cloud Computing

Big data and cloud computing are two major technologies that are transforming how organizations work today.

Big data helps organizations collect and analyze huge amounts of information from sources like social media, sensors, websites, and online transactions. This creates many opportunities, but it is also **complex, expensive, and time-consuming** to manage. Traditional computing systems cannot easily handle the size and speed of big data.

Cloud computing solves many of these challenges. It provides computing resources — like storage, servers, and processing power — **on demand**, so organizations only pay for what they use. This makes cloud computing flexible, cost-effective, and scalable.

Cloud computing = delivering IT resources on demand.

Cloud computing improves big data processing by:

- Allowing organizations to store much more data at lower cost (Cloud offers **scalable storage**.)
- Reducing the need to buy expensive hardware (You pay only for what you use (**pay-as-you-go**).)
- Automatically adjusting resources based on workload
- Running multiple applications on the same physical machine using virtualization

Cloud systems from companies like Amazon, Microsoft, and Google offer powerful tools for handling large-scale data.

Technologies such as **NoSQL databases** and **Hadoop** were built to solve big data problems. Hadoop uses distributed storage and parallel processing, allowing a cluster of computers to work together and share the workload.

A very important concept in big data is **data locality**. Because big data is so large, it is not practical to move data across the network for processing. Instead, cloud systems move the **computation** closer to where the data is stored. This greatly improves processing speed.

Although cloud computing has many advantages, it also brings concerns about **security and privacy**. Since data is stored on the cloud provider's servers, organizations worry about whether sensitive information — like credit card data or government records — is safe. A security breach can have serious consequences for users, companies, and cloud providers.

10. Cloud Computing Types

Cloud computing makes it easy to share and access computing resources (like storage, servers, and applications) through public or private networks. Users do not have to manage hardware or update software—the cloud provider handles everything.

Cloud computing is generally divided into **three types**:

1. **Public Cloud**
2. **Private Cloud**
3. **Hybrid Cloud**

A. Public Cloud

In a **public cloud**, services are delivered over the Internet by third-party providers (e.g., Amazon AWS, Microsoft Azure, IBM Cloud).

Key Features

- Accessible to anyone over the Internet
- Pay-as-you-go pricing (only pay for what you use)
- Provider handles updates, maintenance, hardware, and software
- Highly scalable with large resource pools

Advantages

- Very low cost
- No maintenance for the client
- Easy to scale up or down based on demand

Limitations

- Some organizations avoid public clouds due to concerns about security, data control, and system configuration.

B. Private Cloud

A **private cloud** is dedicated to a single organization. It is also called an **internal** or **corporate** cloud.

Key Features

- Owned or used exclusively by one organization
- Better security and privacy because everything stays within the company
- Can be hosted:
 - **On-premise** (inside the organization's own data center)
 - **Externally** (managed by a provider but used only by one company)

Advantages

- High security and full control over data
- Suitable for businesses with strict compliance requirements

Limitations

- More expensive than public cloud
- Limited scalability compared to public cloud
- Requires more management effort if hosted internally

C. Hybrid Cloud

A **hybrid cloud** combines both **public** and **private** cloud environments.

Key Features

- Some resources are managed in-house (private cloud)
- Some resources come from public cloud providers
- Workloads can move between public and private clouds

Advantages

- Offers the flexibility of public cloud plus the security of private cloud

- Useful during peak workloads or maintenance periods
- Supports on-demand scalability

Common Use Cases

- When organizations need strong security for some applications but also want the cost benefits of public cloud
- When workloads change frequently and require flexible scaling

11. Cloud Service Models

Cloud computing delivers computing capabilities through **three layered service models**:

1. **Software as a Service (SaaS)**
2. **Platform as a Service (PaaS)**
3. **Infrastructure as a Service (IaaS)**

These services form a layered architecture, where each layer serves different users — from system administrators to application developers to end users.

1. Software as a Service (SaaS)

SaaS provides **ready-to-use software applications** to end users over the internet

Key Points

- Provides **complete software applications** to users.
- Accessed via the internet with **no local installation required**.
- Pricing models:
 - Subscription-based
 - Pay-as-you-go (on-demand)
- Software and data are:
 - Centrally managed
 - Securely shared among multiple users
- Designed primarily for:
 - **End users**

Examples

- Salesforce
- Microsoft Office 365
- Oracle Cloud Apps
- IBM Cloud Software

Who Uses SaaS?

- **End users** (e.g., employees, customers)

2. Platform as a Service (PaaS)

PaaS provides a platform for developers to build, test, and deploy applications.

Key Points

- Enables users to:
 - Develop (using tools provided by the cloud platform)
 - Run
 - Maintain applications
- Accessed through a **web browser**.
- Pricing is based on **pay-per-use**. (Charged on a pay-per-use basis.)
- Users do not manage:
 - Servers
 - Storage
 - Networking
 - Operating Systems.

Examples

- Amazon Web Services (AWS Elastic Beanstalk)
- Google App Engine

- AppFog
- Heroku

Who Uses PaaS?

Intended mainly for

- **Application developers**

3. Infrastructure as a Service (IaaS)

Key Points

- Provides **basic virtualized computing resources**, including:
 - Servers
 - Networking
 - Data center space
 - Storage
- Resources are offered:
 - On-demand
 - Self-service
 - Pay-per-use
- Eliminates the need for purchasing physical infrastructure. (Clients rent computing resources instead of buying physical hardware.)
- Resources can be offered as:
 - Dedicated (**physical**)
 - Shared (**virtual**)
- Highly scalable and flexible.

Examples

- Amazon EC2

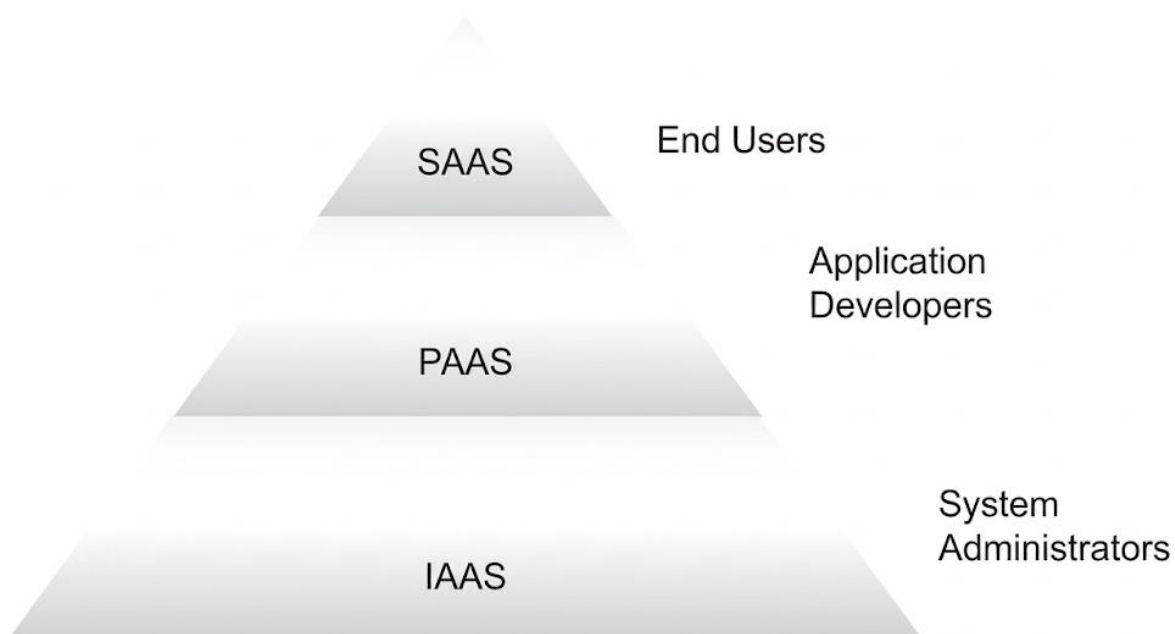
- Google Compute Engine
- IBM Cloud Infrastructure
- Oracle Cloud Infrastructure
- Fujitsu
- Hewlett-Packard

Who Uses IaaS?

- **System administrators**
- Organizations needing full control over system infrastructure

Service Layer Diagram

The service-oriented architecture forms a stacked hierarchy:



Service-oriented architecture.

- **SaaS** → Used by end users
- **PaaS** → Used by application developers (builds on IaaS (development platform).)
- **IaaS** → Used by system administrators (forms the base layer (infrastructure).)

Each layer builds on the one below it, offering increasing abstraction and ease of use.

12. Cloud Storage & The Google File System (GFS)

Cloud Storage Requirements

- Designed to handle **exponentially growing storage demands** driven by big data.
- Must be:
 - Highly scalable
 - Highly reliable and highly available
 - Cost-effective
 - Decentralized
 - Fault-tolerant
- Built using:
 - **Distributed file systems** for large-scale file storage
 - **Distributed NoSQL databases** for data processing and analytics

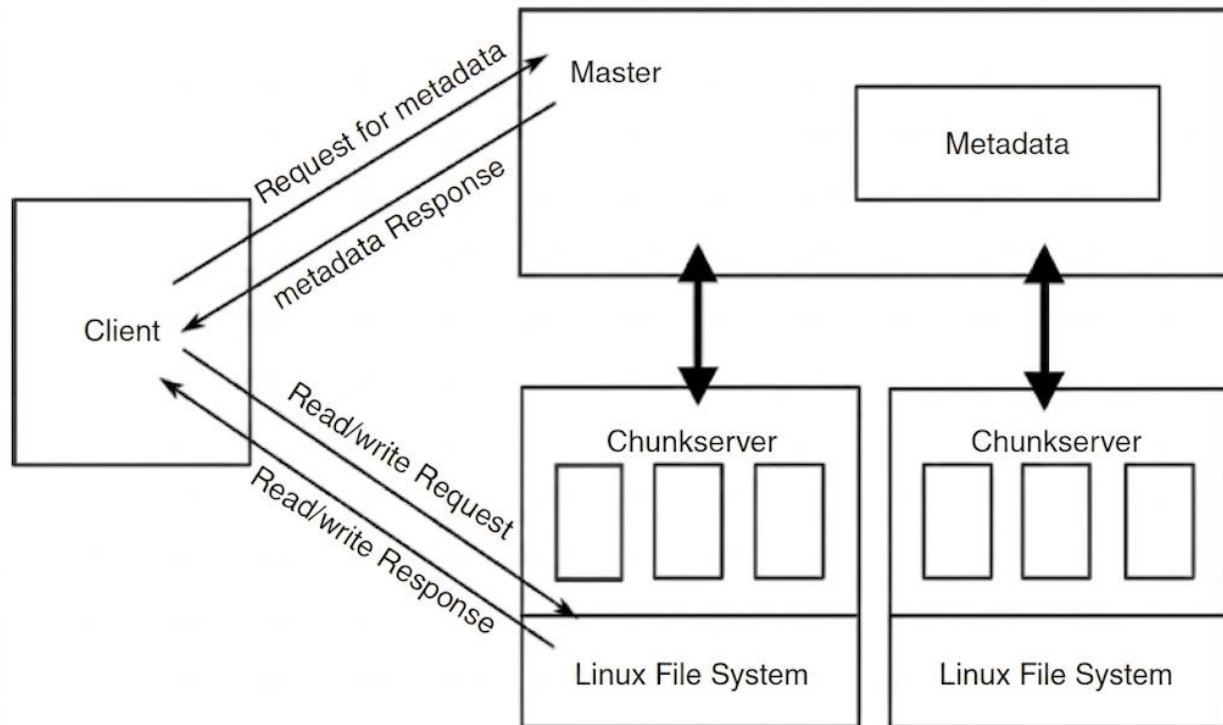
Evolution of Cloud Storage Technologies

- Google developed:
 - **Google File System (GFS)** for distributed storage
 - **MapReduce** for scalable data processing
 - **Bigtable** for high-performance distributed database management
- Since Google's systems were not open-source:
 - **Hadoop** was developed by Yahoo to implement MapReduce
 - **HDFS** (Hadoop Distributed File System) was designed similar to GFS
 - **HBase** was developed as an open-source alternative to Bigtable

- Hadoop and HBase are now:
 - Managed by Apache
 - Widely adopted in industry

Architecture of Google File System (GFS)

General Architecture



Google File System architecture.

- Follows a **master–chunkserver architecture**
- A GFS cluster consists of:
 - One **Master server**
 - Multiple **Chunkservers**
- Files are:
 - Divided into fixed-size **chunks (default: 64 MB)**
 - Stored as Linux files on chunkserver disks

- Each chunk:
 - Is identified by a **64-bit unique chunk handle**
 - Is assigned by the master at creation time
 - Has **three replicas by default** for reliability

Role of the Master

- Maintains **all metadata**, including:
 - File-to-chunk mappings
 - Chunk replica locations
 - Namespace and access control
- Metadata size is very small:
 - Less than 64 bytes per 64 MB chunk
- Additional responsibilities:
 - Chunk management
 - Removing stale replicas
 - Monitoring chunkserver health
- Communicates with chunkservers using:
 - **Heartbeat messages**
 - Used for instruction delivery and status reporting

Role of the Client

- Acts as an intermediary between application and GFS
- Responsibilities:
 - Contacts master to obtain metadata
 - Identifies which chunkserver to communicate with
- Performs **all actual read/write operations directly with chunkservers**

- Reduces master workload by avoiding data transfer through master

Chunk Characteristics

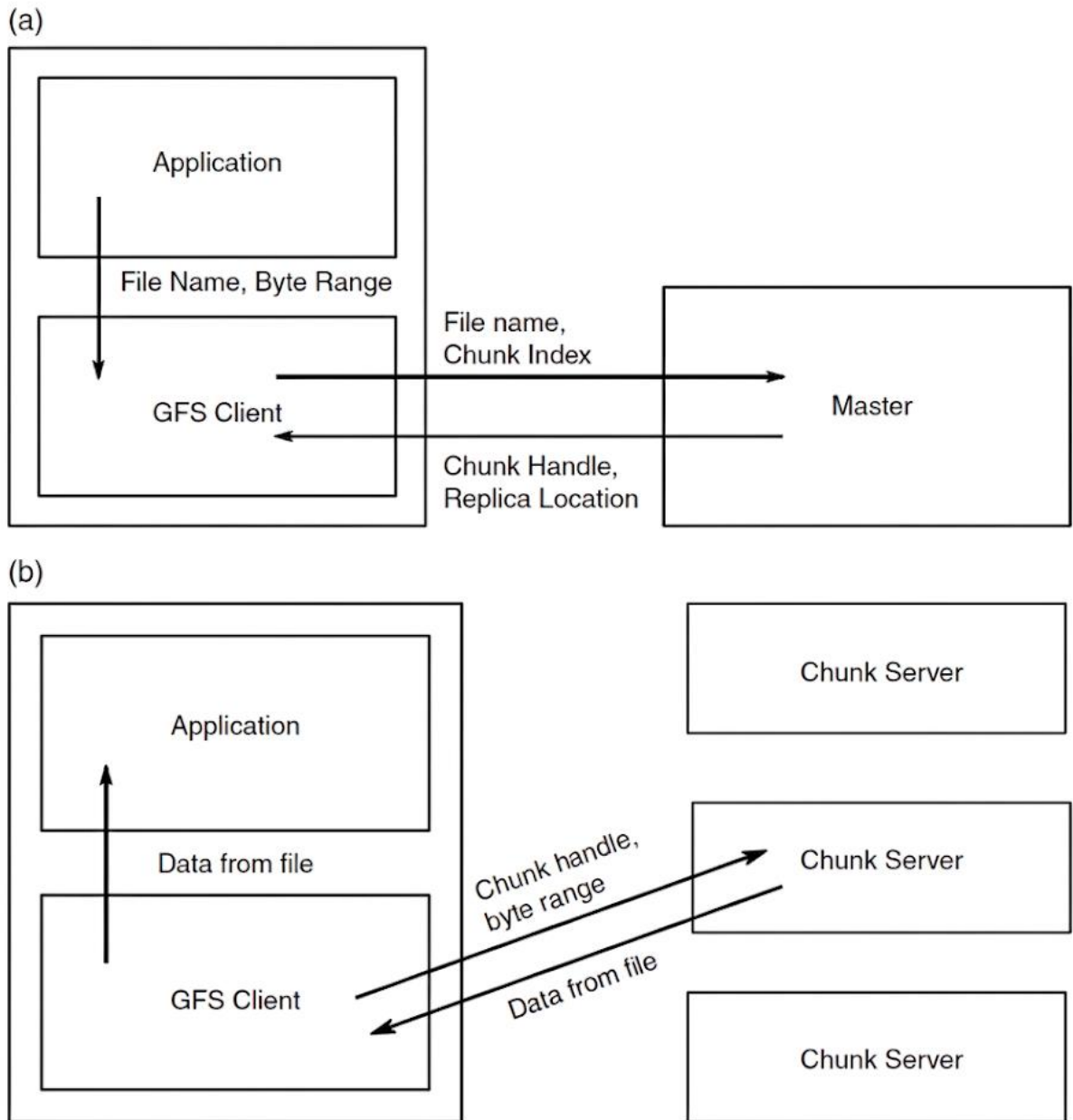
- Chunks are similar to file system blocks but:
 - Much larger in size
- Comparison:
 - Traditional blocks → KBs
 - GFS chunks → **64 MB**
- Advantages of large chunk size:
 - Reduces metadata size
 - Reduces number of master accesses
 - Improves scalability
- Example benefit:
 - Larger chunks → fewer metadata entries → better efficiency

Read Algorithm in GFS

Read Operation Steps

1. Application initiates a read request.
2. GFS client:
 - Translates filename and byte range
 - Sends filename and chunk index to master.
3. Master returns:
 - Chunk handle
 - Replica locations.
4. Client selects a chunkserver replica.
5. Client sends read request directly to the chunkserver.

6. Chunkserver sends requested data back to client.
7. Client delivers data to the application.



Read algorithm: (a) The first three steps. (b) The last three steps.

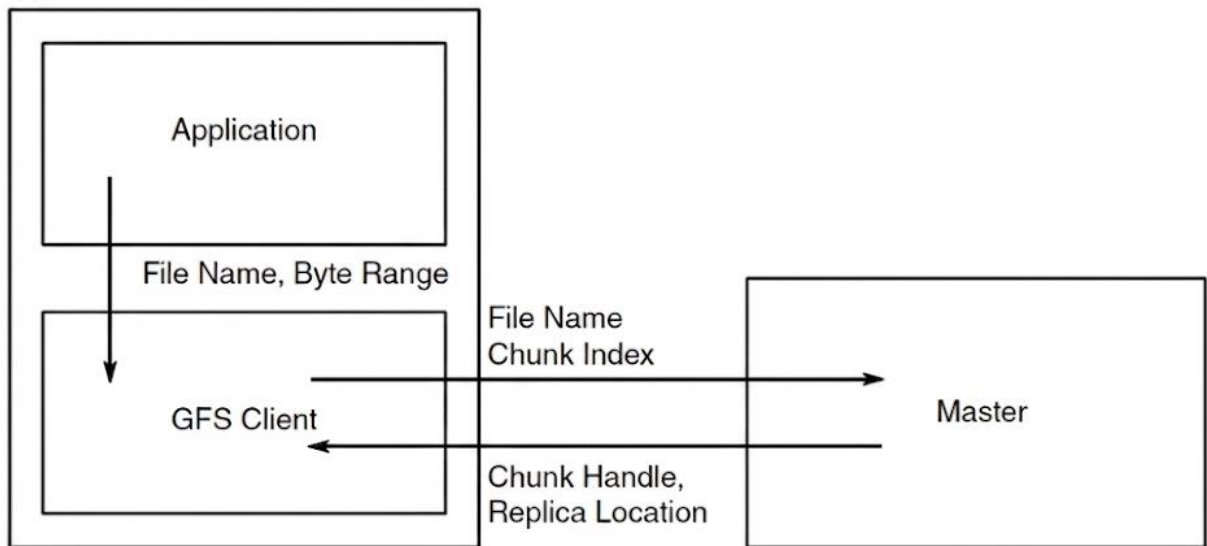
Write Algorithm in GFS

Write Operation Steps

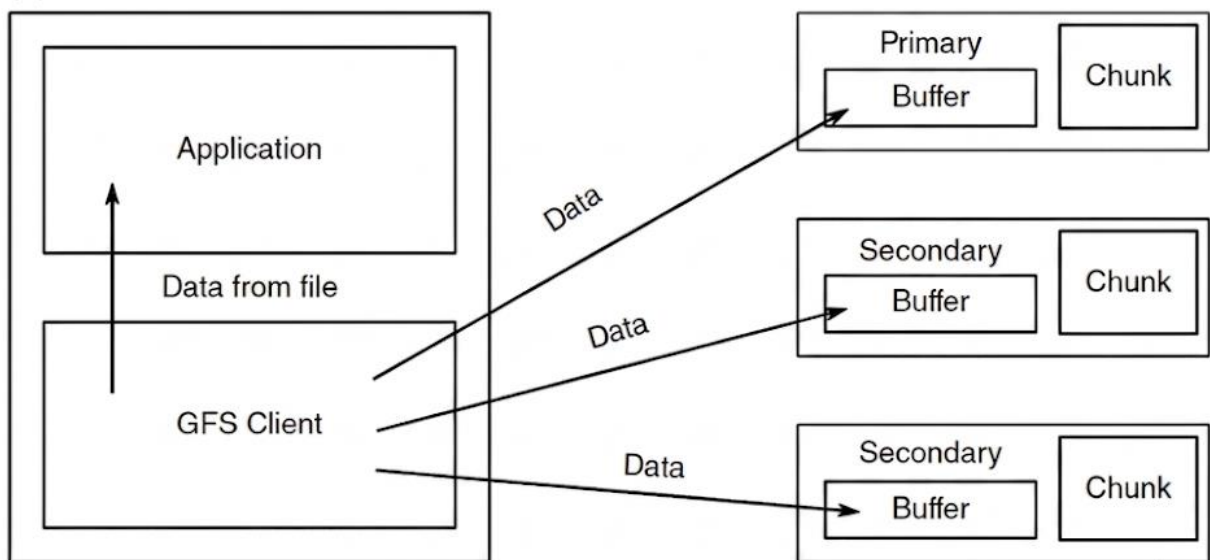
1. Application initiates a write request.

2. GFS client sends filename and data metadata to master.
3. Master responds with:
 - Chunk handle
 - Primary and secondary replica locations.
4. Client pushes data to all replicas' buffers.
5. Client sends write command to the **primary replica**.
6. Primary determines **serial order** of writes.
7. Primary forwards ordered write commands to secondary replicas.
8. Secondary replicas acknowledge completion to primary.
9. Primary sends success response to client.

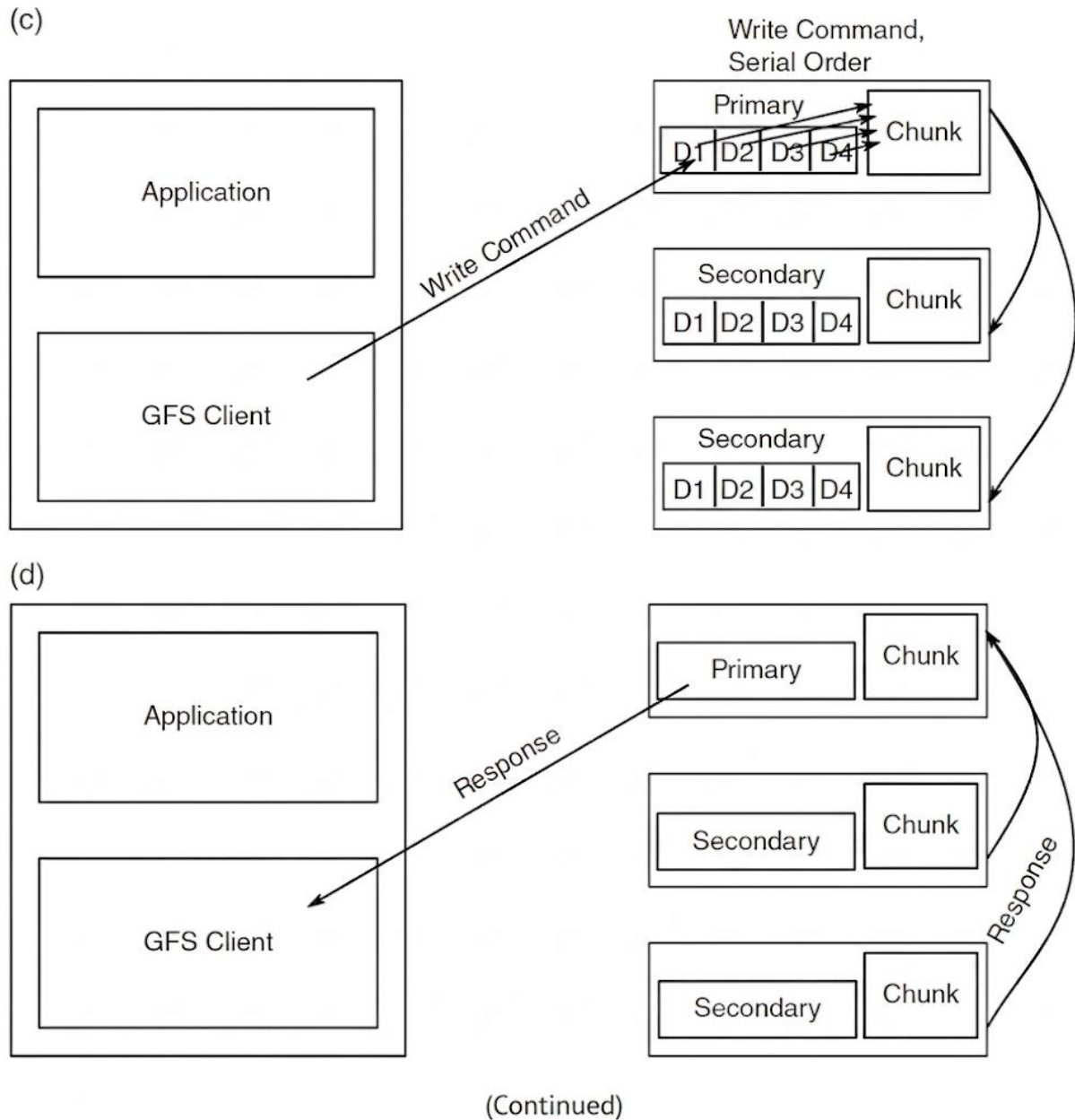
(a)



(b)



Write algorithm: (a) The first three steps. (b) Steps 4 and 5. (c) Steps 6 and 7
(d) Steps 8 and 9.



Key Architectural Advantages of GFS

- Separation of:
 - Metadata management
 - Data transfer
- High reliability via:
 - Replication

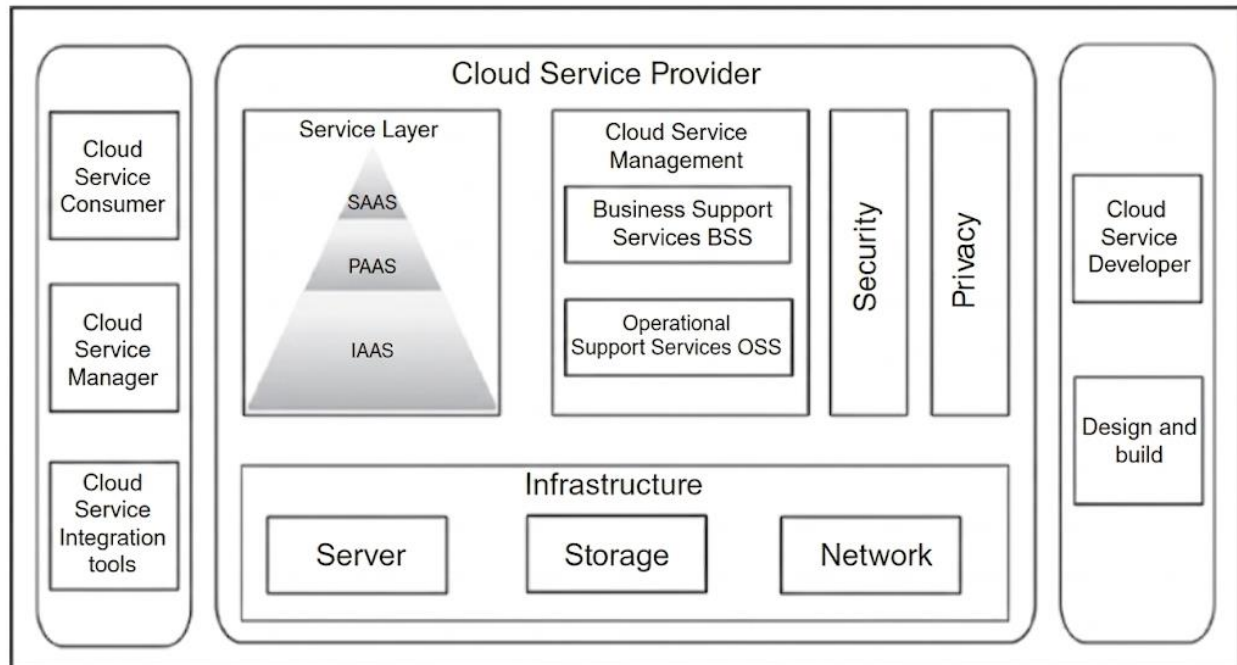
- Fault detection through heartbeat messages
- High scalability through:
 - Large chunk sizes
 - Distributed storage
- Reduced bottleneck at master by:
 - Direct client–chunkserver communication

Cloud Architecture

Cloud architecture provides a scalable, flexible, and cost-effective computing environment, but it is challenged by concerns related to security, performance, portability, and interoperability.

General Cloud Architecture

- Cloud architecture consists of:
 - **Front end**
 - **Back end**
- Front end and back end are connected through a **network**, usually the **Internet**.
- The **network** is the key enabling component of cloud computing.



Cloud architecture.

Front End (Client Side)

- Represents the **client infrastructure**.
- Consists of:
 - Client applications
 - Interfaces that access cloud services
- Used by clients to request and consume cloud resources and services.

Back End (Cloud Infrastructure)

- Represents the **cloud service provider side**.
- Includes core resources:
 - Servers
 - Storage
 - Networking
- Responsible for:

- Security
- Privacy
- Protocol handling
- Traffic control
- Uses **middleware** to allow communication among connected devices.

Cloud Servers

- Cloud servers are **virtual servers**.
- Function similarly to physical servers, but:
 - Operate through virtualization
 - Offer higher flexibility
- Responsible for:
 - Resource allocation and de-allocation
 - Security enforcement
- Clients are charged based on:
 - Hours of resource usage

Hosting Models

- **Shared hosting**
 - Resources are shared among multiple clients
 - Cost-effective
 - Cannot handle heavy traffic efficiently
- **Dedicated hosting**
 - Entire server dedicated to one client
 - Better performance and reliability
 - Higher cost

- Clients may use:
 - One or multiple dedicated servers
- Resources can be:
 - Scaled up or down according to demand

Benefits of Cloud Architecture

- Cost effectiveness
- Ease of setup
- Reliability
- Flexibility
- Scalability

Cloud Storage Reliability

- Data is stored with **multiple replicas**.
- If one storage resource fails:
 - Data is recovered from replica storage
- Ensures:
 - Fault tolerance
 - High availability

Cloud Service Models in Architecture

- **Infrastructure as a Service (IaaS)**
 - Provides servers, networking, storage, load balancers, and data center space
 - Pay-per-use and self-service model
 - Implemented using server virtualization
 - Gives clients full control over resources

- **Platform as a Service (PaaS)**
 - Provides platform to develop, run, and maintain applications
 - Accessed through a web browser
- **Software as a Service (SaaS)**
 - Provides licensed applications to users
 - Delivered via subscription or pay-as-you-go

Cloud Service Management

- Includes:
 - Business Support Services (BSS)
 - Operational Support Services (OSS)
- Purpose:
 - Enable automation
 - Improve service monitoring and management

Cloud Challenges

Major Cloud Computing Challenges

- Security and Privacy
- Portability
- Computing Performance
- Reliability and Availability
- Interoperability

Security and Privacy

- Biggest challenge in cloud computing.
- Data resides:

- Outside the corporate firewall
 - On shared cloud infrastructure
- A single attack can affect multiple clients.
- Mitigated by:
 - Security applications
 - Security hardware
 - Monitoring unusual activities

Portability

- Ability to:
 - Easily migrate applications between cloud platforms
 - Avoid vendor lock-in
- Lack of portability increases dependency on one provider.

Computing Performance

- Data-intensive applications require:
 - High network bandwidth
- High performance leads to:
 - Increased cost
- Low bandwidth cannot meet performance demands.

Reliability and Availability

- Cloud services must be:
 - Reliable
 - Robust
 - Available round the clock (24/7)
- Service outages reduce user trust and system reliability.

Interoperability

- Ability of cloud systems to:
 - Work with applications from different platforms
 - Exchange services and data across heterogeneous environments

Lecture 4 Summary

You learned:

- Centralized vs Distributed processing
- Shared-everything vs Shared-nothing architectures
- Batch vs real-time processing
- Parallel vs distributed computing
- Virtualization and its types
- What cloud computing is
- Cloud types (public, private, hybrid)
- Cloud services (SaaS, PaaS, IaaS)
- Basics of GFS and cloud storage

Driving Big Data with Hadoop Tools and Technologies

- The chapter explains the **main components of Hadoop**:
 - **HDFS (Hadoop Distributed File System)**
 - **MapReduce**
 - **YARN (Yet Another Resource Negotiator)**
- It describes **HDFS features**, such as:
 - Scalability
 - Reliability
 - Robust and fault-tolerant design
- The **architecture and storage methods of HDFS** are discussed.
- The chapter introduces **important big data tools** used throughout the **big data lifecycle**.

Hadoop Ecosystem Tools

- **HBase**
 - A non-relational (NoSQL) database.
 - Designed to handle very large and sparse datasets.
- **Hive**
 - Uses **Hive Query Language (HQL)**.
 - Allows SQL-like queries on large, unstructured data.
- **Pig**

- Provides a high-level language called **Pig Latin**.
 - Simplifies writing MapReduce programs.
- **Mahout**
 - Supports scalable **machine learning algorithms**.
- **Avro**
 - Used for **data serialization**.
- **Sqoop**
 - Transfers large volumes of data between **RDBMS and Hadoop**.
- **Oozie**
 - A **workflow scheduler** for managing Hadoop jobs.

1. Apache Hadoop

- Apache Hadoop is:
 - An **open-source framework**.
 - Written in **Java**.
- It is used for:
 - **Storing** and **processing** very large datasets.
 - Handling structured, semi-structured, and unstructured data.
- Data processing is performed:
 - In **parallel**.
 - Across multiple machines (nodes).

Key Characteristics of Hadoop

Distributed Computing

- Data and processing tasks are spread across many computers.

Scalability

- Performance remains stable as load increases.
- New nodes can be added easily.

Cost-Effective

- Uses clusters of low-cost hardware.

Write-Once, Read-Many Model

- Data is written once.
- Data cannot be modified after being written.

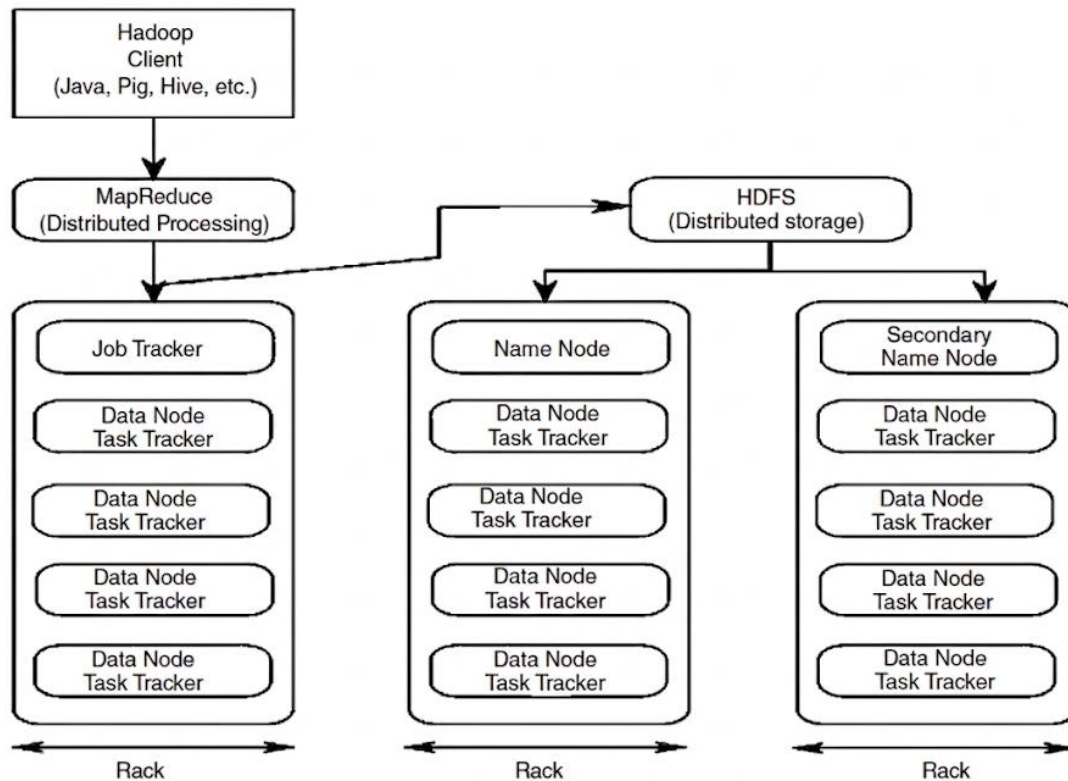
Cluster-Based System

- A cluster is a group of interconnected computers.
- The cluster works as a single system to process large data efficiently.

1.1 Architecture of Apache Hadoop

- The Hadoop architecture consists of **two main layers**:
 - **Storage Layer** → HDFS
 - **Processing Layer** → MapReduce
- **HDFS** is responsible for:
 - Storing data in a **distributed manner** across multiple machines.
- **MapReduce** is placed on top of HDFS and is responsible for:
 - Processing large datasets in parallel.
- Each component of the Hadoop architecture works together to:
 - Store data efficiently.
 - Process data quickly across a cluster.
- The detailed functioning of these components is explained in later sections.

Hadoop Cluster Architecture



Hadoop architecture.

Hadoop Client

- Used to submit jobs.
- Examples: Java, Pig, Hive applications.

MapReduce Layer

- Performs distributed data processing.

JobTracker

- Manages and schedules MapReduce jobs.

TaskTracker

- Executes tasks assigned by the JobTracker.

HDFS Layer

- Handles distributed data storage.

NameNode

- Maintains metadata (file names, locations).

Secondary NameNode

- Supports the NameNode by managing checkpoints.

DataNodes

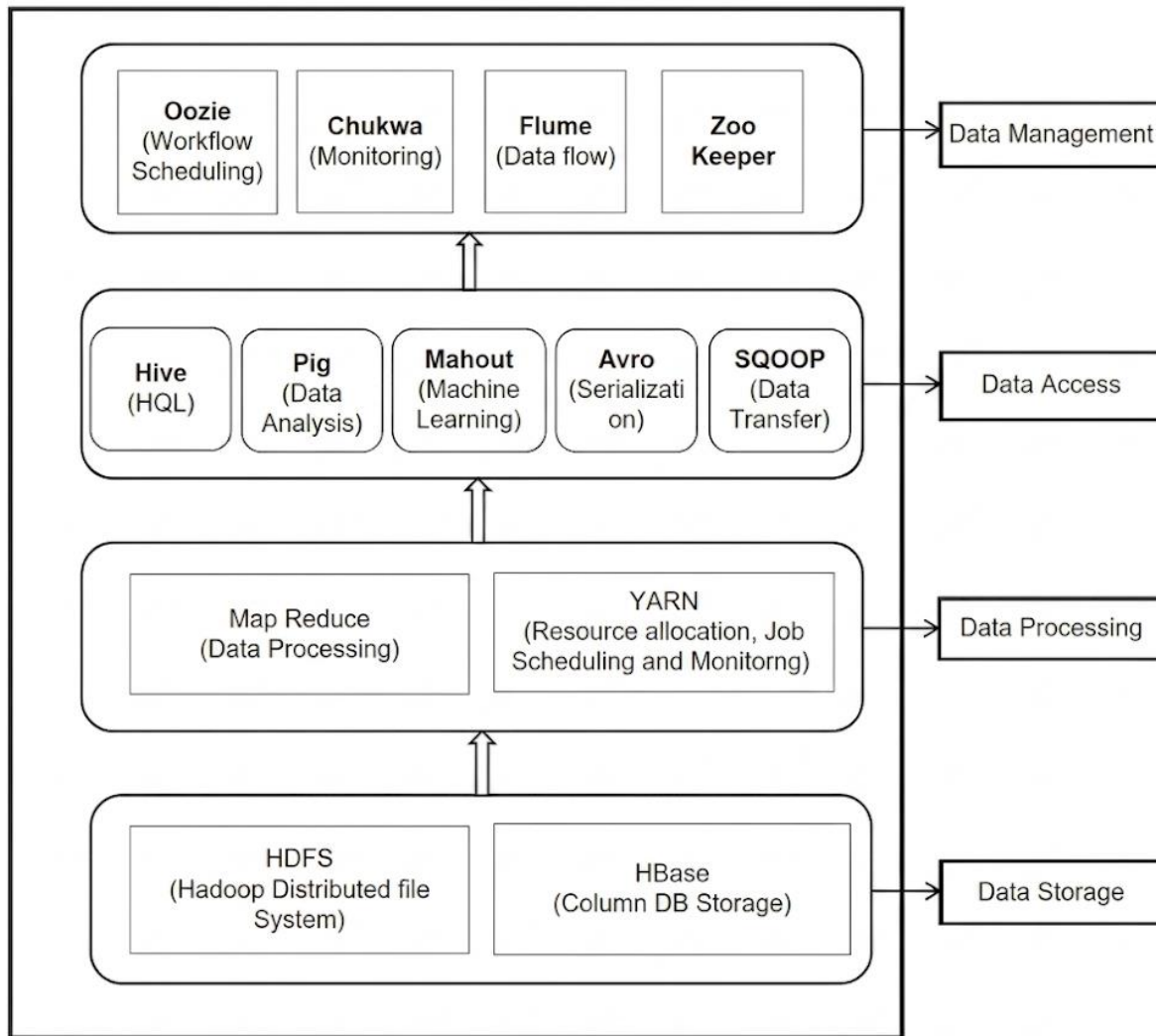
- Store actual data blocks.
- Also run TaskTracker services.

Rack

- Group of nodes connected in the same network segment for fault tolerance.

1.2 Hadoop Ecosystem Components Overview

The Hadoop ecosystem is divided into **four logical layers**.



Hadoop ecosystem.

1. Data Storage Layer

- Responsible for **storing data**.
- Includes:
 - **HDFS**
 - Distributed file system.
 - Stores data across multiple nodes.
 - **HBase**

- Column-oriented NoSQL database.
- Stores structured data on top of HDFS.

2. Data Processing Layer

- Responsible for **processing data**.
- Includes:
 - **MapReduce**
 - Handles distributed data processing.
 - **YARN**
 - Manages resource allocation.
 - Performs job scheduling and monitoring.

3. Data Access Layer

- Provides tools to **access and analyze data**.
- Includes:
 - **Hive**
 - Uses HQL (SQL-like query language).
 - Queries data stored in HDFS.
 - **Pig**
 - High-level scripting language for data analysis.
 - **Mahout**
 - Machine learning platform.
 - **Avro**
 - Data serialization framework.
 - **Sqoop**
 - Transfers data between traditional databases and HDFS.

4. Data Management Layer

- Interacts directly with the **end user**.
- Manages workflows, coordination, and data movement.
- Includes:
 - **Oozie**
 - Workflow scheduling tool.
 - **Chukwa**
 - Used for data collection and monitoring.
 - **Flume**
 - Transfers data from sources to HDFS.
 - **ZooKeeper**
 - Coordinates and manages distributed applications.

Overall

- **Store data** → HDFS, HBase
- **Process data** → MapReduce, YARN
- **Access data** → Hive, Pig, Mahout, Avro, Sqoop
- **Manage data & jobs** → Oozie, Chukwa, Flume, ZooKeeper

2. Hadoop Storage

- Hadoop storage is mainly handled by **HDFS**.
- It is designed to store **very large datasets**.
- Data is stored across **multiple machines** instead of one system.

2.1 HDFS (Hadoop Distributed File System)

Purpose of HDFS

- Stores **large volumes of data** efficiently.
- Uses **low-cost commodity hardware**.
- Does not require expensive or high-end machines.

Data Access Pattern

- Follows **write once, read many times** model.
- Data is written once and cannot be modified.
- Data analysis is performed by reading the stored data multiple times.

Performance Characteristics

- Optimized for **high throughput**, not low latency.
- Not suitable for applications requiring **fast, real-time access**.
- **HBase** is used when low-latency access is needed.

Data Storage Method

- Data is divided into **small blocks (chunks)**.
- Each block of data is:
 - Stored on multiple machines.
 - **Replicated** for fault tolerance.

Fault Tolerance

- Multiple copies of each block are maintained.
- If a block is corrupted or a node fails:
 - Data is recovered from another copy.
- Blocks are stored on **physically separate machines** to ensure reliability.

2.2 Why HDFS?

Problem with a Single Machine

- Reading **500 GB of data** on one machine:
 - Takes about **22.5 minutes**.
 - Data analysis increases total processing time.
- A single machine becomes a **bottleneck**.

Advantage of Distributed File System (HDFS)

- Data is distributed across **many machines**.
- Example:
 - Distributing data across **100 machines**:
 - Read time reduces to about **13.5 seconds**.
- Data access occurs **in parallel**.
- Improves performance significantly.

Single Machine



1 Machine

Distributed File System



100 Machines

Distributed file system vs. single machine.

Key Benefits of HDFS

- Eliminates data access bottlenecks.
- Enables **parallel data retrieval**.

- Increases speed and efficiency.
- Suitable for **big data storage and analysis**.

Overall

- **Single Machine** → Slow, bottleneck, limited scalability
- **HDFS** → Fast, parallel, scalable, fault-tolerant

2.3 HDFS Architecture

General Characteristics of HDFS

- HDFS is **highly fault-tolerant**.
- Designed to run on **low-cost (commodity) hardware**.
- Supports very large data sizes:
 - From **terabytes to petabytes**.
- Easily portable across different platforms.
- Uses a **master-slave architecture**.

Master-Slave Architecture in HDFS

- **One master node** controls the system.
- **Multiple slave nodes** store and process data.
- The architecture improves
 - Scalability
 - Reliability.

Master Node: NameNode

- The master node contains:
 - **NameNode**
- Responsibilities of the NameNode:
 - Manages the **file system namespace**.

- Maintains metadata (not actual data).
- Stores information such as:
 - File names
 - Directory structure
 - Block locations
- Monitors DataNodes using **heartbeat signals**.
- Controls access to files for users.
- Important Characteristics
 - NameNode **does not store actual data**.
 - Acts as a directory for all DataNodes.
 - Keeps track of which blocks form a file and their locations.
 - Performs operations such as:
 - Opening files
 - Closing files
 - Renaming files and directories
- Limitation:
 - NameNode is a **single point of failure**.
 - If NameNode fails, the file system becomes unavailable.
 - If it fails, **manual intervention** is required.
 - Memory limitation:
 - All metadata is stored in NameNode memory.
 - Limits the number of files HDFS can store.

Metadata and Small File Limitation

- All metadata is stored in the **NameNode memory**.

- HDFS is **not suitable for storing many small files**.
- Reason:
 - Each file needs metadata.
 - More small files → more metadata.
 - More metadata → more NameNode memory usage.
- Storage capacity is controlled by:
 - NameNode memory size.
- The total number of files in HDFS depends on:
 - The memory capacity of the NameNode.

Slave Nodes: DataNodes

- Slave nodes are called **DataNodes**.
- DataNodes:
 - Store the **actual data blocks**.
 - Are distributed across the cluster.
- Each DataNode runs:
 - A daemon called **TaskTracker**.
- User files are:
 - Split into blocks.
 - Default block size is **64 MB**.
- Blocks are stored across multiple DataNodes.

Block Placement and Replication

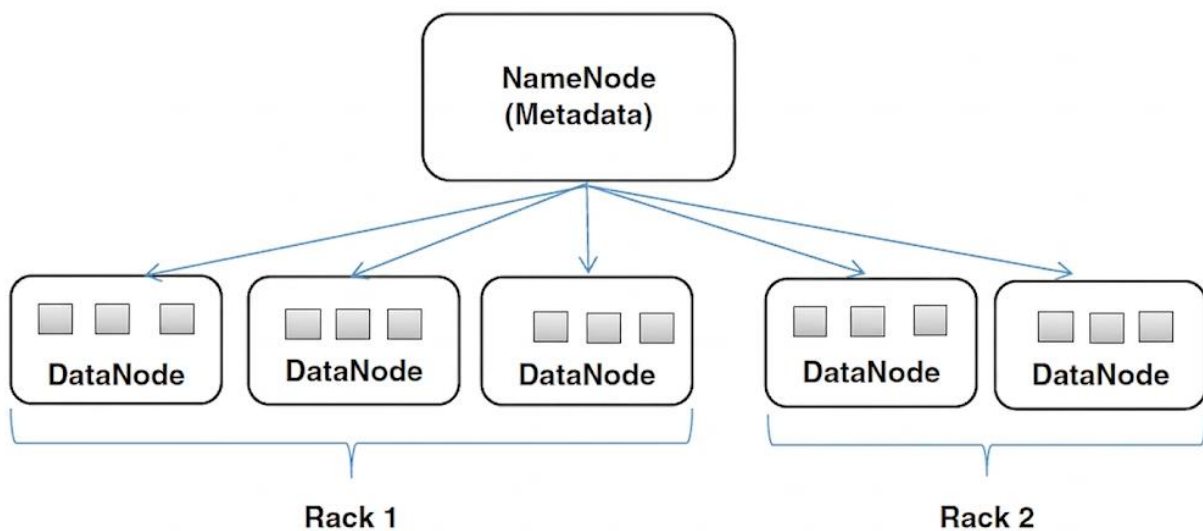
Block Management

- NameNode decides:
 - Which block goes to which DataNode.

- Large files are divided into **blocks**.
- Multiple blocks of the same file:
 - Are stored on different DataNodes.
 - Are stored across different machines.
 - Improve fault tolerance and availability.
- The **NameNode**:
 - Decides which block goes to which DataNode.
 - Maintains block-to-node mapping.

Replication

- Each block is replicated **three times by default**.
- Purpose:
 - Fault tolerance
 - High availability
- Replication factor:
 - Can be changed by the application.



HDFS architecture.

Rack Awareness and Block Placement

Rack Concept

- A **rack** is a group of DataNodes stored together.
- Helps manage hardware failures.

Replica Placement Strategy

- Block replicas are placed as follows:
 - **Replica 1** → One rack
 - **Replica 2 & 3** → Same rack but different DataNodes
- Replica 1 is placed on a **different rack** than replicas 2 and 3.
- This ensures:
 - Protection against rack failure.

Performance Optimization

- Block placement is based on **proximity**.
- Closer nodes → faster communication.

Secondary NameNode

Purpose

- Periodically **backs up NameNode metadata**.
- Stores metadata snapshots.

Key Points

- Does **not replace** the NameNode on failure.
- Acts only as a **recovery mechanism**.
- Runs on a **separate machine**.
- Requires memory similar to NameNode.

Limitation

- Does **not provide high availability**.
- NameNode is still a **single point of failure**.

HDFS Federation

Why HDFS Federation?

- NameNode memory limits cluster scalability.
- Large clusters require more metadata storage.

How Federation Works

- Multiple **NameNodes** are added.
- Each NameNode manages:
 - Its **own independent namespace**.
- NameNodes:
 - Do **not communicate** with each other.

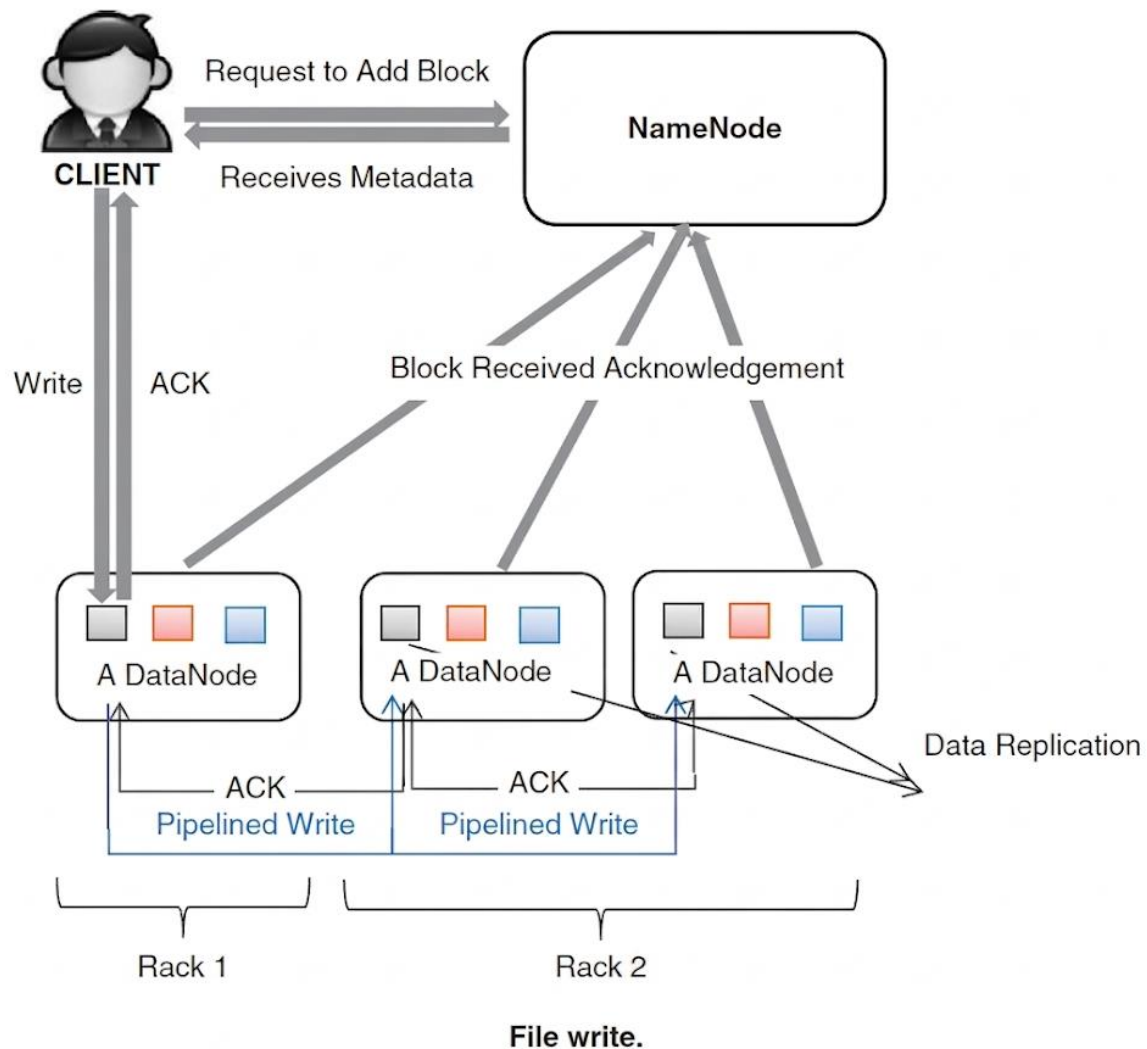
Advantages

- Improves scalability.
- Failure of one NameNode:
 - Does **not affect** other namespaces.

2.4 HDFS Read / Write Operation

HDFS handles file operations in a **distributed and fault-tolerant manner**. The read and write processes involve coordination between the **Client, NameNode, and DataNodes**.

A. HDFS Write Operation (File Write)



Step-by-Step Write Process

1. Client Request

- The HDFS client sends a **write request** to the Distributed File System (DFS).

2. Communication with NameNode

- DFS contacts the **NameNode**.
- NameNode:
 - Checks if the file already exists (duplicate check).

- Creates a new metadata record for the file.

3. DataNode Selection

- NameNode selects DataNodes based on:
 - **Replication factor** (default = 3).
 - **Proximity** (nearest nodes first).

4. Block Creation

- The input file is divided into blocks.
- Default block size:
 - **64 MB**

5. Packet Transfer

- Blocks are sent as **packets**.
- Writing is done in a **pipelined manner**.

6. Pipelined Write Process

- Client sends data packet to the **first DataNode**.
- First DataNode forwards the packet to the **second DataNode**.
- Second DataNode forwards the packet to the **third DataNode**.

7. Acknowledgement (ACK)

- After receiving the complete block:
 - Third DataNode sends ACK to second.
 - Second sends ACK to first.
 - First sends ACK to client.

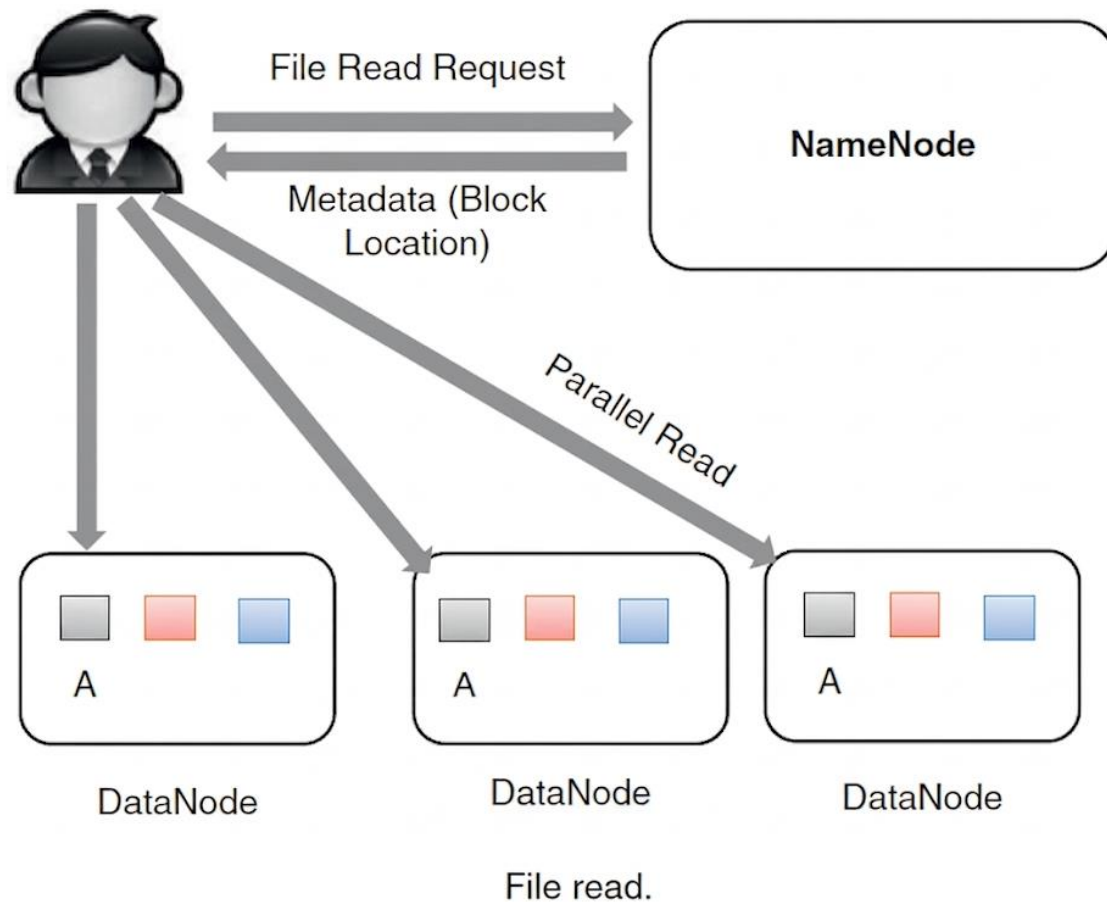
8. Completion

- Once data is written successfully on all replicas:
 - Connections between Client and DataNodes are closed.

Key Points of Write Operation

- Data is written **in parallel**.
- Replication ensures:
 - Fault tolerance
 - Data reliability
- NameNode manages metadata only.
- Actual data is written directly to DataNodes.

B. HDFS Read Operation (File Read)



Step-by-Step Read Process

1. Client Read Request

- Client sends a **read request** to DFS.

2. Metadata Request

- DFS contacts the **NameNode**.
- NameNode returns:
 - Metadata
 - Block locations
 - List of DataNodes storing the blocks
 - Nearest DataNode listed first

3. Client Chooses DataNode

- Client selects the **closest DataNode** for faster access.

4. Data Reading

- Client reads data blocks directly from the DataNode.
- Data is read **block by block**.

5. Repeat Process

- For remaining blocks:
 - Client keeps requesting block locations from NameNode.
 - Reads from DataNodes until the full file is read.

6. Connection Close

- After completing the read:
 - Client closes the connection with DataNodes.

Fault Tolerance During Read

- If a DataNode fails:
 - Data is read from another replica.
- Replication guarantees:
 - Continuous data availability

- No interruption in read operation

Key Differences: Read vs Write

Aspect	Write Operation	Read Operation
Client interacts with	NameNode + DataNodes	NameNode + DataNodes
Data flow	Client → DataNodes	DataNodes → Client
Replication	Happens during write	Uses existing replicas
Fault handling	ACK-based confirmation	Reads from replicas

Overall

- **Write** → File split → Blocks → Replicated → Stored using pipelining
- **Read** → Metadata request → Nearest DataNode → Parallel read
- **NameNode** → Metadata only
- **DataNodes** → Actual data storage
- **Replication** → Fault tolerance

2.6 Features of HDFS

HDFS provides several important features that make it suitable for storing and managing big data efficiently.

2.6.1 Cost-Effective

- HDFS is an **open-source** storage platform.
- It is available **free of cost** to organizations.
- Does **not require expensive or high-end hardware**.
- Uses **commodity (low-cost) hardware** for storage.
- This significantly reduces the overall cost of big data storage.
- Using specialized hardware would make big data storage expensive, which HDFS avoids.

2.6.2 Distributed Storage

- HDFS stores data in a **distributed manner** across multiple nodes.
- Input files are split into **blocks**.
- Default block size:
 - **64 MB**
- Example:
 - A **200 MB file** is divided into:
 - Three 64 MB blocks
 - One 8 MB block
- The smaller block (8 MB):
 - Does not waste space
 - Can share the remaining block space with other files
- This ensures **efficient use of storage space**.

2.6.3 Data Replication

- HDFS provides **data replication** for reliability.
- By default:
 - Each data block is replicated **three times**.
- The replicas are stored on:
 - **Different nodes** in the cluster
- Benefits:
 - Fault tolerance
 - High data availability
- If a node fails:
 - The lost data is retrieved from another replica.

- Data remains accessible without interruption.

Hadoop Computation

1 MapReduce

1. Definition and Purpose

- MapReduce is a *batch-processing programming model* designed for the Hadoop framework.
- It operates based on the *divide-and-conquer* principle to manage large-scale data processing.

2. Key Characteristics

- Highly **scalable**, allowing processing to expand across many computing nodes.
- **Reliable** and **fault-tolerant**, ensuring continued operation despite node failures.
- Capable of processing **input data of any format**.
- Designed specifically for **batch workloads**, not real-time or streaming tasks.

3. Distributed and Parallel Processing

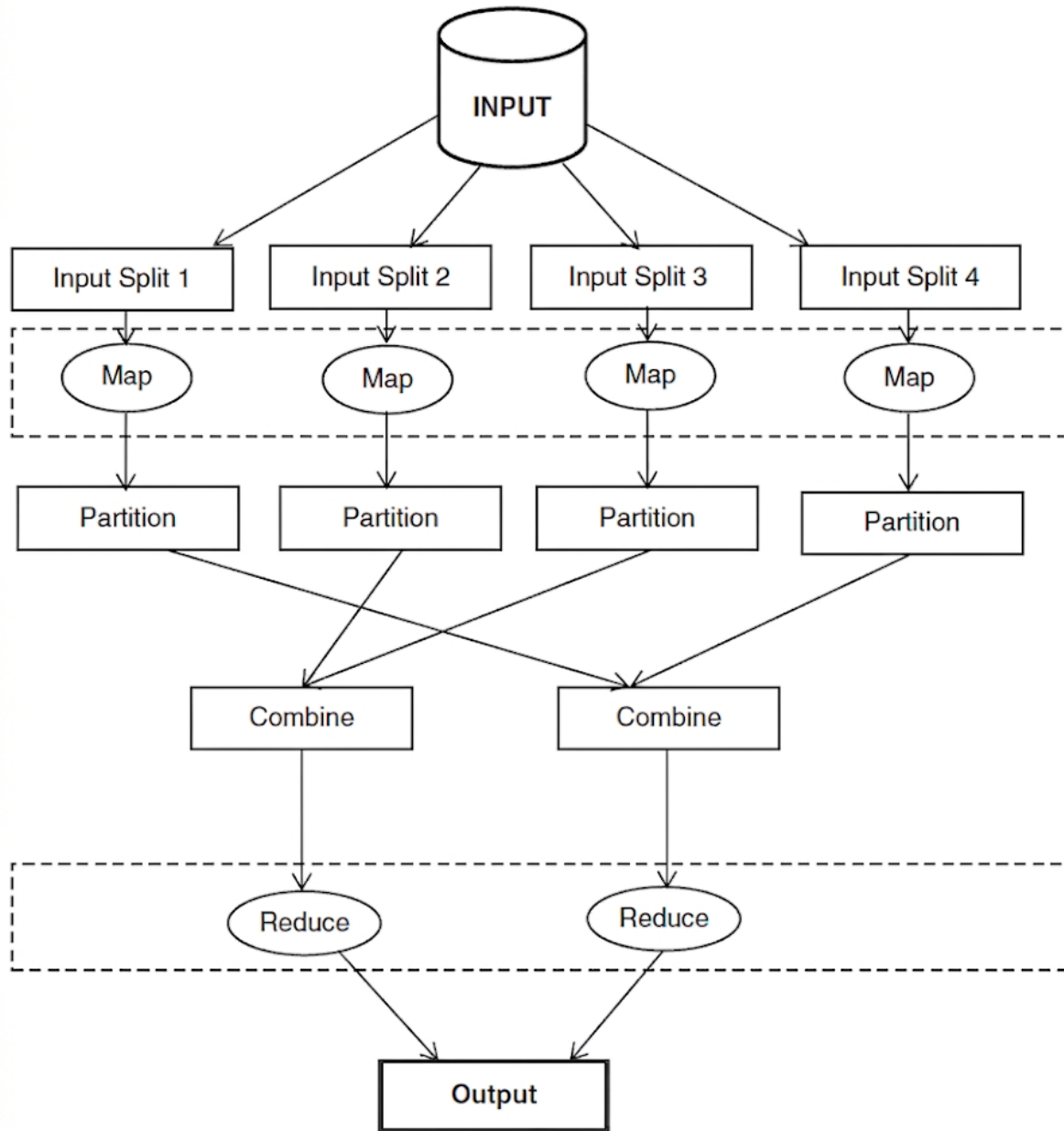
- Executes data processing in a **parallel and distributed computing environment**.
- Significantly reduces processing time compared to traditional batch-processing paradigms.

4. Comparison with Traditional Batch Processing

- Traditional approaches move data from the storage platform to the processing platform.
- In contrast, MapReduce executes the processing *within the framework where the data resides*, minimizing data movement and improving efficiency.

5. Conceptual Model

- The MapReduce computational model is illustrated in the following figure



MapReduce model.

6. Processing Phases

- The MapReduce workflow is divided into **two primary phases**:
 - **Map phase**

- **Reduce phase**

7. Stages within Each Phase

- **Map Phase Stages:**
 - Map
 - Combine (optional)
 - Partition (optional)
- **Reduce Phase Stages:**
 - Shuffle
 - Sort
 - Reduce

8. Optional Components

- The **combiner** and **partitioner** stages are optional and depend on the specific data processing requirements.

9. Role of the Programmer

- The programmer's responsibility is limited to:
 - Supplying the MapReduce program.
 - Providing the input data.
- All subsequent execution and coordination are handled automatically by the Hadoop framework.

10. Overall Advantage

- This abstraction simplifies large-scale data processing and enhances usability by reducing the programmer's involvement in low-level execution details.

1.1 Mapper

1. Position in MapReduce Workflow

- The *Mapper* constitutes the **first stage of the map phase** in the MapReduce processing model.

2. Data Decomposition

- During the mapper stage, a **large dataset is divided into multiple smaller data blocks** to facilitate parallel processing.

3. Input Representation

- Each data block is transformed into a set of **key–value pairs**, denoted as (K_1, V_1) .
- These key–value pairs serve as the **input** to the mapper function.

4. Parallel Execution

- Each data block is processed by an **individual map task (map job)**.
- Multiple map tasks execute **concurrently**, enabling large-scale parallelism.

5. User-Defined Logic

- The mapper applies the **logic defined by the user** within the MapReduce program.
- This logic determines how the input key–value pairs are analyzed and transformed.

6. Intermediate Output

- The mapper generates **intermediate key–value pairs** as output.
- These intermediate pairs are represented as (K_2, V_2) .

7. Multiplicity of Values

- A single intermediate key (K_2) may be associated with **multiple values** (V_2).
- The mapper output is therefore expressed as a **list of intermediate key–value pairs**, denoted as (K_2, V_2) .

8. Role in the Overall Framework

- The mapper stage lays the foundation for downstream processing by producing structured intermediate data that is subsequently handled by the shuffle, sort, and reduce phases.

1.2 Combiner

1. Purpose of the Combiner

- The *Combiner* is used to **optimize the output of the mapper** before the data is transferred to the reducer.
- Its primary objective is to **reduce the overhead associated with transferring large data sets** between the mapper and reducer stages.

2. Position in the MapReduce Workflow

- The combiner operates **after the mapper phase and before the reducer phase**.
- It acts as an intermediate processing step within the map job.

3. Functional Role

- The combiner functions as a **local reducer** for the mapper output.
- It performs logical grouping of the mapper's output, which consists of multiple key-value pairs.

4. Processing Mechanism

- Repeated keys produced by the mapper are **combined locally** within the combiner.
- The values corresponding to the same key are **aggregated or listed**, depending on the defined logic.

5. Input and Output Characteristics

- Input to the combiner consists of **intermediate key-value pairs** generated by the mapper.
- Output from the combiner is a **reduced set of key-value pairs**, resulting in less data being forwarded to the reducer.

6. Performance Benefits

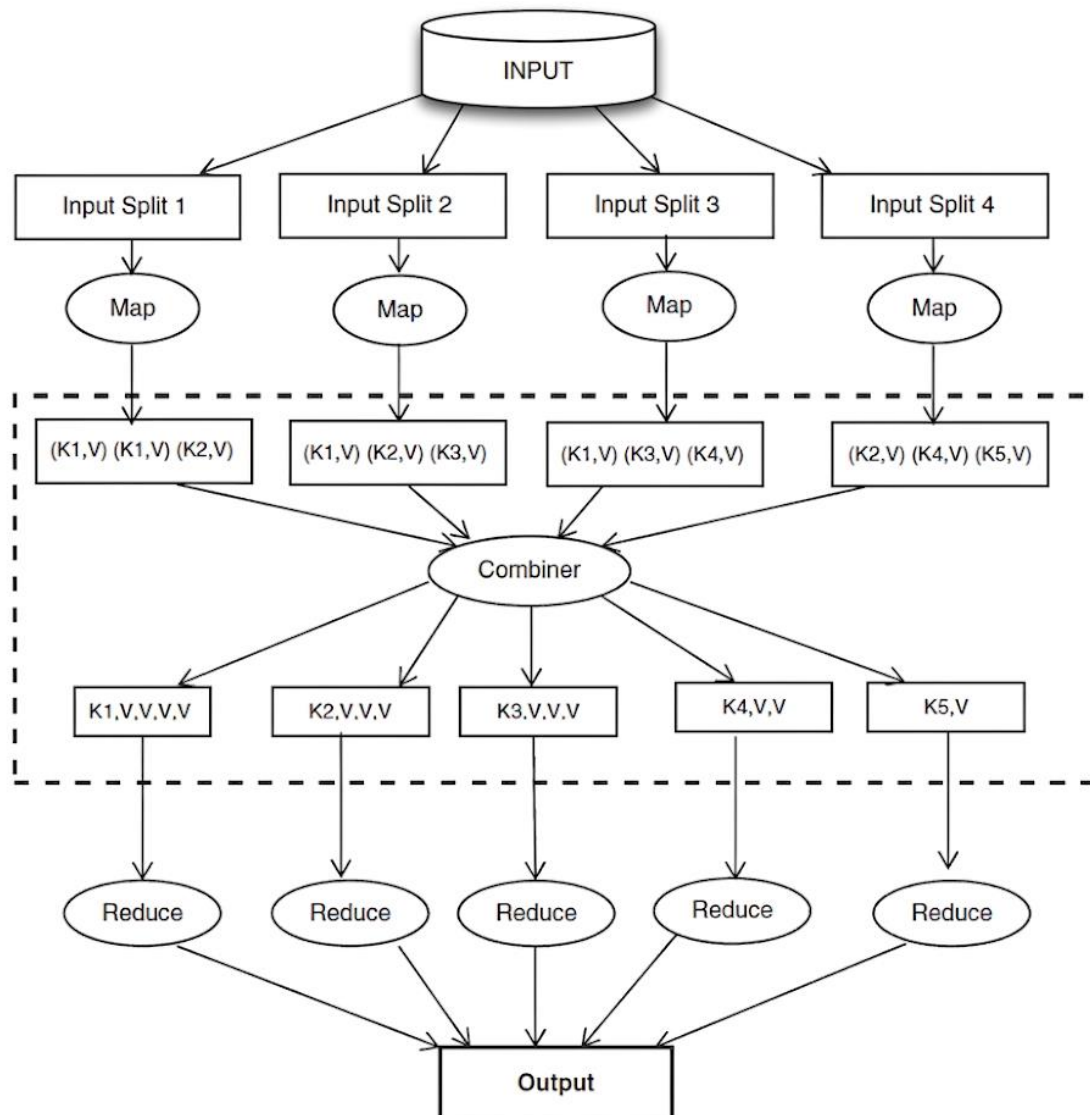
- By minimizing the volume of data transmitted across the network, the combiner **improves overall performance and efficiency** of MapReduce jobs.
- It helps in reducing network congestion and execution time.

7. Optional Nature

- The use of a combiner is **optional** and depends on the nature of the MapReduce application and the suitability of the aggregation operation.

8. Combiner Illustration

- The following figure demonstrates how processing is carried out in the combiner stage.



Combiner illustration.

1.3 Reducer

1. Role and Function

- The *Reducer* performs the **logical operation specified by the user** in the MapReduce program.
- It is responsible for producing the **final output** of the MapReduce computation.

2. Execution Model

- Each reducer executes **independently and in isolation** from other reducers.
- Reducers **do not communicate with one another**, ensuring parallel and isolated processing.

3. Input Characteristics

- The input to the reducer consists of **intermediate key–value pairs** generated by the previous stages.
- These inputs are **sorted and grouped based on their keys** before being processed by the reducer.

4. Processing Mechanism

- For each distinct key, the reducer processes the **set of associated values**.
- The reducer applies the user-defined logic to compute results from these values.

5. Output Representation

- The reducer generates **output key–value pairs** as a result of its processing.
- The output key–value pair may:
 - Remain identical to the input key–value pair, or
 - Be transformed according to the user-defined reduce function.

6. Final Data Storage

- The output produced by the reducer is **written back to the Distributed File System (DFS)**.
- This output represents the final result of the MapReduce job.

7. Significance in the MapReduce Framework

- The reducer consolidates and summarizes intermediate results, enabling meaningful aggregation and analysis of large-scale data.

1.4 JobTracker and TaskTracker

1. Architectural Overview

- Hadoop MapReduce follows a **master–slave architecture**.
- The framework comprises:
 - **One JobTracker** (master component).
 - **Multiple TaskTrackers** (slave components).

2. Deployment and Node Roles

- The **JobTracker** executes on the **master node**.
- Each **TaskTracker** executes on a **slave node**, with **exactly one TaskTracker per slave node**.
- System components are co-located such that:
 - **TaskTracker and NameNode** may run on one machine.
 - **JobTracker and DataNode** may run on another machine.
- This configuration enables each node to perform both **computation and storage tasks**.

3. Responsibilities of JobTracker

- Manages **parallel processing** in the MapReduce framework.
- Accepts **job submission requests** from clients.
- Divides jobs into tasks and **schedules tasks** on slave nodes.
- Monitors the **health and availability of slave nodes**.
- Tracks and maintains the **progress of running tasks**.
- Acts as the **central coordinator** for workflow execution.

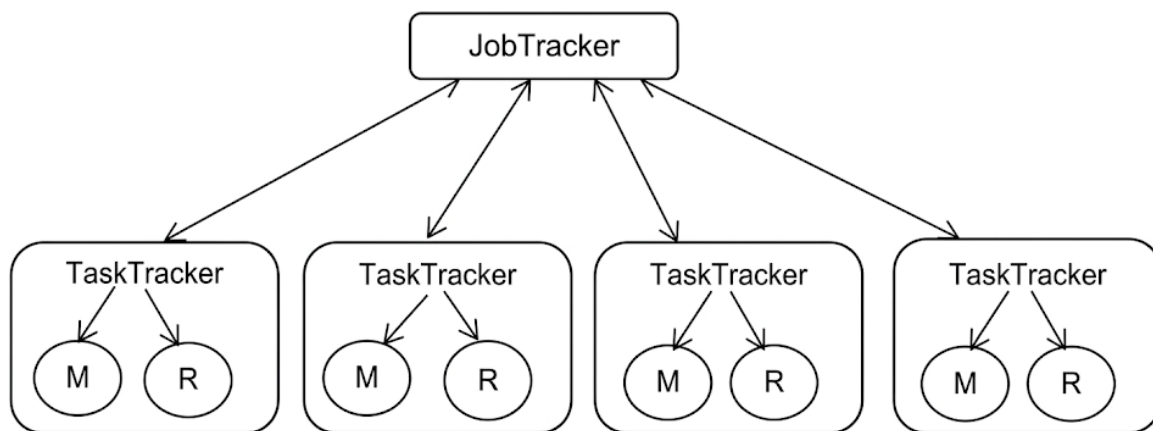
4. Responsibilities of TaskTracker

- Accepts tasks assigned by the JobTracker.
- Executes the **user-defined map and reduce code**.

- Manages **workflow execution and local resource utilization**.
- Sends **periodic progress updates** to the JobTracker.

5. Communication Model

- Communication between JobTracker and TaskTracker occurs via **Remote Procedure Calls (RPC)**.
- Communication is **bidirectional**:
 - JobTracker assigns tasks to TaskTrackers.
 - TaskTrackers report task status and progress back to the JobTracker.
- The following figure illustrates:
 - JobTracker as the **master**.
 - TaskTrackers as the **slaves**, executing map (M) and reduce (R) tasks.



JobTracker and TaskTracker.

6. Heartbeat Mechanism

- Each TaskTracker sends a **heartbeat signal** to the JobTracker at regular intervals.
- The heartbeat indicates:

- That the node is alive.
- Whether it is currently processing tasks or available for new tasks.
- If a heartbeat is not received within a predefined time interval, the TaskTracker is **assumed to have failed**.

7. Fault Detection and Recovery

- When a task fails:
 - The failure is detected by the TaskTracker.
 - The failure is reported to the JobTracker.
- The JobTracker **reschedules the failed task**, either:
 - On the same node, or
 - On a different node in the cluster.

8. Failure Handling Policies

- If **multiple tasks from the same job fail** on a single TaskTracker:
 - That TaskTracker is prevented from executing further tasks of that job.
- If tasks from **different jobs fail** on the same TaskTracker:
 - The TaskTracker is barred from executing **any tasks for the next 24 hours**.

9. Single Point of Failure

- The JobTracker represents a **single point of failure** in the Hadoop MapReduce architecture.
- If the JobTracker fails:
 - All running tasks in the cluster eventually fail.
- Therefore, the machine hosting the JobTracker must be **highly reliable**.

10. Job Monitoring and Memory Management

- Upon job submission, information about all active tasks is **stored in memory**.
- Task progress is updated with each heartbeat signal.
- This mechanism provides the **end user with real-time visibility** into job execution.
- In clusters executing multiple jobs simultaneously:
 - It becomes difficult to predict **RAM consumption**.
 - Consequently, **monitoring memory utilization of the JobTracker** is critically important.

11. Overall Significance

- The JobTracker–TaskTracker model provides centralized coordination and distributed execution.
- While effective for batch processing, the architecture is constrained by scalability and fault-tolerance limitations due to the centralized JobTracker.

2 MapReduce Input Formats

1. Primitive Data Types in Hadoop

Hadoop defines a set of *primitive writable data types* that are used for serialization and data exchange in MapReduce:

- **BooleanWritable**
- **ByteWritable**
- **IntWritable**
- **VIntWritable**
- **FloatWritable**
- **LongWritable**
- **VLongWritable**

- **DoubleWritable**

These writable types support efficient storage, transmission, and processing of data in distributed environments.

2. Supported MapReduce Input File Formats

MapReduce supports multiple input formats for reading source data:

1. **TextInputFormat**
2. **KeyValueTextInputFormat**
3. **NLineInputFormat**
4. **SequenceFileInputFormat**
5. **SequenceFileAsTextInputFormat**

Each input format determines how input data is split and how key–value pairs are generated for the mapper.

3. TextInputFormat

1. **Default Input Format**
 - TextInputFormat is the **default input format** in MapReduce.
2. **Data Splitting**
 - The input file is divided into **lines of text**.
 - Each line corresponds to a single record.
3. **Key–Value Generation**
 - **Key:** A value of type *LongWritable*, representing the **byte offset** of the start of the line within the file.
 - **Value:** The **content of the line**, excluding line terminators (newline or carriage return).
4. **Example Input**

This is the first line of the input file,

*This is the second line of the input file,
And this is the last line of the input file.*

5. Generated Key–Value Pairs of the Above Input

*(0, This is the first line of the input file,)
(41, This is the second line of the input file,)
(82, And this is the last line of the input file.)*

6. Significance of Offset

- The byte offset acts as a **unique identifier** for each record.
- The combination of offset and file name uniquely identifies records within a file.

4. KeyValueTextInputFormat

1. Purpose

- KeyValueTextInputFormat is used for **plain text files** where each line represents an explicit key–value pair.

2. Data Interpretation

- Similar to TextInputFormat, the file is split into lines.
- Each line is interpreted as a key–value pair using a **separator character**.

3. Separator

- The default separator is a **tab character**.
- For illustration purposes, a **comma** is used in the example.

4. Example Input

*Line1, First line of input,
Line2, Second line of input,
Line3, Third line of input.*

5. Key–Value Extraction Rule

- All characters **before the first separator** are treated as the key.
- All characters **after the separator** are treated as the value.

6. Generated Key–Value Pairs

(Line1, First line of input,)

(Line2, Second line of input,)

(Line3, Third line of input.)

5. NLineInputFormat

1. Motivation

- In TextInputFormat and KeyValueTextInputFormat, the number of lines processed by a mapper depends on input split size and line length.

2. Functionality

- NLineInputFormat ensures that each mapper receives a **fixed number of lines** as input.

3. Use Case

- It is particularly useful when uniform input distribution across mappers is required.

6. SequenceFileInputFormat

1. Purpose

- SequenceFileInputFormat is used to process **binary files** containing serialized key–value pairs.

2. Data Storage

- Stores key–value pairs in **sequence**, enabling efficient read and write operations.

3. Applications

- Commonly used for reading:
 - Sequence files.

- Map files.

7. SequenceFileAsTextInputFormat

1. Purpose

- SequenceFileAsTextInputFormat is used to **convert binary sequence file key-value pairs into text format**.

2. Use Case

- Facilitates human-readable processing of sequence file contents within MapReduce jobs.

8. Overall Significance

- MapReduce input formats provide **flexibility in data ingestion** by supporting text, structured, fixed-line, and binary data.
- Selection of an appropriate input format directly impacts **performance, load balancing, and correctness** of MapReduce applications.

3 MapReduce Example

1. Purpose of the Example

1. The example demonstrates the **working principle of MapReduce** using a small dataset.
2. Although the dataset is minimal for explanatory purposes, in real-world applications MapReduce is capable of processing **terabytes to petabytes of data**.
3. The example uses **temperature records from different cities**, collected across multiple files.

File 1 <table> <tr> <th>City</th><th>Temperature Recorded</th></tr> <tr><td>Leeds</td><td>20</td></tr> <tr><td>Bexley</td><td>17</td></tr> <tr><td>Bradford</td><td>11</td></tr> <tr><td>Bradford</td><td>15</td></tr> <tr><td>Bexley</td><td>19</td></tr> <tr><td>Bradford</td><td>21</td></tr> </table>	City	Temperature Recorded	Leeds	20	Bexley	17	Bradford	11	Bradford	15	Bexley	19	Bradford	21	File 2 <table> <tr> <th>City</th><th>Temperature Recorded</th></tr> <tr><td>Leeds</td><td>16</td></tr> <tr><td>Bexley</td><td>12</td></tr> <tr><td>Bradford</td><td>11</td></tr> <tr><td>Leeds</td><td>13</td></tr> <tr><td>Bexley</td><td>18</td></tr> <tr><td>Bradford</td><td>17</td></tr> </table>	City	Temperature Recorded	Leeds	16	Bexley	12	Bradford	11	Leeds	13	Bexley	18	Bradford	17
City	Temperature Recorded																												
Leeds	20																												
Bexley	17																												
Bradford	11																												
Bradford	15																												
Bexley	19																												
Bradford	21																												
City	Temperature Recorded																												
Leeds	16																												
Bexley	12																												
Bradford	11																												
Leeds	13																												
Bexley	18																												
Bradford	17																												
File 3 <table> <tr> <th>City</th><th>Temperature Recorded</th></tr> <tr><td>Leeds</td><td>19</td></tr> <tr><td>Bexley</td><td>15</td></tr> <tr><td>Bradford</td><td>12</td></tr> <tr><td>Bexley</td><td>13</td></tr> <tr><td>Bexley</td><td>14</td></tr> <tr><td>Bradford</td><td>15</td></tr> </table>	City	Temperature Recorded	Leeds	19	Bexley	15	Bradford	12	Bexley	13	Bexley	14	Bradford	15	File 4 <table> <tr> <th>City</th><th>Temperature Recorded</th></tr> <tr><td>Leeds</td><td>22</td></tr> <tr><td>Bexley</td><td>15</td></tr> <tr><td>Bradford</td><td>12</td></tr> <tr><td>Leeds</td><td>18</td></tr> <tr><td>Leeds</td><td>21</td></tr> <tr><td>Bradford</td><td>20</td></tr> </table>	City	Temperature Recorded	Leeds	22	Bexley	15	Bradford	12	Leeds	18	Leeds	21	Bradford	20
City	Temperature Recorded																												
Leeds	19																												
Bexley	15																												
Bradford	12																												
Bexley	13																												
Bexley	14																												
Bradford	15																												
City	Temperature Recorded																												
Leeds	22																												
Bexley	15																												
Bradford	12																												
Leeds	18																												
Leeds	21																												
Bradford	20																												

4. In this context:

- **Key:** City name
- **Value:** Temperature recorded

2. Input Data Description

1. Structure of Input

- There are **four input files**.
- Each file contains **two columns**:
 - City
 - Temperature recorded
- Each row corresponds to a temperature observation for a city on a given day.

2. Nature of the Dataset

- Cities appear **multiple times across and within files**.
- The dataset is suitable for aggregation operations such as **maximum temperature per city**.

3. Map Phase Processing

1. Mapper Function

- Each record is processed independently by the mapper.
- The mapper emits intermediate **key–value pairs** of the form:
 - *(City, Temperature)*

2. Intermediate Output

- After the map job, all records from the four input files are transformed into city–temperature pairs.
- Example mapped output includes:

Result After the Map job		
Bradford,21	Bexley,19	Leeds,20
Bradford,17	Bexley,18	Leeds,16
Bradford,15	Bexley,15	Leeds,19
Bradford,20	Bexley,15	Leeds,22

3. Characteristics of Mapper Output

- Multiple values are associated with the same key.
- No aggregation is performed at this stage.
- Output is unordered and distributed.

4. Shuffle and Sort (Implicit Stage)

1. The MapReduce framework automatically:
 - Groups intermediate values by key.
 - Sorts the data so that all values for a given city are sent to the same reducer.

5. Reduce Phase Processing

1. Reducer Function

- The reducer processes all values associated with a given city.
- In this example, the reducer computes the **maximum temperature recorded** for each city.

2. Reduced Output

- After applying the reduce logic, the final output consists of one record per city:

Result After the reduce job		
Bradford,21	Bexley,19	Leeds,22

3. Output Characteristics

- Each key appears exactly once in the final output.
- The reducer output represents **aggregated and meaningful results** derived from the raw data.

6. Conceptual Significance of the Example

1. The example illustrates:

- How MapReduce handles **distributed input files**.
- Transformation of raw data into intermediate key–value pairs.
- Aggregation of values using reducers.

2. It highlights core MapReduce principles:

- Parallel data processing.
- Key-based grouping.
- Scalable aggregation.

3. The workflow demonstrated is representative of common analytics tasks such as:

- Maximum/minimum computation

- Summarization
- Statistical aggregation

4 MapReduce Processing

1. Role of the RecordReader

1. Each input file is first **divided into blocks** according to the Hadoop Distributed File System (HDFS) block size.
2. These blocks are read by the **RecordReader** component.
3. The RecordReader:
 - Identifies and verifies the **input file format**.
 - Defaults to **TextInputFormat** if no format is explicitly specified.
4. It reads **one logical record at a time** from each block.
5. The RecordReader converts raw input data into **key–value pairs**, which are then supplied to the mapper.

2. Example Input File (TextInputFormat)

The example considers a simple text file used for a **word count application**, where each line represents a record:

Hi how are you

How is your job

How is your family

How is your brother

How is your sister

How is your mother

How is the climate there in your city

Each line is treated as an independent input record.

3. File Splitting into Blocks

1. Consider an input file size of **150 MB**.
2. With a default block size of **64 MB**, the file is divided into:
 - Block 1: 64 MB
 - Block 2: 64 MB
 - Block 3: 22 MB

Hi how are you How is your Job How is your family	64MB	}
How is your brother How is your sister How is your mother	64MB	}
How is the climate there in your city	22MB	

3. Each block is processed **independently and in parallel** by separate map tasks.

4. RecordReader Output to Mapper

1. The RecordReader emits records in the form:
 - *(byteOffset, entireLine)*
2. For example:
 - The first record generates:
 - *(0, "Hi how are you")*
3. The next record begins at a byte offset of **15**, since the first line contains 14 characters plus a delimiter.
4. These key–value pairs form the **input to the mapper**.

5. Mapper Processing (Word Count Example)

1. The mapper processes each input record independently.
2. In the word count algorithm:
 - The **word** acts as the key.

- The **occurrence count (initially 1)** acts as the value.
- 3. The mapper emits intermediate key–value pairs such as:
 - (how, 1), (is, 1), (your, 1), etc.
- 4. This transformation organizes unstructured text into structured intermediate data.

6. Shuffle and Sort Phase

1. The MapReduce framework groups all intermediate key–value pairs by **identical keys**.
2. Values corresponding to the same key from different mappers are brought together.
3. The grouped key–value pairs are **sorted**, typically in alphabetical order of keys.

7. Reducer Processing

1. The reducer receives a key and a list of all its associated values.
2. In the word count example:
 - All values for a word are **summed** to compute the total frequency.
3. The reducer produces a final key–value pair of the form:
 - (*word*, *totalOccurrences*)

8. Role of the Combiner

1. Each mapper may use a **combiner**, which acts as a *mini-reducer*.
2. The combiner:
 - Aggregates key–value pairs **locally within each mapper output**.
 - Reduces the amount of data transferred to the reducer.
3. This optimization:
 - Decreases network traffic.
 - Improves job performance.

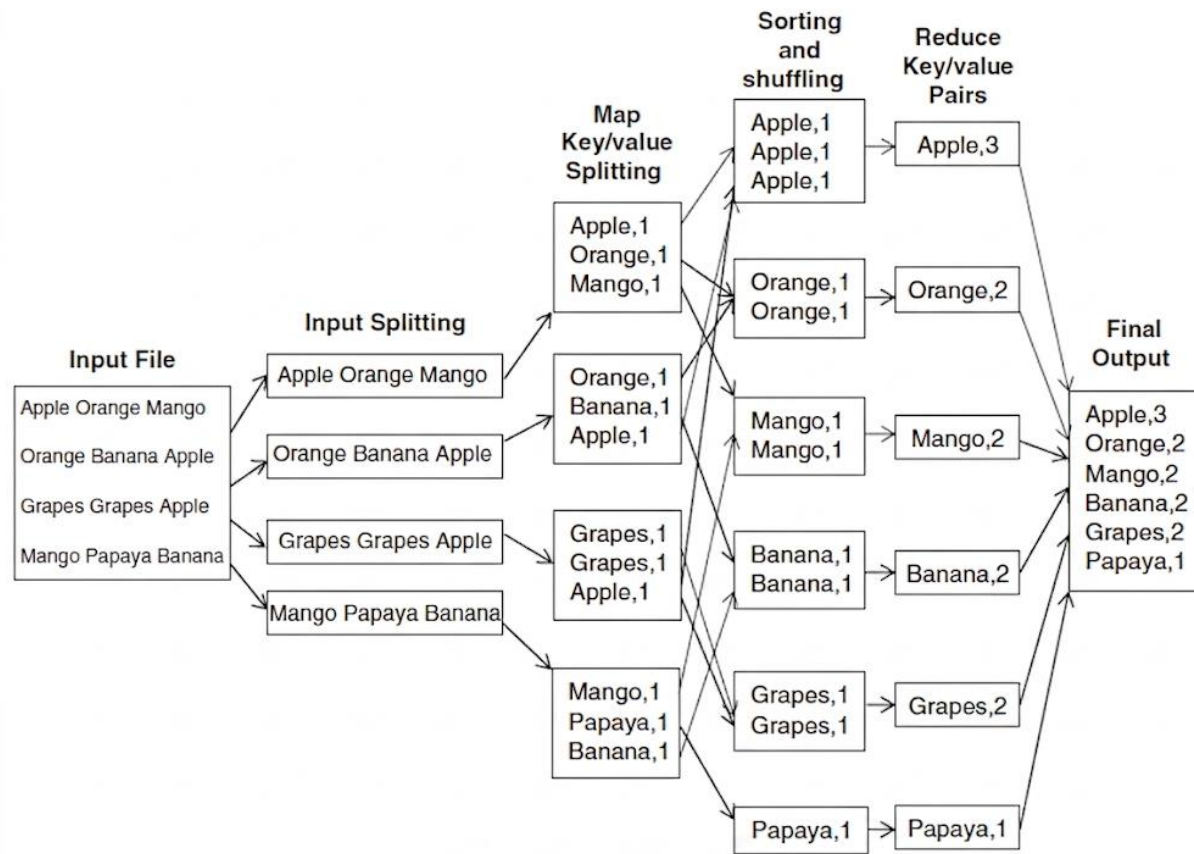
4. The combiner output is subsequently sent to the reducer for final aggregation.

9. Final Output Generation

1. After the reduce phase is completed:
 - The final output consists of aggregated key–value pairs.
2. This output is:
 - Written back to the **Distributed File System (DFS)**.
3. The number of output files generated depends on the **number of reducers**:
 - One output file is produced per reducer.

10. Conceptual Significance

1. This example illustrates the **end-to-end execution flow** of a MapReduce job:
 - File splitting
 - Record reading
 - Mapping
 - Combining
 - Reducing
2. It highlights how MapReduce:
 - Abstracts complexity from the programmer.
 - Efficiently processes large datasets using parallelism.
3. The word count example serves as a canonical demonstration of:
 - Data locality
 - Key-based aggregation
 - Scalable batch processing



Word count algorithm.

5 MapReduce Algorithm

1. Structural Components of a MapReduce Task

1. A MapReduce task is composed of two primary classes:
 - **Mapper class**
 - **Reducer class**
2. Each class has a distinct role in processing and transforming data within the MapReduce framework.

2. Responsibilities of the Mapper Class

1. The mapper class performs the following functions:
 - **Tokenizing** the input data.

- **Mapping** input records into intermediate key–value pairs.
- Participating in **shuffling and sorting**, which organize mapper outputs.
- 2. The mapper processes the input dataset and produces intermediate results that are forwarded to the reducer.
- 3. Mapper output consists of **intermediate key–value pairs**, which serve as the foundation for subsequent reduce operations.

3. Responsibilities of the Reducer Class

1. The reducer class:
 - Takes the **output of the mapper** as its input.
 - Performs a **searching and aggregation task** to find matching key–value pairs.
2. The reducer combines and processes grouped values for each key to generate the **final reduced output**.

4. Role of Algorithms in MapReduce

1. MapReduce employs various algorithms to:
 - Divide a large task into **multiple smaller tasks**.
 - Distribute these tasks across **multiple nodes** in a cluster.
2. Efficient task assignment ensures:
 - Load balancing.
 - Parallel execution.
 - Improved scalability.

5. Sorting Algorithm in MapReduce

1. A **sorting algorithm** is used to process and analyze mapper output.
2. The intermediate key–value pairs generated by the mapper are:
 - Automatically sorted by Hadoop.

- Grouped to form key-based lists, represented as $(K_1, \{V_1, V_1, \dots\})$.
- 3. The **RawComparator** class is employed to:
 - Identify and group similar keys.
 - Facilitate efficient sorting of intermediate data.
- 4. Sorted and grouped data is then presented to the reducer as input.

6. Searching Algorithm in MapReduce

1. A **searching algorithm** is applied when pattern matching or lookup operations are required.
2. Searching is particularly useful when:
 - Filenames and textual content are provided as input.
 - Specific patterns or values need to be identified.
3. Example use case:
 - In a file containing employee names and salaries, a searching algorithm can determine the **employee with the maximum salary** and produce the corresponding name and salary as output.

7. Indexing in MapReduce

1. **Indexing** in MapReduce refers to the association of data with its corresponding address.
2. The indexing technique used is known as an **inverted index**.
3. In an inverted index:
 - Terms are mapped to the documents in which they appear.
4. Inverted indexing is widely used in:
 - Search engines such as Google.
 - Large-scale text retrieval systems.

8. Term Frequency–Inverse Document Frequency (TF-IDF)

1. **TF-IDF** stands for *Term Frequency–Inverse Document Frequency*.
2. It is a **text-processing and information retrieval algorithm**.
3. **Term Frequency (TF):**
 - Measures the number of times a term appears in a document.
4. **Inverse Document Frequency (IDF):**
 - Is calculated by dividing:
 - The total number of documents in a dataset
 - By the number of documents containing the given term.
5. TF-IDF helps:
 - Identify the importance of a term within a document relative to the entire dataset.
 - Reduce the influence of commonly occurring but less informative words.

9. Overall Significance of MapReduce Algorithms

1. Algorithms such as **sorting, searching, indexing, and TF-IDF** enable MapReduce to:
 - Perform complex data analytics.
 - Support large-scale text processing and mining tasks.
2. These algorithms are fundamental in transforming raw data into **organized, meaningful, and searchable results**.

6 Limitations of MapReduce

1. General Perspective

1. MapReduce is recognized as one of the most **successful parallel processing frameworks**.

2. It is widely used by the research community to solve **data-intensive problems** in domains such as:
 - Environmental science
 - Finance
 - Bioinformatics
3. Despite its success and widespread adoption, MapReduce exhibits several **inherent limitations**.

2. Scalability Constraints

1. The primary intrinsic limitation of MapReduce lies in its **one-way scalability**.
2. MapReduce is designed to:
 - Scale **upwards** efficiently for processing very large datasets.
 - Be less effective when processing **smaller datasets**.
3. The limitation arises because:
 - The **NameNode** must store metadata for all files.
 - Maintaining metadata for a very large number of small files leads to **inefficient memory utilization**.
4. Consequently, MapReduce is not well-suited for workloads involving numerous small data sets.

3. Availability and Fault Tolerance Issues

1. The **NameNode** acts as a **single point of failure** in the Hadoop ecosystem.
2. If the NameNode fails:
 - The entire Hadoop cluster becomes unavailable.
3. This limitation restricts the system from being **highly available**.
4. High availability is a critical requirement for many modern applications.
5. As a result, it became necessary to design systems that are:

- Highly scalable
- Highly available simultaneously

4. Rigid Execution Model

1. In standard MapReduce:
 - The **reduce phase cannot begin until all map tasks have completed.**
2. Similarly:
 - A new map task cannot start until the reduce task of the previous application has completed.
3. This strict sequential dependency causes:
 - Increased job completion time
 - Reduced flexibility in task scheduling

5. Resource Underutilization

1. When **map tasks are executing**:
 - Reducer nodes remain idle.
2. When **reduce tasks are executing**:
 - Mapper nodes remain idle.
3. This behavior leads to:
 - **Inefficient utilization of cluster resources**
 - Increased operational cost
4. In some scenarios:
 - Resources remain idle while waiting for map tasks to complete before the reduce phase can start.

6. Performance Bottlenecks

1. Because of the tightly coupled execution phases:
 - Parallelism between map and reduce stages is limited.

2. The design can cause delays when:
 - Large map phases occupy resources while reducers wait.
3. These bottlenecks impact overall **system performance and throughput**.

7. Architectural Implications

1. Due to the outlined limitations:
 - MapReduce is less suitable for interactive, real-time, or iterative workloads.
2. These shortcomings motivated the development of:
 - More flexible data processing frameworks
 - Systems that support better resource utilization and high availability

Lecture 8

4 Hadoop 2.0

1. Architectural Advancements

1. Hadoop 2.0 introduces a **revised architectural design** that significantly enhances the reliability of the Hadoop Distributed File System (HDFS).
2. This architecture addresses critical availability concerns present in earlier Hadoop versions.

2. High Availability of HDFS

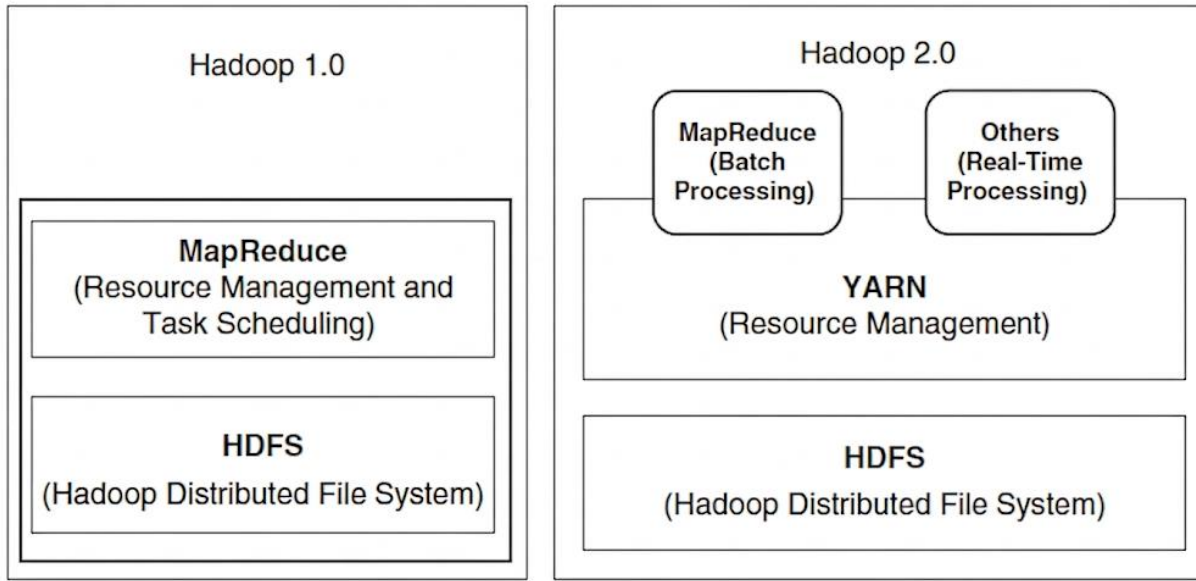
1. In Hadoop 2.0, HDFS is designed as a **highly available filesystem**.
2. This is achieved through the use of:
 - An **Active NameNode**
 - A **Standby NameNode**
3. Both NameNodes operate within an **active-standby configuration**, ensuring continuity of service.

3. Failover Mechanism

1. Under normal operation:
 - The **Active NameNode** handles all client requests and filesystem operations.
2. In the event of failure of the Active NameNode:
 - The **Standby NameNode automatically takes over** the responsibilities.
3. This failover occurs **without interruption** to client requests or filesystem accessibility.

4. Improvement over Hadoop 1.0

1. Hadoop 2.0 eliminates the **single point of failure associated with the NameNode** in Hadoop 1.0.
2. This architectural enhancement results in:
 - Improved system reliability
 - Increased fault tolerance
 - Higher availability for mission-critical applications
3. The comparison between Hadoop 1.0 and Hadoop 2.0 is illustrated in **the following figure**.



Hadoop 1.0 vs Hadoop 2.0.

5. Overall Significance

1. The introduction of active-standby NameNodes marks a major milestone in Hadoop evolution.
2. Hadoop 2.0 enables:
 - Continuous operation
 - Better support for enterprise-grade workloads
 - Greater confidence in large-scale distributed data processing systems

1 Hadoop 1.0 Limitations

1. Scalability Limitations

1. In Hadoop 1.0, the **JobTracker runs on a single machine**, which creates scalability constraints.
2. The JobTracker is responsible for multiple critical system-level functions, including:
 - **Task scheduling**
 - **Resource management**

- **Tracking and administering task progress**
- **Monitoring the health and status of TaskTrackers**
- 3. Concentrating these responsibilities on a single node limits horizontal scalability and increases system overhead as cluster size grows.

2. Single Point of Failure

1. Both the **JobTracker** and the **NameNode** act as **single points of failure** in Hadoop 1.0.
2. If either component fails:
 - The entire MapReduce job fails.
 - Cluster operation is disrupted.
3. This design significantly reduces system availability and reliability, particularly for long-running or critical workloads.

3. Limitations in Running Applications

1. Hadoop 1.0 is restricted to running **only MapReduce applications**.
2. It supports **only batch-mode processing**.
3. As a result:
 - Interactive applications are not supported.
 - Streaming and real-time data processing workloads cannot be efficiently handled.
4. This limitation reduces the flexibility of Hadoop 1.0 as a general-purpose data processing platform.

4. Imbalance in Resource Utilization

1. In Hadoop 1.0, each TaskTracker is allocated a **fixed number of map slots and reduce slots**.
2. This static slot allocation leads to inefficient resource usage:
 - Map slots may be fully utilized while reduce slots remain idle.

- Conversely, reduce slots may be occupied while map slots are underutilized.
- 3. Resources assigned to a reducer may remain idle even when:
 - There is an immediate need for mapper resources.
- 4. This imbalance results in:
 - Poor cluster utilization
 - Increased job completion time
 - Reduced overall system efficiency

5. Architectural Implications

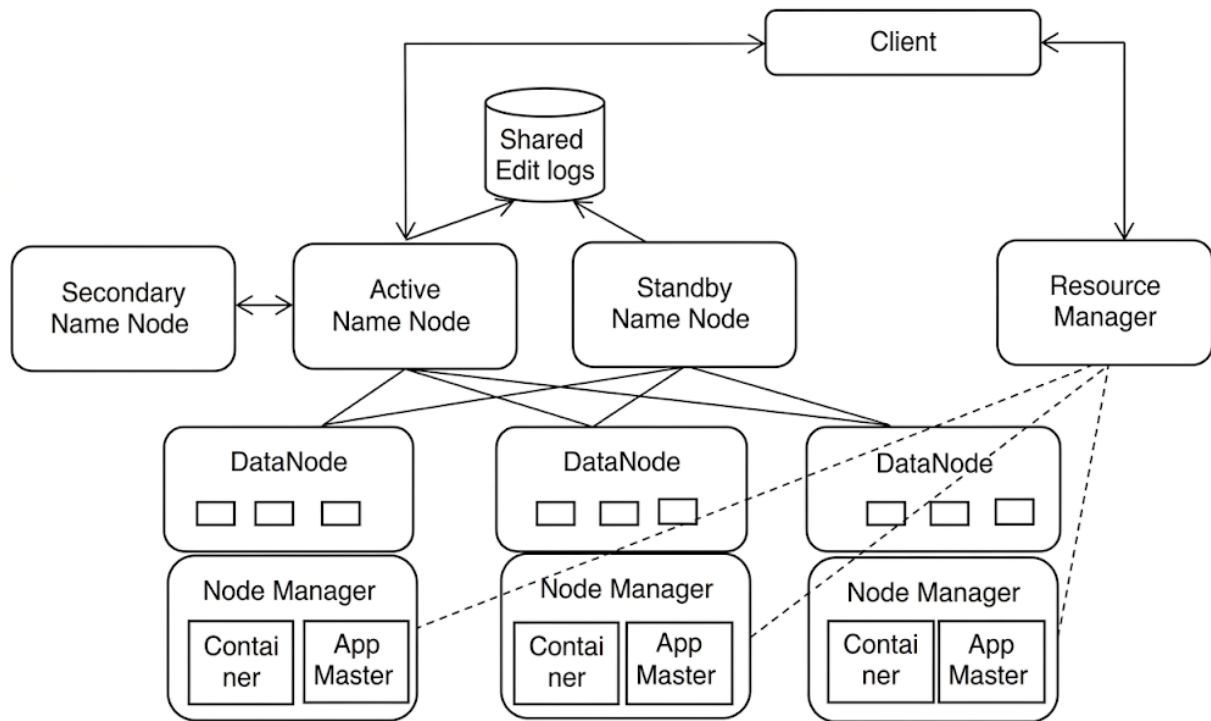
1. The tight coupling of:
 - Job scheduling
 - Resource management
 - Fault monitoringwithin the JobTracker creates performance bottlenecks.
2. These architectural constraints hinder:
 - Scalability to very large clusters
 - Efficient multi-application support
3. The limitations motivated the redesign introduced in **Hadoop 2.0**, particularly the separation of resource management and application execution.

2 Features of Hadoop 2.0

1. High Availability of NameNode

1. The **NameNode** is a critical Hadoop component responsible for storing all filesystem metadata.
2. In Hadoop 1.0, failure of the NameNode caused the **entire Hadoop cluster to become unavailable**.

3. Hadoop 2.0 addresses this critical issue by introducing **NameNode high availability**.
4. The cluster runs **two NameNodes**:
 - **Active NameNode**
 - **Standby NameNode**
5. Under normal operation:
 - The Active NameNode handles all client requests.
6. In the event of Active NameNode failure:
 - The Standby NameNode automatically assumes the role of the Active NameNode.
7. This mechanism ensures:
 - Continuous service
 - Improved fault tolerance
 - Enhanced system reliability
8. The active–standby NameNode configuration is illustrated in **the following figure**:



Active NameNode and standby NameNode.

2. Support for Non-MapReduce Applications

1. Hadoop 1.0 is limited to running **only MapReduce jobs** for processing data stored in HDFS.
2. To use other processing paradigms in Hadoop 1.0:
 - Data had to be transferred to external systems such as **HBase** or **Cassandra**.
3. Hadoop 2.0 introduces **YARN (Yet Another Resource Negotiator)**.
4. YARN enables Hadoop to:
 - Run **non-MapReduce applications** directly on the Hadoop cluster.
5. Examples of frameworks supported on Hadoop 2.0 include:
 - **Apache Spark**
 - **Giraph**
 - **Hama**

6. This feature significantly expands Hadoop's applicability beyond traditional MapReduce workloads.

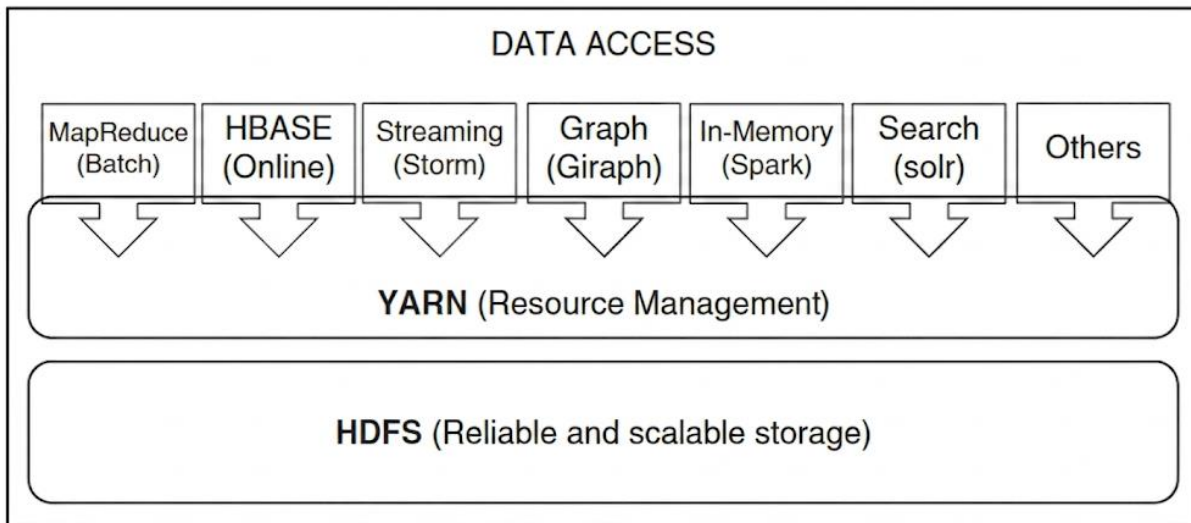
3. Improved Resource Utilization

1. In Hadoop 1.0:
 - Resource management and task monitoring are handled solely by the **JobTracker**.
2. Hadoop 2.0 improves resource utilization by **decoupling JobTracker responsibilities**.
3. YARN divides JobTracker functionality into two independent components:
 - **Global Resource Manager**
 - Responsible for cluster-wide resource allocation and management.
 - **Per-Application ApplicationMaster**
 - Responsible for job scheduling and application-level monitoring.
4. This separation leads to:
 - Better scalability
 - More efficient resource usage
 - Support for multiple concurrent applications

4. Beyond Batch Processing

1. Hadoop 1.0 is restricted to **batch-oriented processing**.
2. Hadoop 2.0 extends Hadoop's capabilities to support:
 - **Real-time applications**
 - **Near real-time applications**
3. This enhancement allows Hadoop 2.0 to be used for:
 - Interactive analytics
 - Streaming workflows

- Low-latency processing tasks
4. The overall Hadoop 2.0 architecture is illustrated in **the following figure**.



Hadoop 2.0.

5. Architectural Significance

1. Hadoop 2.0 represents a major architectural evolution by:
 - Eliminating single points of failure
 - Improving cluster utilization
 - Supporting diverse application frameworks
2. These features position Hadoop 2.0 as a **general-purpose distributed data processing platform** rather than a MapReduce-only system.

3 Yet Another Resource Negotiator (YARN)

1. Motivation for YARN

1. The YARN architecture was developed to **overcome the drawbacks of the Hadoop MapReduce architecture**, particularly its scalability and performance limitations.
2. In Hadoop 1.0, the **JobTracker** handled both:
 - Resource management

- Job scheduling
- 3. This centralization led to bottlenecks and restricted scalability as the cluster size increased.

2. Fundamental Architectural Change

1. Hadoop YARN **splits the responsibilities of the JobTracker** to improve performance and scalability.
2. Resource management and job scheduling are no longer managed by a single component.
3. Each job submitted to the cluster is provided with its own **ApplicationMaster**, enabling better isolation and parallelism.

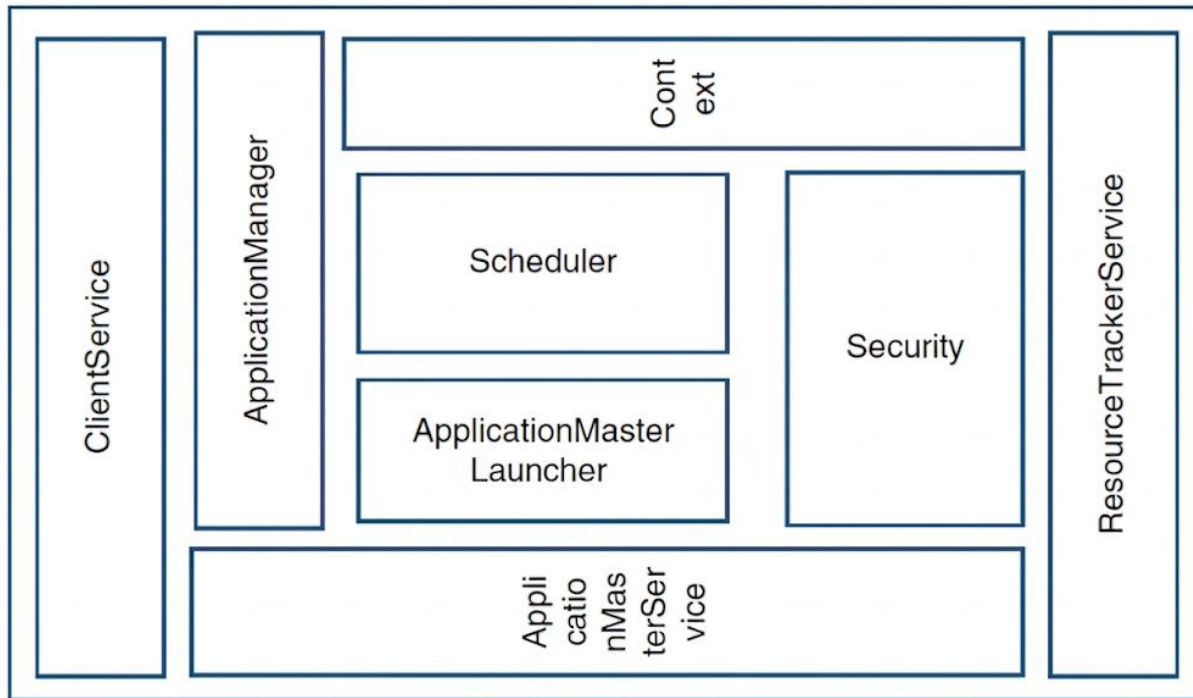
3. Purpose and Evolution of YARN

1. The primary purpose of YARN's evolution is to support **multiple data processing models**, not just MapReduce.
2. YARN enables Hadoop to efficiently run diverse frameworks, including:
 - **Apache Storm**
 - **Apache Spark**
 - **Apache Giraph**
3. This architectural evolution transforms Hadoop into a **general-purpose distributed processing platform**.

4. Core Components of YARN

YARN divides the responsibilities of the JobTracker into two major daemons:

4.1 ResourceManager



ResourceManager.

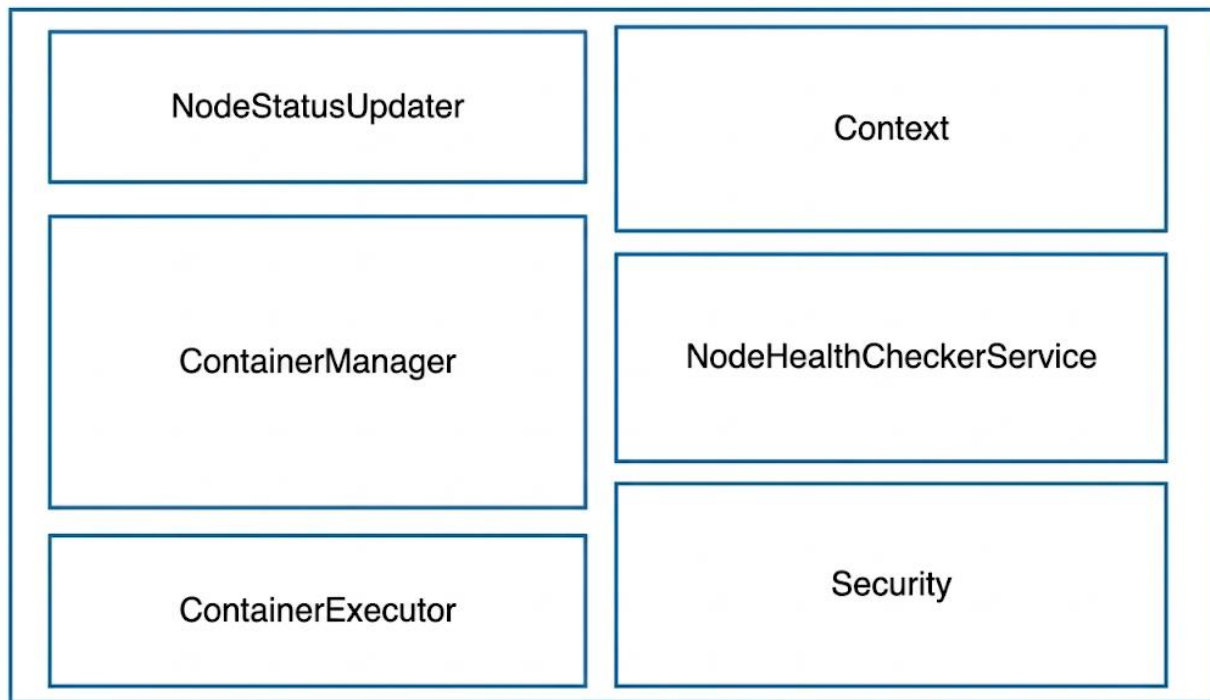
1. The **ResourceManager** is a **cluster-level component**.
2. It is responsible for:
 - Global **resource management**
 - Allocation of resources across all applications running in the cluster
3. It ensures fair and efficient utilization of cluster resources.

4.2 ApplicationMaster

1. Each application launched on YARN has its own **ApplicationMaster**.
2. The ApplicationMaster is **application-specific** and is responsible for:
 - Job scheduling
 - Monitoring execution progress
 - Handling application-level fault recovery
3. It negotiates required resources with the ResourceManager.

5. Replacement of JobTracker and TaskTracker

1. In the YARN architecture:
 - The **JobTracker** is replaced by:
 - ResourceManager (resource management)
 - ApplicationMaster (job scheduling and monitoring)
2. The responsibilities of the **TaskTracker** are effectively assumed by the:
 - ApplicationMaster
 - **NodeManager**, which executes tasks on individual nodes
3. Thus:
 - JobTracker → ResourceManager + ApplicationMaster
 - TaskTracker → NodeManager (working under ApplicationMaster control)



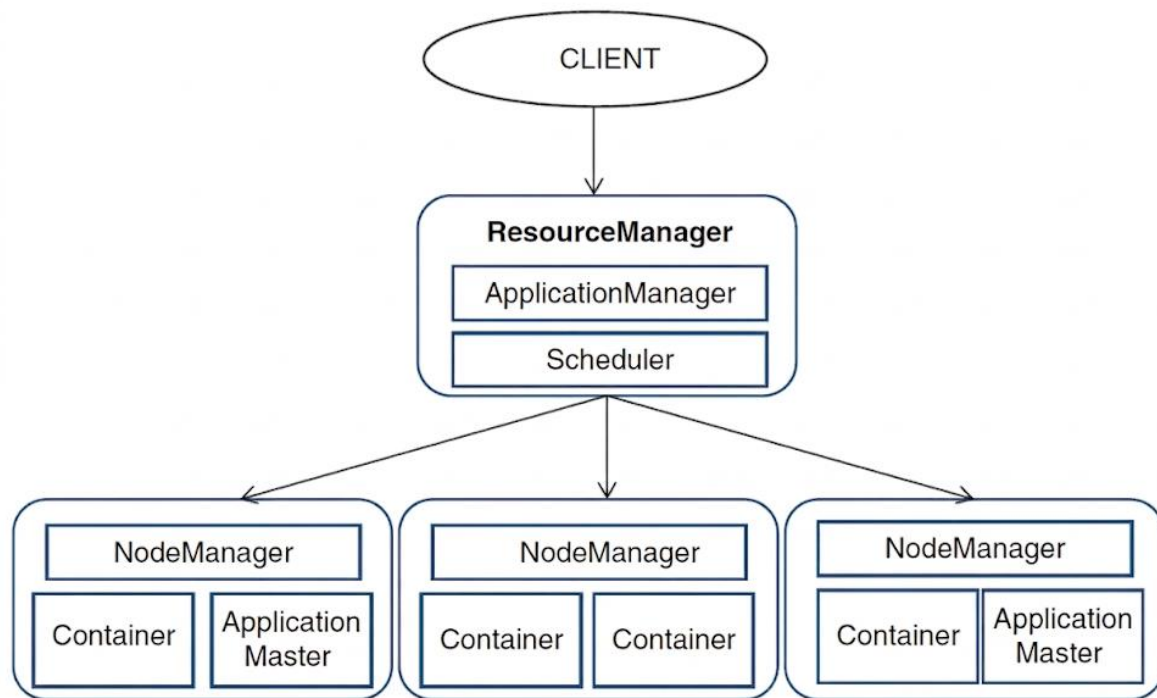
NodeManager.

6. Interaction with NodeManager

1. The ApplicationMaster works closely with the **NodeManager** on each node.

2. The NodeManager is responsible for:
 - Launching and monitoring containers
 - Executing application tasks as instructed
3. Resource negotiation occurs between the ApplicationMaster and the ResourceManager, while execution occurs via NodeManagers.

7. Architectural Significance of YARN



YARN architecture.

1. YARN introduces:
 - Better **scalability**
 - Improved **resource utilization**
 - Enhanced **multi-tenancy**
2. It decouples resource management from application logic.
3. This design enables multiple applications to **coexist and run concurrently** on the same Hadoop cluster.

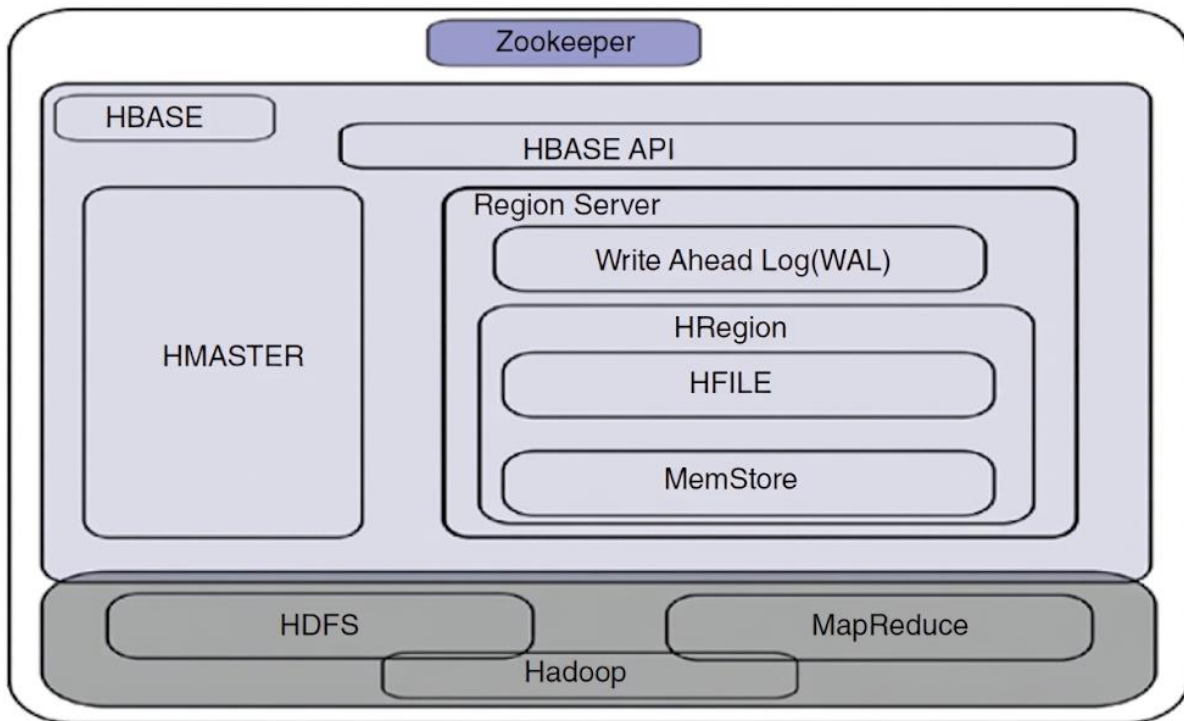
8. Summary

- YARN represents a major architectural shift in Hadoop.
- By decomposing the JobTracker into:
 - ResourceManager
 - ApplicationMasterYARN eliminates bottlenecks and enables support for diverse processing frameworks.
- As a result, Hadoop evolves from a MapReduce-centric system into a **flexible, scalable, and extensible big-data platform**.

HBase

- **HBase** is a **NoSQL database** (not like regular relational databases).
- It is **built on top of Hadoop's storage system (HDFS)**.
- It stores data in a **column-based format**, not rows like traditional databases.
- Data is saved as **key-value pairs**.
- It is **distributed**, meaning data is spread across many machines.
- This allows **horizontal scaling** (you can add more servers easily as data grows).
- It supports **both structured and unstructured data**.
- It **does not require a fixed schema**, so data format can be flexible.
- Unlike traditional Hadoop:
 - Hadoop processes data **sequentially (batch processing)**.
 - HBase allows **real-time and random access** to data.
- Because of this, HBase:
 - **Reduces latency** (faster data retrieval).
 - Is good for **applications that need quick reads and writes**.
- It supports **frequent read and write operations**.
- This is different from Hadoop's model of **write-once, read-many**.
- HBase is inspired by **Google's Bigtable** (its design is based on it).

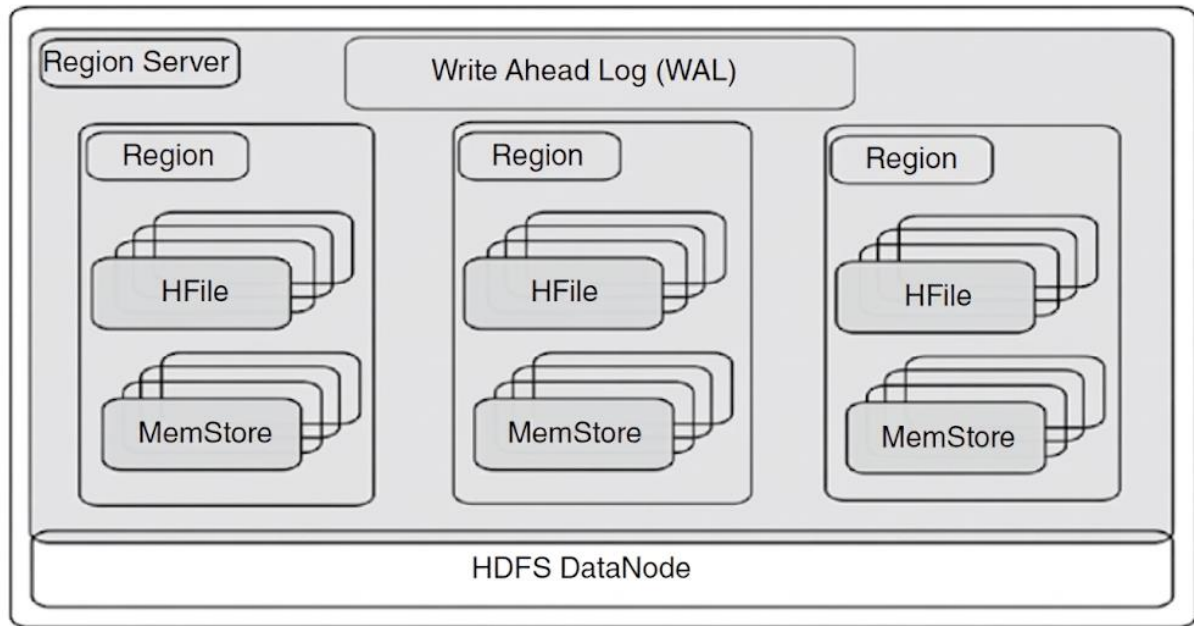
1. Architecture



HBase architecture.

HBase follows a master–slave architecture consisting of several core components:

- **HMaster:** The master node responsible for coordinating the cluster. It manages metadata, region allocation, load balancing, automatic sharding, and failover handling. Although multiple HMaster instances may exist for fault tolerance, only one remains active at a time.
- **RegionServer:** Slave nodes that store and manage actual data. Each RegionServer hosts multiple regions and handles client read/write requests.



RegionServer architecture.

- **Region:** A horizontal partition of a table. Tables are split into regions and distributed across RegionServers for scalability.
- **ZooKeeper:** A coordination service that manages synchronization and communication between distributed components, ensuring system reliability.

2. Storage Components

Each region consists of three main storage structures:

- **Write-Ahead Log (WAL):** A logging mechanism that records data changes before they are applied. This ensures data recovery in case of system failures.
- **MemStore:** An in-memory buffer where incoming data is temporarily stored before being flushed to disk. Although fast, it is volatile, which necessitates the use of WAL for durability.
- **HFile:** The permanent storage format on HDFS, where data is stored in blocks. When MemStore reaches capacity, data is written to HFile on disk.

3. Data Management and Reliability

Data is first written to the WAL and MemStore, then eventually flushed to HFiles. This design ensures both performance and fault tolerance. HBase itself does not handle data replication; instead, it relies on HDFS, which provides built-in replication, fault tolerance, and scalability.

4. Applications

HBase is widely used in domains that require real-time processing of large datasets, such as healthcare, sports analytics, web services, and e-commerce.

5. Conclusion

In summary, HBase extends Hadoop's capabilities by enabling efficient, low-latency, random access to massive datasets. Its distributed architecture, combined with strong consistency and fault-tolerant storage mechanisms, makes it suitable for large-scale, real-time data applications.

6. Features of HBase

HBase provides several key features that enable efficient storage and processing of large-scale distributed data:

- **Automatic Failover:** HBase ensures high availability through RegionServer replication. In case of server failure, the system automatically redistributes data handling responsibilities to maintain continuity.
- **Auto Sharding:** Data is horizontally partitioned into regions. Initially, a table contains a single region, but as data grows and reaches a predefined size threshold, it is automatically split into smaller regions. These regions are distributed across multiple RegionServers, enabling efficient load balancing.
- **Horizontal Scalability:** HBase supports scale-out architecture, allowing additional nodes to be dynamically added to the cluster. This eliminates the need for expensive hardware upgrades and ensures cost-effective expansion using commodity hardware.
- **Column-Oriented Storage:** Unlike traditional row-oriented relational databases, HBase stores data in column families. This structure improves performance for

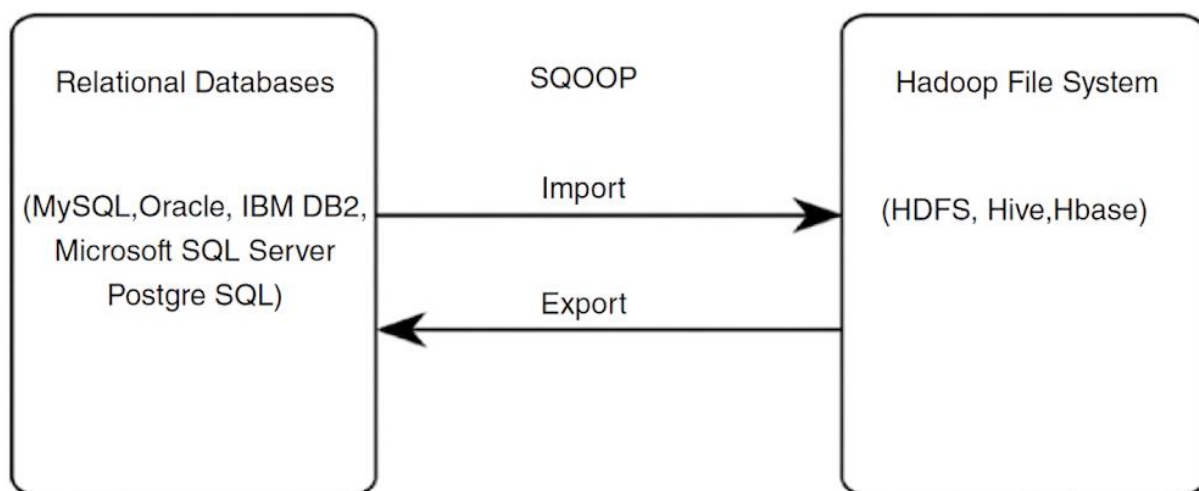
sparse datasets and analytical queries by organizing data by columns rather than rows.

- **Integration with HDFS and MapReduce:** HBase commonly operates on HDFS, leveraging its fault tolerance, replication, and scalability. It also integrates with the MapReduce framework to support parallel data processing, while its pluggable architecture allows compatibility with other file systems.

SQOOP

SQOOP (SQL-to-Hadoop) is a data transfer tool designed to efficiently move large volumes of structured data between relational database management systems (RDBMS) and the Hadoop ecosystem. It is primarily used when traditional RDBMS solutions cannot handle large-scale data processing requirements, enabling seamless integration between external databases and HDFS.

1. Core Functionality



SQOOP import and export.

SQOOP supports two main operations:

- **Import:** Transfers data from relational databases (such as MySQL, Oracle, SQL Server, or PostgreSQL) into Hadoop systems, including HDFS, Hive, and HBase.

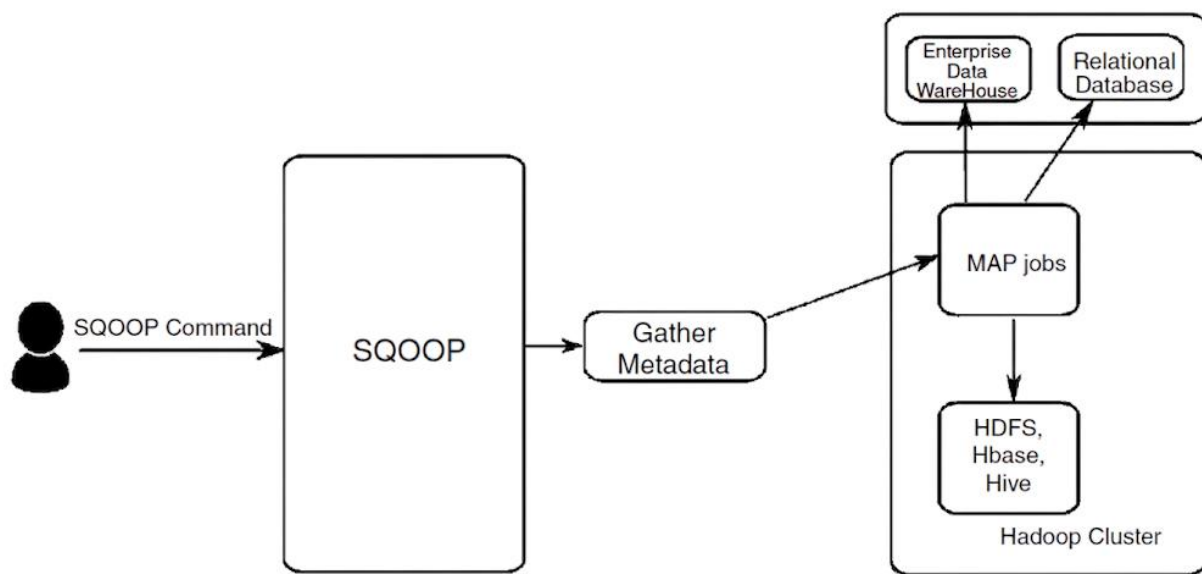
- **Export:** Transfers processed data from Hadoop back to relational databases for further use.

Data is imported row by row from database tables and stored in distributed files within HDFS. After processing in Hadoop, the results can be exported back into structured database tables.

2. Data Processing Mechanism

SQOOP leverages the MapReduce framework to enable **parallel data transfer**, significantly improving performance. Data extraction from RDBMS occurs in parallel, resulting in multiple output files (e.g., text files or Avro format). During export, these files are read, split into records, and inserted into database tables.

3. Architecture and Workflow



SQOOP 1.0 architecture.

The SQOOP process follows two main steps:

1. **Metadata Collection:** SQOOP retrieves schema information such as column names and data types from the source database.
2. **Parallel Data Transfer:** Map-only jobs are executed to import or export data efficiently across the Hadoop cluster.

The architecture consists of:

- A **SQOOP interface** that accepts user commands.
- A **metadata gathering phase**.
- **MapReduce jobs** for distributed data movement.
- Integration with Hadoop storage systems (HDFS, Hive, HBase) and external relational databases.

4. Key Characteristics

- Enables bidirectional data transfer between Hadoop and RDBMS.
- Uses parallel processing for high-speed data movement.
- Supports multiple data formats, including text and binary (Avro).
- Treats each database row as a record during import and export.
- Allows updates to specific rows during export by specifying column mappings.

5. Conclusion

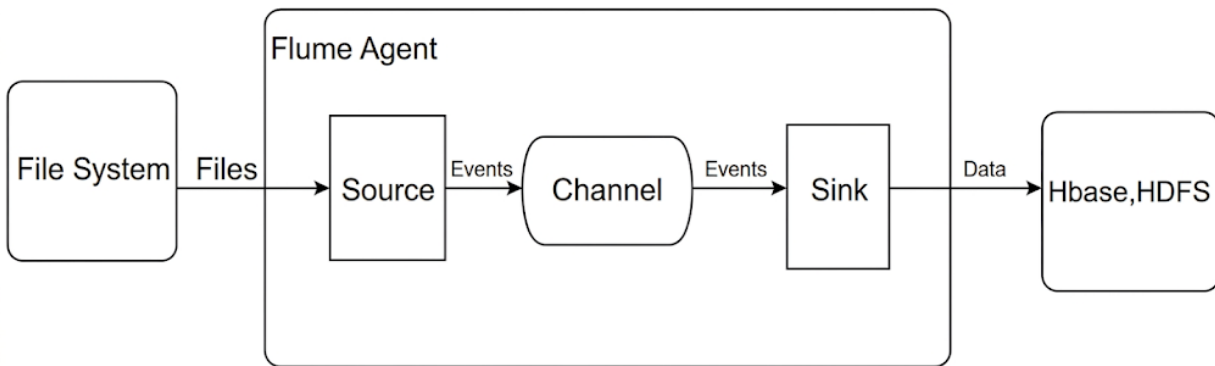
SQOOP serves as a critical bridge between traditional relational databases and the Hadoop ecosystem. By enabling efficient, large-scale, and parallel data transfer, it facilitates data integration, preprocessing, and analytics, thereby enhancing the overall capability of big data systems.

Flume

- **Apache Flume** is a tool used to **collect and move large amounts of data**.
- It is:
 - **Distributed** (runs on multiple machines)
 - **Reliable** (ensures data is delivered)
 - **Fault-tolerant** (keeps working even if some parts fail)
- Its main job is to:

- **Collect**
 - **Combine (aggregate)**
 - **Transport data**
- It handles **streaming data**, such as:
 - Logs (e.g., server logs)
 - Sensor data
 - Social media feeds
- It sends this data into **Hadoop systems**, like:
 - **HDFS**
 - **HBase**
- Compared to **Sqoop**:
 - **Sqoop** → works with **structured, batch data**
 - **Flume** → works with **continuous streaming data**
- It supports:
 - **Real-time data ingestion**
 - **Parallel processing** (multiple processes at once)
 - **Scalability** (can handle growing data easily)
- Overall, Flume is best for:
 - **Moving live/continuous data into Hadoop efficiently**

1. Flume Architecture



Flume architecture.

Flume operates based on a **data flow model** in which data is represented as events and transferred through agents. The core components include:

- **Event:** The fundamental unit of data in Flume, representing a piece of information flowing from a source to a destination.
- **Agent:** A process that manages the data flow. Each agent consists of three main components:
 - **Source:** Receives data from external systems (e.g., log files, sensors) and generates events.
 - **Channel:** Acts as a temporary storage buffer that holds events until they are processed further.
 - **Sink:** Delivers the events from the channel to the final destination, such as HDFS or HBase.

Data flows from the Source to the Channel and then to the Sink within an agent, and multiple agents can work together for large-scale data handling.

2. Channel Types

Flume supports two types of channels:

- **Memory Channel:** Offers high throughput due to in-memory storage but does not guarantee durability, as data may be lost during failures.

- **Disk Channel:** Provides persistent storage, ensuring data recovery in case of failure, though with lower performance compared to memory channels.

3. Data Transfer Mechanism

Data movement in Flume is **transactional and reliable**, involving two separate transactions:

1. From Source to Channel
2. From Channel to Sink

A transaction is completed only when data is successfully written to the Channel. In case of failure, the transaction is rolled back, and the data remains in the Channel for later processing, ensuring no data loss.

4. Key Characteristics

- Designed for **high-volume streaming data ingestion**
- Supports **fault tolerance and recovery mechanisms**
- Enables **parallel and distributed processing**
- Provides **flexible architecture** to integrate multiple data sources and destinations
- Supports data transfer to **HDFS, HBase, and other systems**

5. Conclusion

Flume is a robust and scalable data ingestion tool tailored for real-time streaming data in big data environments. Its modular architecture, transactional reliability, and integration with Hadoop components make it an essential component for handling continuous data flows in distributed systems.

Apache Avro

- **Apache Avro** is a system used for **data serialization**.
- **Serialization** means:
 - Converting data into a format (usually binary) to **store or send it**.

- **Deserialization** means:
 - Converting that data back into its **original usable form**.
- Avro is:
 - **Open-source**
 - Designed for **efficient storage and data exchange**
- It is commonly used in **distributed systems** like Hadoop.
- Its main benefits:
 - **Fast and compact data storage**
 - **Efficient data transfer between systems**
- Avro solves a problem in Hadoop:
 - Hadoop had **limited support for different programming languages**
 - Avro allows **data to work across multiple languages easily**
- Overall, Avro helps:
 - **Store data efficiently**
 - **Share data between different systems and languages smoothly**

1. Core Characteristics

- **Language Independence:** Avro supports multiple programming languages, including Java, C, C++, Python, and Perl, enabling seamless data sharing across heterogeneous systems.
- **Schema-Based System:** It uses schemas defined in JSON format to describe the structure of data. These schemas are stored along with the data in Avro files.
- **Portability and Interoperability:** Since schemas are embedded with the data, Avro eliminates the need for prior schema knowledge, facilitating flexible and cross-platform data processing.

2. Schema Structure

An Avro schema defines the structure and organization of data. Key components include:

- **Type:** Specifies the data structure (e.g., record).
- **Namespace:** Identifies the logical grouping or location of the schema.
- **Name:** Uniquely identifies the schema within a namespace.
- **Fields:** Define the attributes of the record, including:
 - **Field Name**
 - **Data Type**

Avro supports both:

- **Primitive (simple) data types:** such as null, string, int, long, float, double, and bytes.
- **Complex data types:** such as records, arrays, enums, maps, unions, and fixed types.

3. Data Representation and Processing

Data in Avro is typically stored in a compact binary format, which improves efficiency in storage and transmission compared to text-based formats. However, schemas are written in JSON, ensuring readability and ease of implementation. Avro is particularly useful in distributed systems where multiple services must exchange data efficiently.

4. Applications

Avro is widely used in big data environments, especially within Hadoop ecosystems, for:

- Efficient data serialization and storage
- Data exchange across different systems and languages
- Remote Procedure Calls (RPC), where schemas are shared between client and server

5. Conclusion

Apache Avro provides a flexible, language-independent, and schema-driven approach to data serialization. Its ability to embed schemas with data and support multiple programming languages makes it a powerful tool for ensuring interoperability and efficiency in large-scale distributed data processing systems.

Apache Pig

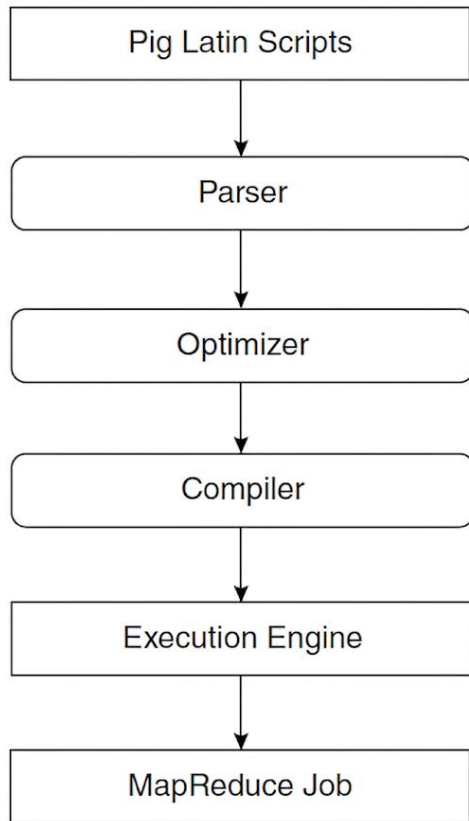
- **Apache Pig** is a tool used to **process and analyze large datasets**.
- It was **developed by Yahoo** for use with **Hadoop**.
- It makes working with big data **simpler and easier** compared to writing complex code.
- Pig has two main parts:
 - **Pig Latin** → a simple **data processing language**
 - **Execution environment** → where the scripts actually run
- **Pig Latin**:
 - Is a **procedural language** (step-by-step instructions)
 - Helps users write **data flow operations easily**
- Pig can handle different types of data:
 - **Structured** (tables)
 - **Semi-structured** (like JSON, XML)
 - **Unstructured** (text, logs)
- Unlike:
 - **Hive or HBase** → mainly focus on structured data
 - **Pig** → is more **flexible and versatile**
- It is useful for:
 - **Data transformation**

- **Data analysis**
 - **Handling large-scale data in Hadoop**
- Overall, Pig helps:
 - **Simplify big data processing**
 - **Work with many types of data efficiently**

1. Core Features

- **Pig Latin Language:** A high-level scripting language designed to simplify complex data transformations and analysis tasks. It significantly reduces the need for extensive Java-based MapReduce programming.
- **Ease of Use:** Users without deep knowledge of Java can perform large-scale data processing using concise Pig Latin scripts, which replace lengthy MapReduce code.
- **Data Flexibility:** Supports multiple data types (structured, semi-structured, and unstructured), enabling broad applicability.

2. Execution Process



Pig – internal process.

Pig Latin scripts are internally translated into MapReduce jobs, which are executed in a Hadoop distributed environment. This transformation is carried out by the **Pig Engine**, and the workflow involves several stages:

1. **Parser**: Verifies the syntax of the Pig Latin script.
2. **Optimizer**: Performs logical optimizations to improve execution efficiency.
3. **Compiler**: Converts optimized scripts into executable MapReduce jobs.
4. **Execution Engine**: Submits and executes the jobs on the Hadoop cluster.

3. Processing Model

The data processing in Pig follows a step-by-step pipeline where scripts are transformed into distributed tasks. Each step ensures correctness, efficiency, and scalability of execution, allowing Pig to handle large-scale data processing effectively.

4. Advantages

- Reduces development time compared to writing raw MapReduce programs
- Provides abstraction over complex distributed processing
- Automatically optimizes execution plans
- Integrates seamlessly with Hadoop components

5. Conclusion

Apache Pig provides a powerful and user-friendly interface for big data processing by abstracting the complexity of MapReduce programming. Its flexible data handling capabilities, combined with an optimized execution framework, make it an essential tool for efficient large-scale data analysis.

Apache Mahout

- **Apache Mahout** is a **machine learning framework** (a tool for building ML applications).
- It is:
 - **Open-source**
 - Designed for **big data environments**
- Its main goal is to help with **scalable data analysis** (handling very large datasets).
- It provides ready-made **machine learning algorithms**, such as:
 - **Clustering** → grouping similar data
 - **Classification** → assigning data to categories
 - **Recommendation systems** → suggesting items (like Netflix or Amazon)
- It is built to:
 - Work efficiently with **large-scale data**
 - Run on **distributed systems** like Hadoop
- It also allows:
 - **Custom algorithm development** (you can create your own models)
- Overall, Mahout helps:
 - **Apply machine learning on big data**
 - **Analyze patterns and make predictions efficiently**

1. Core Characteristics

- **Scalable Machine Learning:** Mahout is specifically designed to handle very large datasets that exceed the capacity of traditional machine learning tools. Its scalability is a key distinguishing feature.
- **Algorithm Support:** It includes built-in algorithms for:
 - Clustering (grouping similar data)
 - Classification (assigning predefined categories)
 - Recommendation (predicting user preferences)
- **Flexibility:** Users can extend Mahout to implement custom machine learning algorithms beyond its standard library.

2. Role in Big Data

Mahout is particularly suitable for applications where datasets are too large to be processed efficiently by conventional tools such as R. It addresses the challenges of the **big data era** by enabling distributed processing and large-scale machine learning computations.

3. Conclusion

In summary, Apache Mahout is a scalable and extensible machine learning framework tailored for big data analytics. Its ability to process massive datasets and support key analytical algorithms makes it a valuable tool for implementing machine learning solutions in distributed environments.

Apache Oozie

- **Apache Oozie** is a tool used to **manage and schedule jobs** in Hadoop.
- It is:
 - **Open-source**
 - **A workflow management system**
- Its main purpose is to:

- **Control and organize multiple jobs**
 - Ensure they run in the **correct order**
- It is useful when:
 - Tasks are **complex**
 - Jobs depend on each other (**interdependent**)
- Oozie allows you to:
 - Create **workflows** (a series of steps/tasks)
 - Define **which job runs first, next, etc.**
- It can manage different types of Hadoop jobs, such as:
 - MapReduce
 - Pig
 - Hive
 - Others
- It ensures:
 - **Automation of tasks**
 - **Proper execution sequence**
- Overall, Oozie helps:
 - **Coordinate multiple jobs**
 - **Automate complex data processing workflows in Hadoop**

1. Core Functionality

- **Oozie** is used to **combine many Hadoop jobs into one workflow.**
- It can manage jobs like:
 - **MapReduce**
 - **Hive**

- **Pig**
 - **Sqoop**
- It helps to:
 - **Organize multiple tasks into one logical process**
- Jobs can run in two ways:
 - **Sequentially** → one after another
 - **In parallel** → multiple jobs at the same time
- This flexibility helps:
 - **Save time**
 - **Improve efficiency**
- It gives better:
 - **Control over job execution**
 - **Management of complex data processes**
- Overall, Oozie helps:
 - **Coordinate and optimize large-scale data workflows**

2. Key Features

- **Workflow Management:** Defines and manages job execution order using workflows represented as **Directed Acyclic Graphs (DAGs)**.
- **Scheduling Capability:** Supports execution of jobs **on demand** or at **predefined time intervals**.
- **Job Coordination:** Handles dependencies between jobs and ensures proper execution flow.
- **Multi-Job Integration:** Supports various Hadoop components, enabling unified task execution.

3. Types of Oozie Jobs

1. **Workflow Jobs:**

Represent sequences of actions arranged in DAGs and executed on demand.

2. **Coordinator Jobs:**

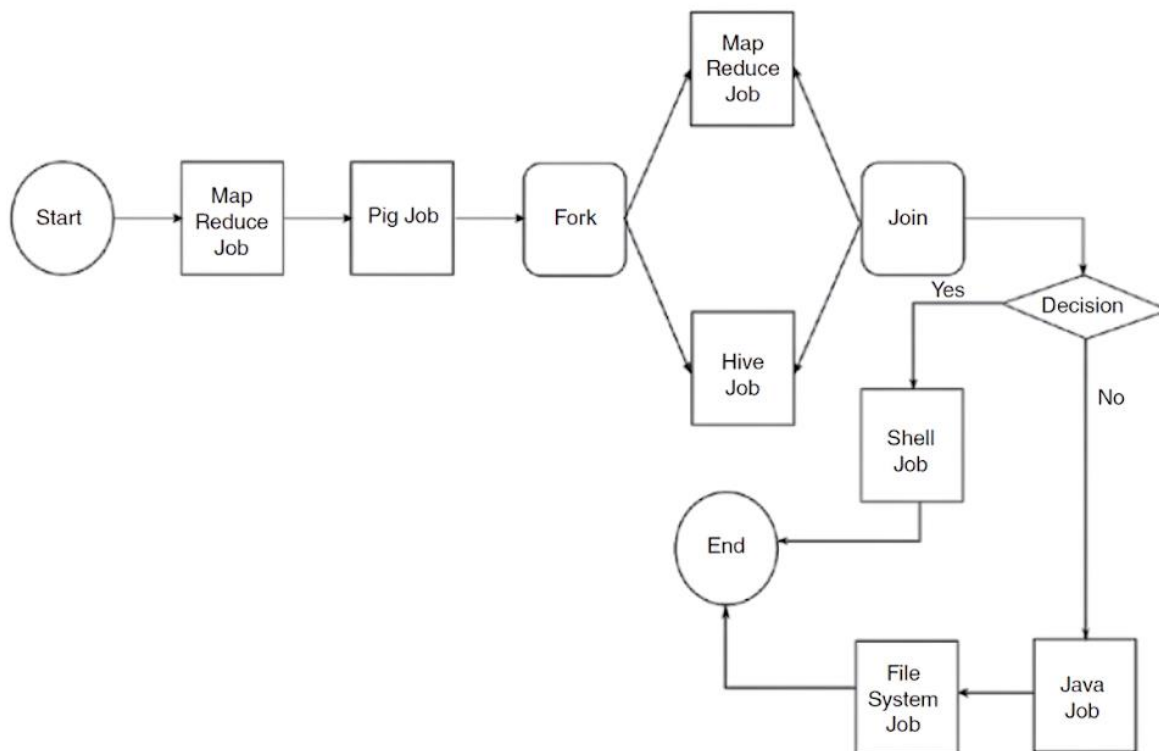
Schedule workflows to run periodically based on time or data availability.

3. **Bundle Jobs:**

Group multiple coordinator jobs into a single manageable unit.

4. Oozie Workflow, Coordinators, and Bundles

Oozie Workflow



Oozie workflow.

An **Oozie workflow** is a structured sequence of dependent tasks represented as a **Directed Acyclic Graph (DAG)**. It consists of multiple stages (actions), which can include Hadoop jobs such as MapReduce, Pig, Hive, and Sqoop, as well as non-Hadoop tasks like email notifications or Java applications. Each action in the workflow follows a

strict dependency order, meaning that a subsequent task cannot begin until its preceding task has successfully completed.

Workflows are composed of two types of nodes:

- **Action nodes**, which define executable tasks.
- **Control nodes**, which govern the execution flow.

The main types of control nodes include:

- **Start and End nodes**: Define the beginning and termination of the workflow.
- **Fork and Join nodes**: Enable parallel execution and synchronization of multiple actions.
- **Decision nodes**: Allow conditional branching based on runtime data.

Workflows may be executed **on demand** or triggered based on time schedules, data availability, or external events.

Oozie Coordinators

Oozie coordinators are responsible for **scheduling workflows**. Unlike workflows, coordinators do not contain execution logic; instead, they trigger workflows based on predefined conditions such as time or data readiness.

Key characteristics include:

- **Time-based scheduling**: Workflows are executed at fixed intervals defined by parameters such as start time, end time, and frequency.
- **Data-driven execution**: Workflows are initiated when required input data becomes available.
- If input data is unavailable at the scheduled time, execution is delayed until the data is ready.

Coordinators require the following parameters:

- Start time and end time
- Execution frequency
- Input data specifications

- Associated workflow

These features make coordinators suitable for recurring tasks such as periodic reporting or batch processing.

Oozie Bundles

Oozie bundles provide a higher-level abstraction by grouping multiple **coordinator jobs** into a single logical unit. Each bundle may contain one or more coordinators, and each coordinator, in turn, may manage one or more workflows.

Bundles are particularly useful for:

- Managing complex pipelines with interdependent workflows
- Coordinating multiple scheduled jobs efficiently
- Handling environments with large numbers of workflows

Conclusion

Oozie extends Hadoop's capabilities by providing a comprehensive framework for **workflow orchestration and scheduling**. Workflows define the execution logic and dependencies of tasks, coordinators manage scheduling based on time and data conditions, and bundles enable higher-level grouping of related jobs. Together, they facilitate efficient, scalable, and automated management of complex data processing pipelines in distributed systems.

Apache Hive

Apache Hive is a data warehouse and query processing tool built on top of the Hadoop ecosystem, designed for analyzing and managing **large-scale structured data**. It provides an SQL-like interface, known as **Hive Query Language (HQL)**, which allows users to perform data analysis without extensive knowledge of low-level MapReduce programming.

1. Core Functionality

Hive interacts with Hadoop through interfaces such as **JDBC and ODBC**, enabling users to submit queries. These queries are parsed and compiled, with the system retrieving metadata from a **metastore**, which stores schema and structural information

about the data. The queries are then converted into execution plans (typically MapReduce jobs) and processed within the Hadoop environment.

2. Key Features

- **SQL-like Language (HQL):** Simplifies data analysis by providing syntax similar to SQL.
- **Structured Data Processing:** Designed specifically for handling structured datasets stored in Hadoop.
- **Schema Management:** Supports Data Definition Language (DDL) operations such as creating, altering, and deleting tables.
- **Multiple Data Types:**
 - Primitive types: int, bigint, float, double, string, boolean, decimal
 - Complex types: array, map, struct, union
- **Platform Flexibility:** Can run on different execution frameworks within Hadoop.

3. Data Organization in Hive

Hive organizes data into three hierarchical levels:

1. Tables

- Similar to relational database tables.
- Stored as directories in HDFS.
- Can be:
 - **Internal tables:** Managed by Hive
 - **External tables:** Refer to existing data in HDFS without managing it

2. Partitions

- Tables are divided into smaller segments based on column values.
- Improves query performance by reducing the amount of data scanned.

- Example: Partitioning a table by “year” allows queries to access only relevant partitions instead of the entire dataset.

3. Buckets

- Further division of partitions based on a hash function applied to a column.
- Organizes data into manageable chunks and enhances query efficiency.
- Useful for operations such as joins and sampling.

4. Performance Optimization

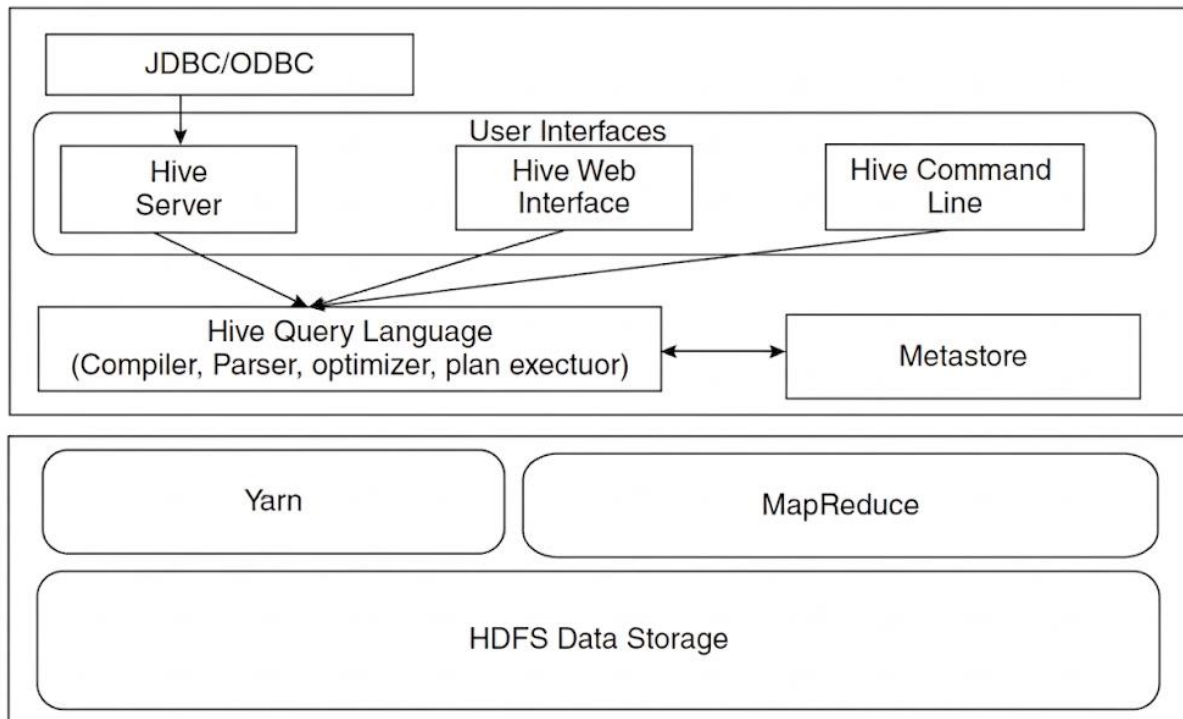
- **Partitioning** reduces query time by limiting data access to relevant subsets.
- **Bucketing** improves processing efficiency by distributing data evenly.

5. Conclusion

Apache Hive provides a powerful abstraction layer over Hadoop for structured data analysis. By offering a familiar SQL-like interface and organizing data into tables, partitions, and buckets, Hive enables efficient querying and management of large datasets. Its integration with Hadoop and ability to translate queries into distributed processing tasks make it a core component of big data analytics systems.

Hive Architecture

Apache Hive follows a layered architecture that enables efficient querying and processing of large-scale data stored in Hadoop. It integrates user interfaces, query processing components, metadata management, and distributed execution frameworks.



Apache Hive architecture

1. Core Components

1. User Interfaces

Hive provides multiple interfaces through which users can submit queries:

- **JDBC/ODBC:** Allow integration with external applications.
- **Hive Server:** Acts as a service layer to handle client requests.
- **Web Interface and Command Line Interface (CLI):** Provide direct user interaction.

2. Hive Query Language (HQL) Layer

HQL is an SQL-like language used to query and manage data. This layer includes several sub-components:

- **Compiler:**
Validates query syntax and requests metadata from the metastore.

- **Parser:**
Converts the query into a **parse tree representation** for further processing.
- **Optimizer:**
Improves the query execution plan for better performance.
- **Plan Executor:**
Executes the optimized plan by translating it into distributed processing jobs.

3. Metastore

The **metastore** is a central repository that stores metadata such as:

- Table schemas
- Data locations in HDFS
- Data types and structure

It provides this information to the compiler and other components for query execution.

4. Execution Framework

Hive executes queries using Hadoop frameworks:

- **MapReduce** (or other engines) performs distributed data processing.
- **YARN** manages cluster resources and job scheduling.

5. Storage Layer

- Data is stored in **HDFS (Hadoop Distributed File System)**.
- Hive tables correspond to directories in HDFS.

2. Query Execution Flow

1. User submits a query via an interface (JDBC/ODBC, CLI, etc.).
2. The **Compiler** checks syntax and retrieves metadata from the **Metastore**.
3. The **Parser** converts the query into an internal representation.
4. The **Optimizer** refines the execution plan.
5. The **Plan Executor** translates the plan into a **MapReduce job**.

6. The job is executed on Hadoop (via YARN).
7. Results are returned to the user interface.

3. Conclusion

Hive architecture provides a structured framework for querying large datasets by combining SQL-like querying with distributed processing. The integration of user interfaces, metadata management, query optimization, and Hadoop execution engines ensures scalability, flexibility, and efficient data analysis in big data environments.

Hadoop Distributions

- **Hadoop distributions** are **pre-packaged versions of Hadoop**.
- They are provided by **different vendors** (companies).
- Their purpose is to:
 - Make Hadoop **easier to install and use**
 - Simplify **deployment and management**
- They include:
 - **Core Hadoop components** (like HDFS, MapReduce, YARN)
 - **Extra tools and services**
- These additions help with:
 - **Better usability**
 - **Improved performance and scalability**
 - **Enterprise-level support**
- They are designed to:
 - Make Hadoop **more practical for businesses**
 - Support **large-scale data processing systems**
- Overall, Hadoop distributions help:

- **Simplify working with Hadoop**
- **Enhance its features for real-world use**

Major Hadoop Distributions

1. Cloudera Hadoop Distribution (CDH)

- **CDH (Cloudera Distribution for Hadoop)** is one of the **earliest and most popular Hadoop distributions**.
- It is designed for:
 - **Enterprise use** (business environments)
 - Providing **support and services** for Hadoop systems
- CDH includes:
 - **Hadoop core components**
 - Additional tools to improve performance and management
- A key feature is **Cloudera Manager**:
 - A **proprietary tool** (owned by Cloudera)
 - Used for **managing and monitoring Hadoop clusters**
- CDH also includes **Impala**:
 - An **open-source query engine**
 - Allows **real-time data querying**
- Compared to **Hive**:
 - **Hive** → slower, batch-style queries
 - **Impala** → much **faster, real-time performance**
- Overall, CDH helps:
 - **Simplify cluster management**
 - **Enable faster data analysis**

- **Support enterprise-level Hadoop usage**

2. Hortonworks Data Platform (HDP)

Hortonworks is an open-source, Apache-licensed Hadoop distribution designed for storing, processing, and analyzing large datasets. It provides stable and up-to-date versions of Hadoop ecosystem components, including:

- HDFS and YARN
- Hive, Pig, and HBase
- Sqoop and Flume
- ZooKeeper and Storm
- Ambari (for cluster management)

HDP emphasizes full open-source integration and enterprise-grade big data processing.

3. MapR

MapR offers a Hadoop-based platform with both free and commercial versions (e.g., M3, M5, M7). Unlike CDH and Hortonworks, MapR is not fully open source. It focuses on **enterprise features** such as:

- High reliability and availability
- Enhanced security
- Real-time processing performance
- Reduced operational costs

Its ecosystem includes components like **MapR-FS**, **MapR-DB**, and **MapR Streams**, which provide advanced data management capabilities.

4. Amazon Elastic MapReduce (EMR)

Amazon EMR is a cloud-based Hadoop distribution that enables scalable data processing on **Amazon Web Services (AWS)**. It distributes data processing tasks across virtual servers, making it highly flexible and cost-effective.

Key applications include:

- **Clickstream Analysis:** Understanding user behavior and preferences
- **Genomics Processing:** Handling large-scale biological data
- **Log Processing:** Analyzing web and application logs

EMR is known for its **ease of use, scalability, security, and low operational cost**, making it suitable for cloud-based big data analytics.

Conclusion

Hadoop distributions play a crucial role in simplifying the deployment and management of big data systems. While CDH and Hortonworks focus on enterprise support and open-source integration, MapR emphasizes performance and reliability, and Amazon EMR provides a scalable cloud-based solution. Together, these distributions enable organizations to efficiently process and analyze large volumes of data across diverse environments.