

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)

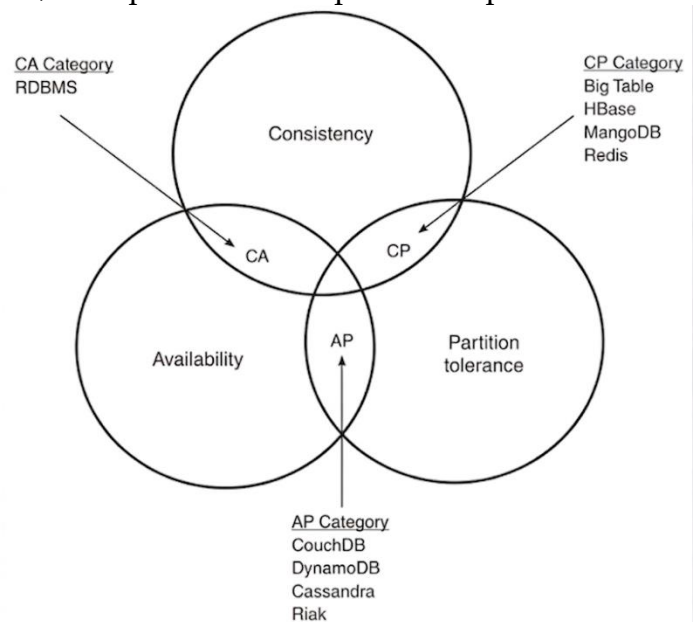
- The system keeps working **even when communication between nodes fails**.
- Even if the network is split into partitions, each part must still process requests.

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

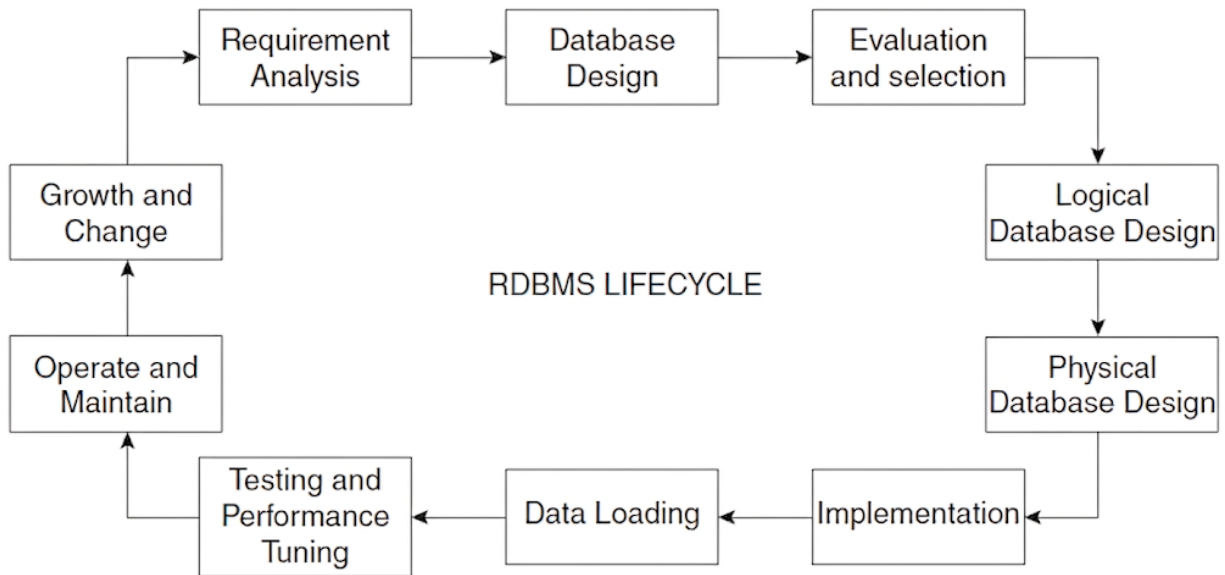
NoSQL databases are designed to achieve either:

- **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.

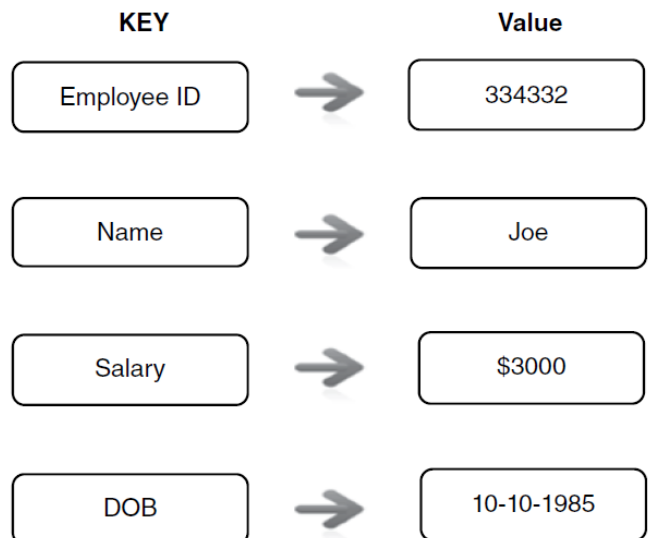
4. Types of NoSQL Databases

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

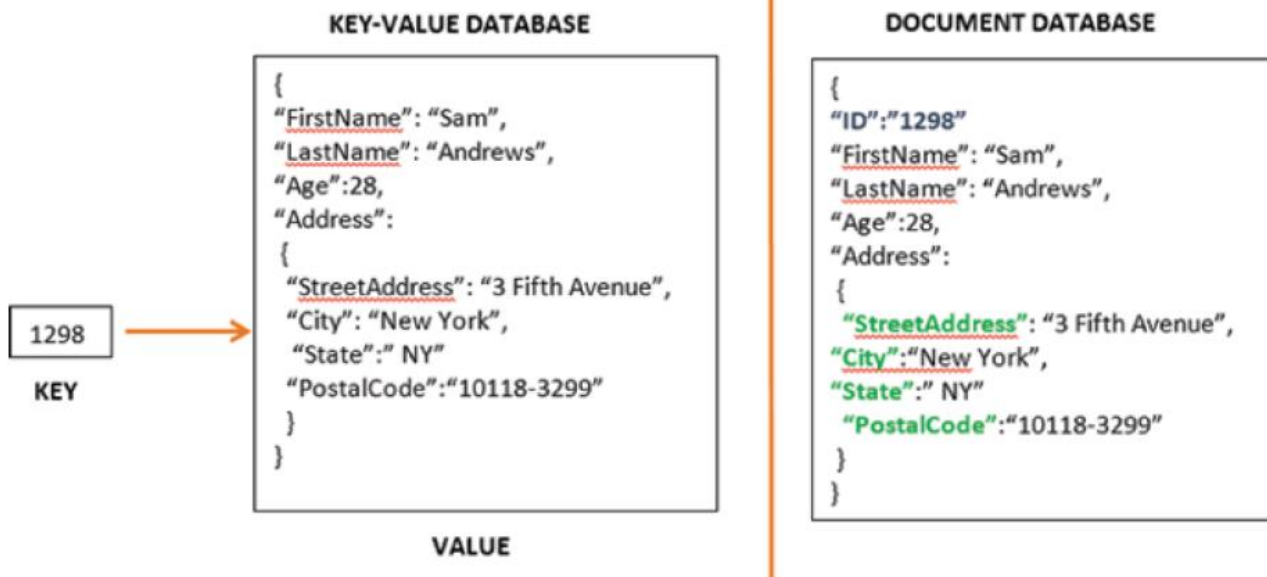
Concept

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**

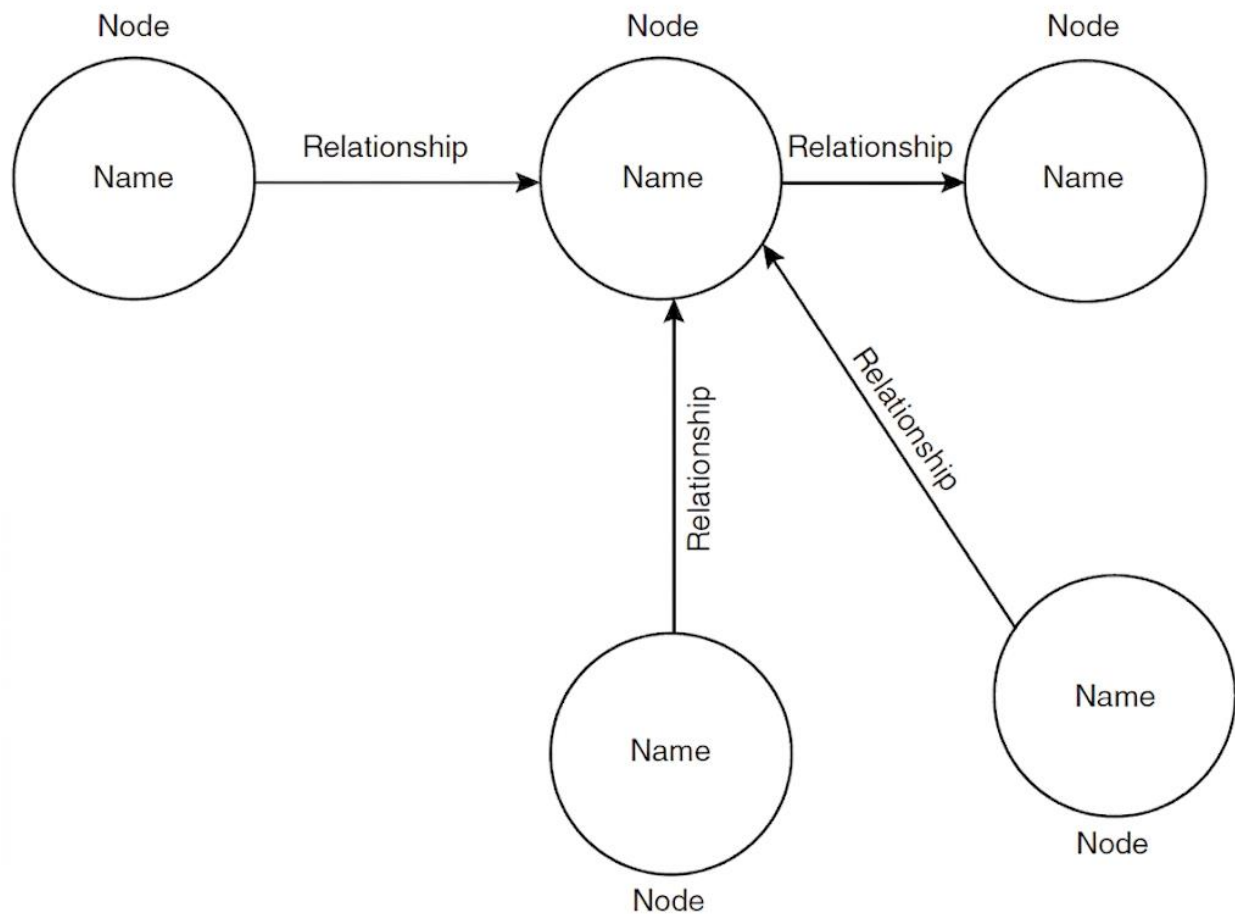
4. Graph Databases

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



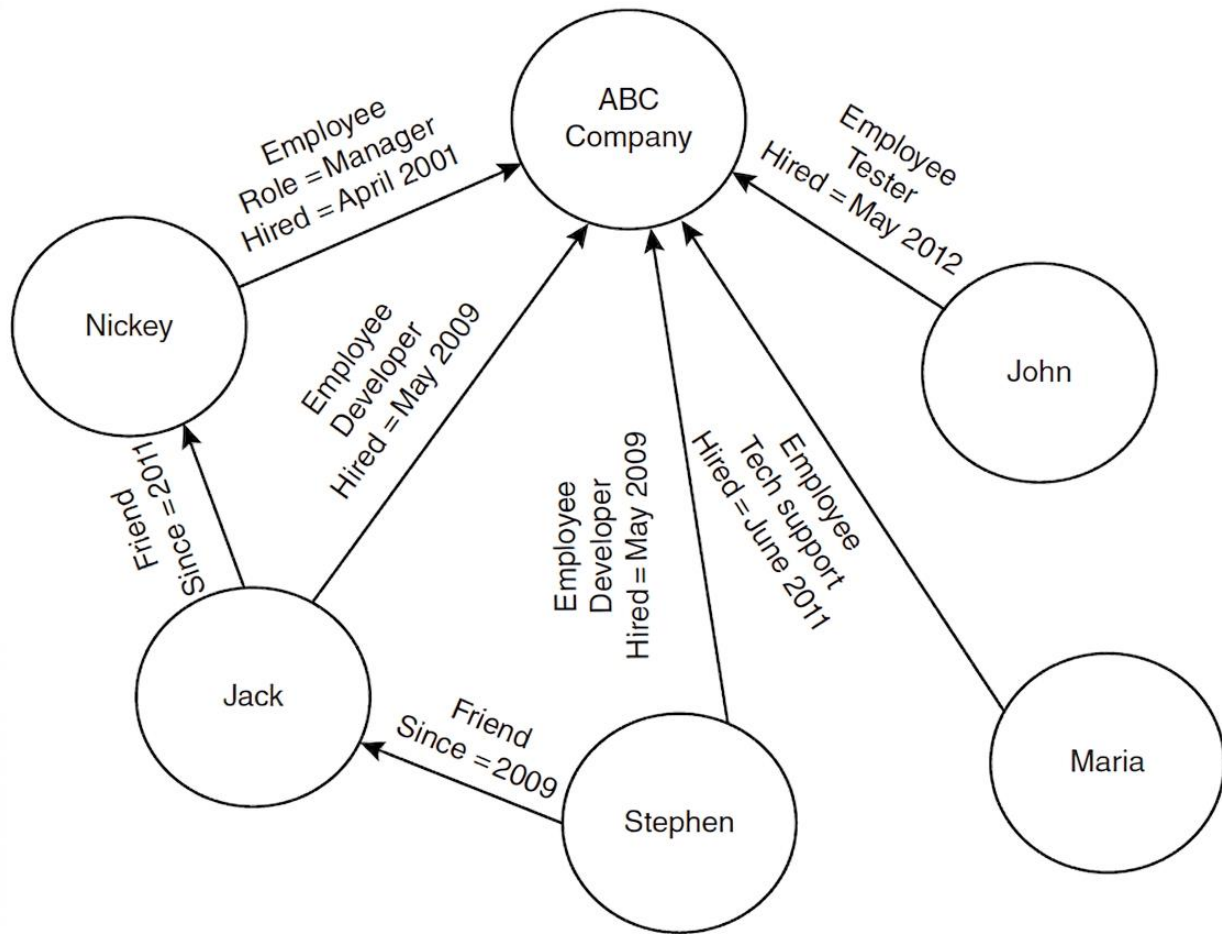
General representation of graph database.

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?

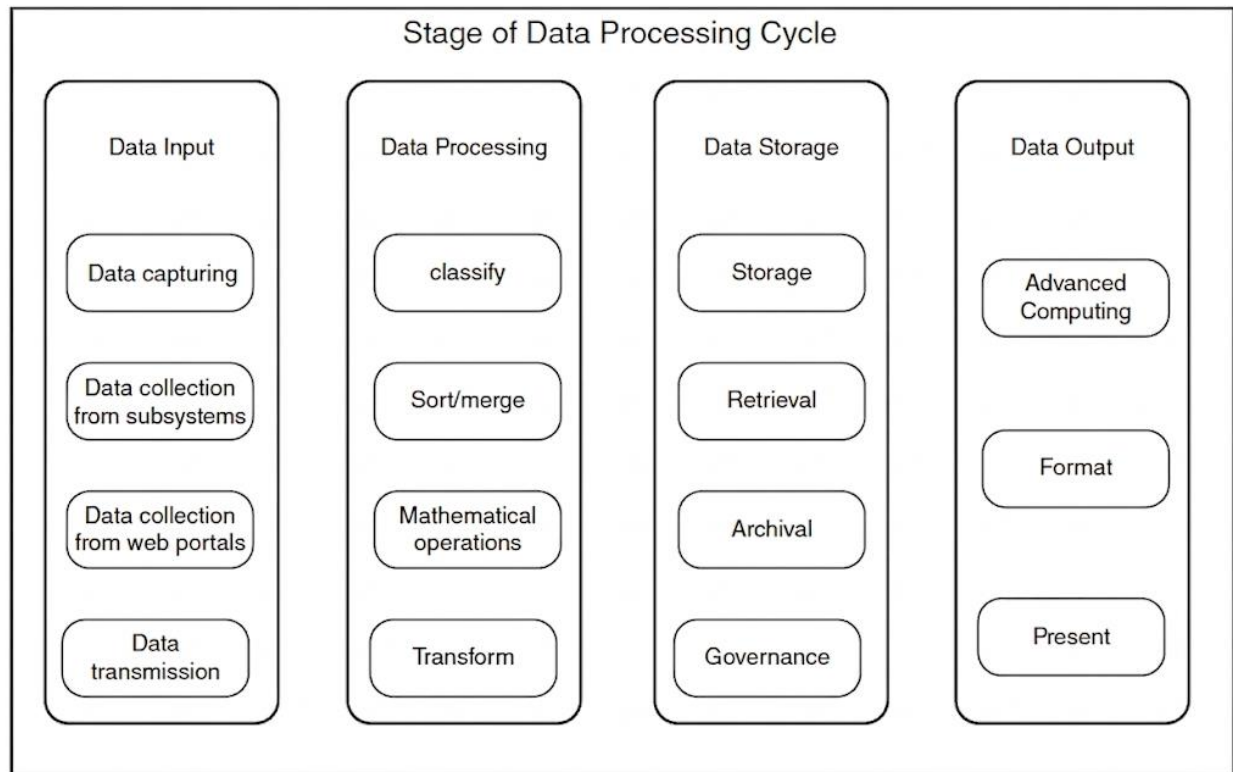
Data may originate from many different sources:

- 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,

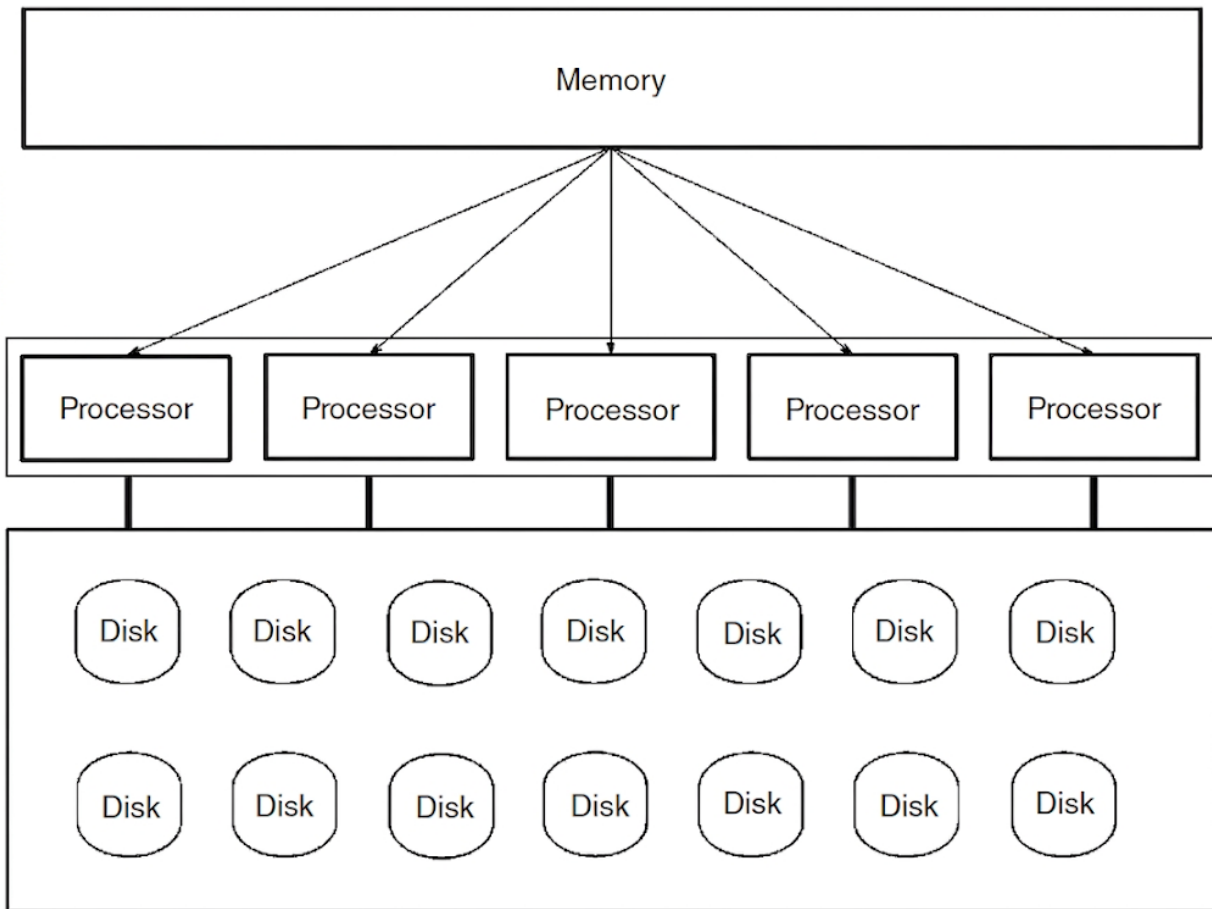
Data processing = turning raw, unorganized data into clear, meaningful information that people or systems can use.

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

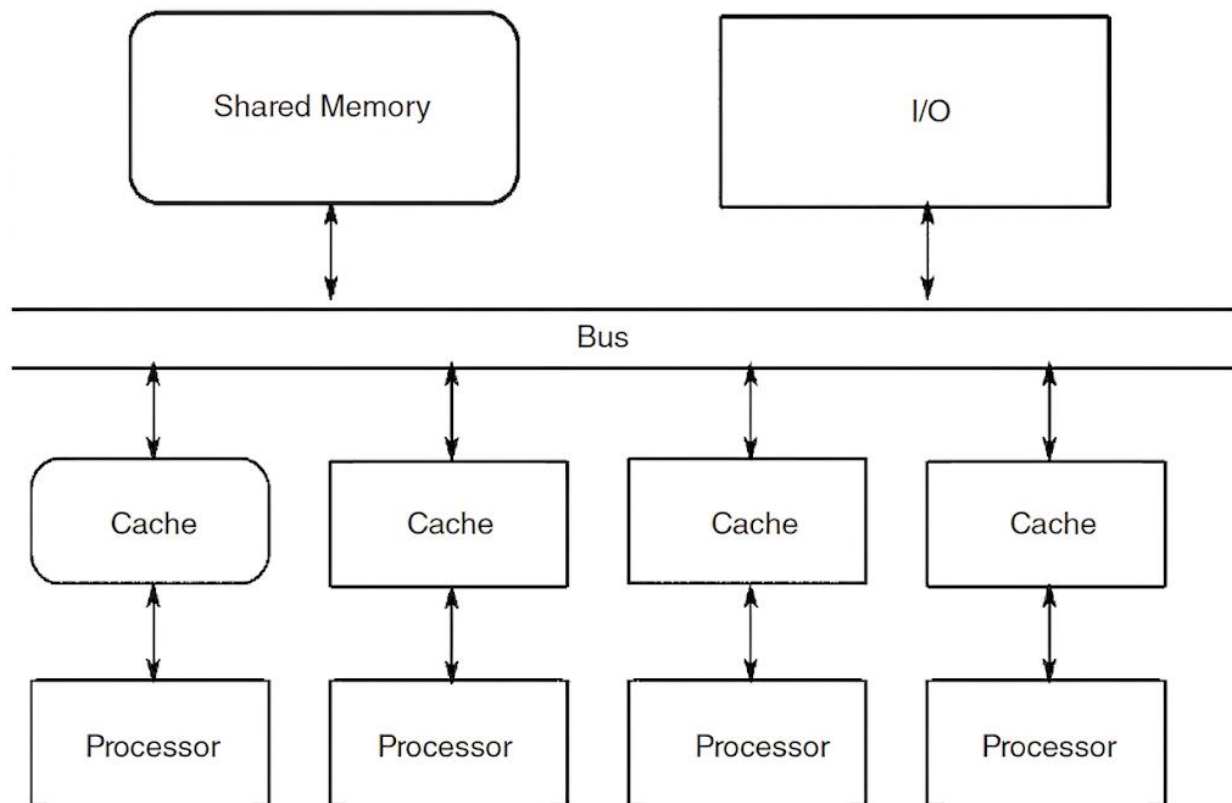
- Data transfer slows down
- Overall responsiveness decreases.)
- Performance degradation (Due to contention and bandwidth limitations, system performance:
 - Stops improving with additional processors
 - Eventually declines in heavy workloads.)

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)

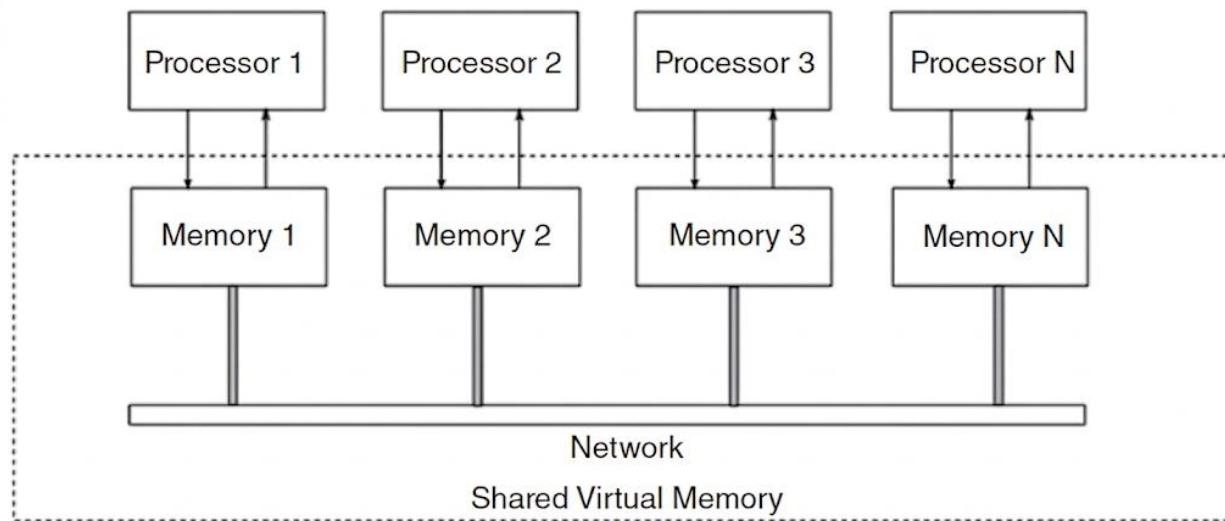
NUMA (Non-Uniform Memory Access) distributes memory across multiple processors, where each processor has its own local memory, but can also access the memory of other processors.

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

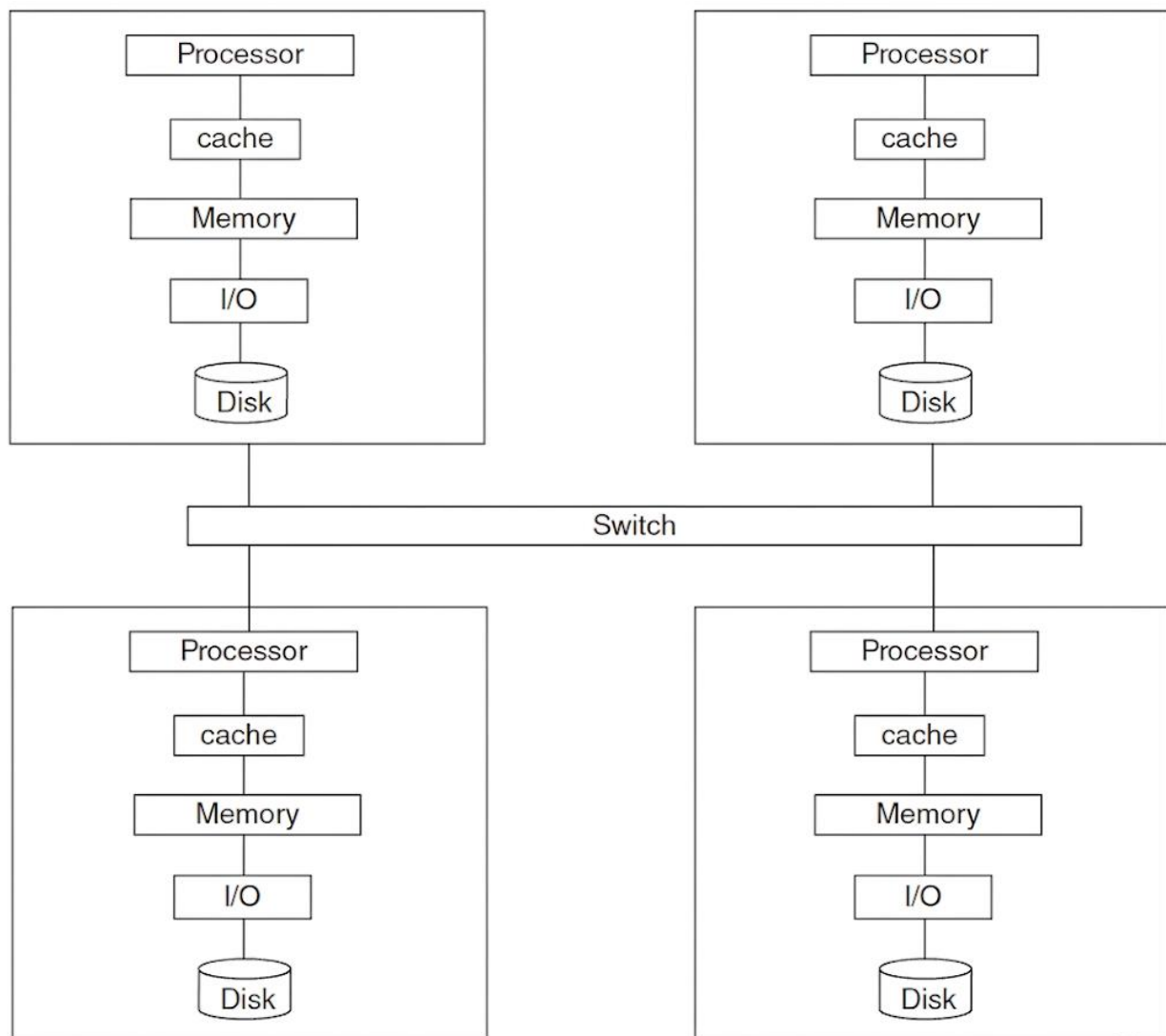
Feature	SMP (UMA)	Distributed Shared Memory (NUMA)
Memory Access	Uniform	Non-uniform
Memory Pools	One	Many
Scalability	Low	Higher
Bottleneck	Shared bus	Reduced contention
Architecture Type	Simple	More complex

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:

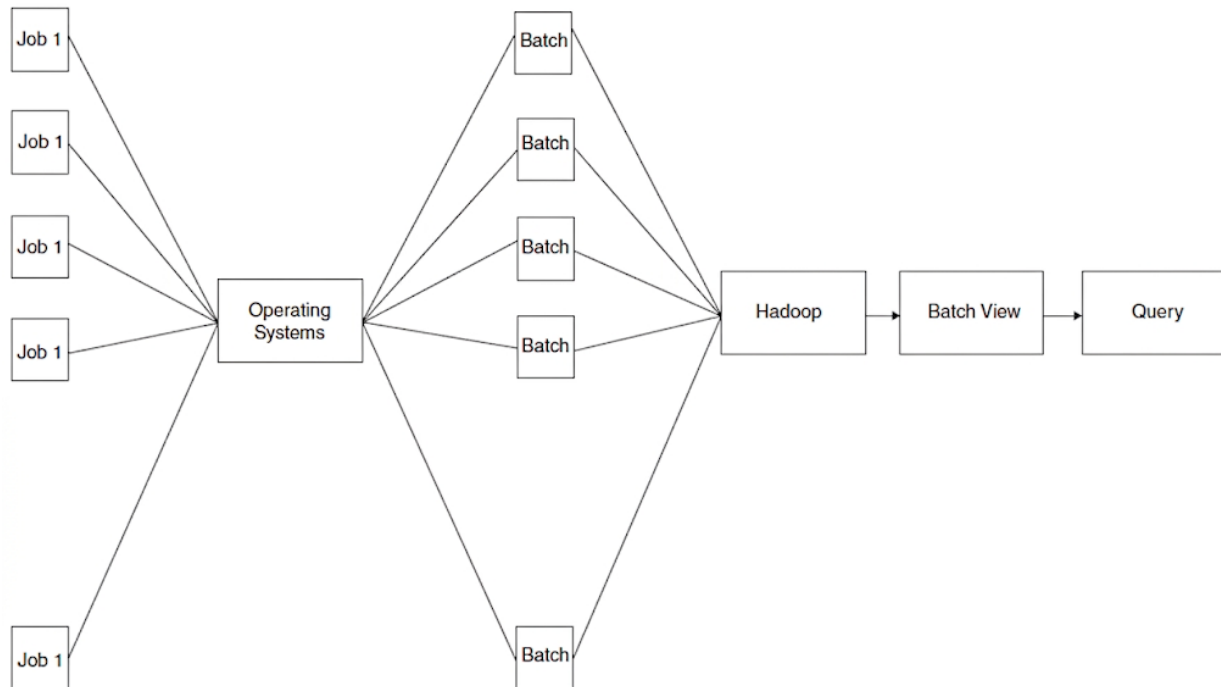
- **Near-infinite scalability**
- **High fault tolerance**
- **Excellent performance for distributed workloads**

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**.

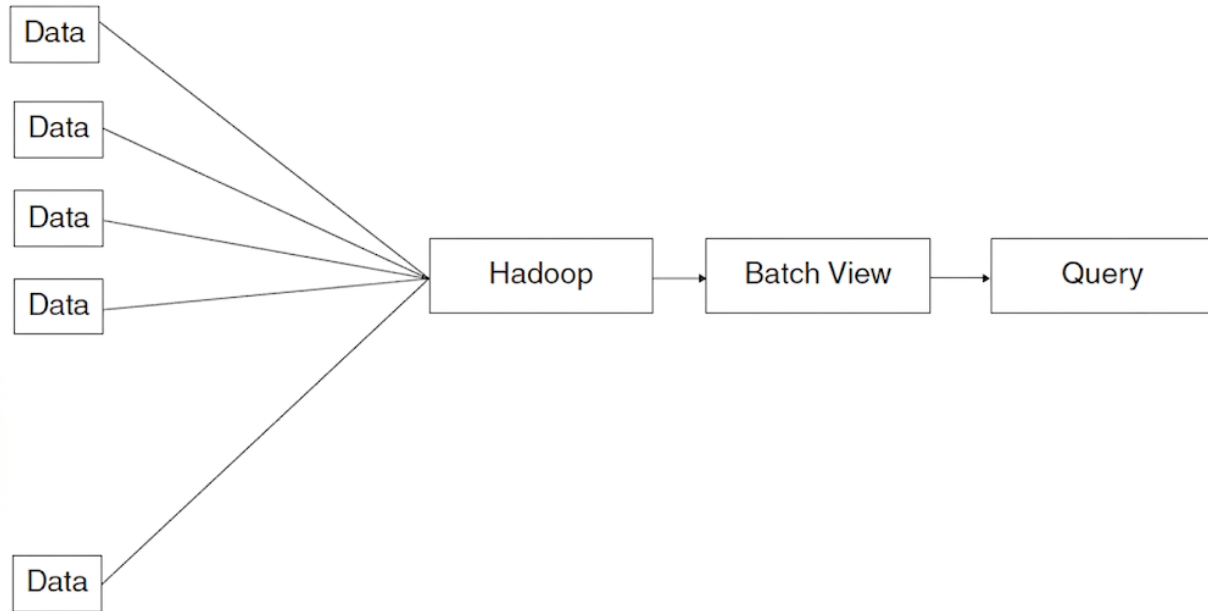
Examples of Real-Time Data Processing

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.

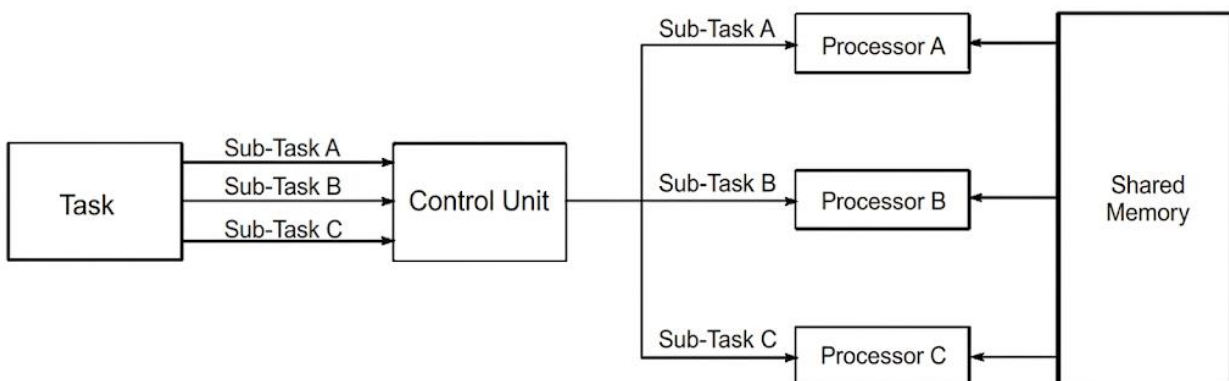
Why Real-Time Processing Matters?

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.

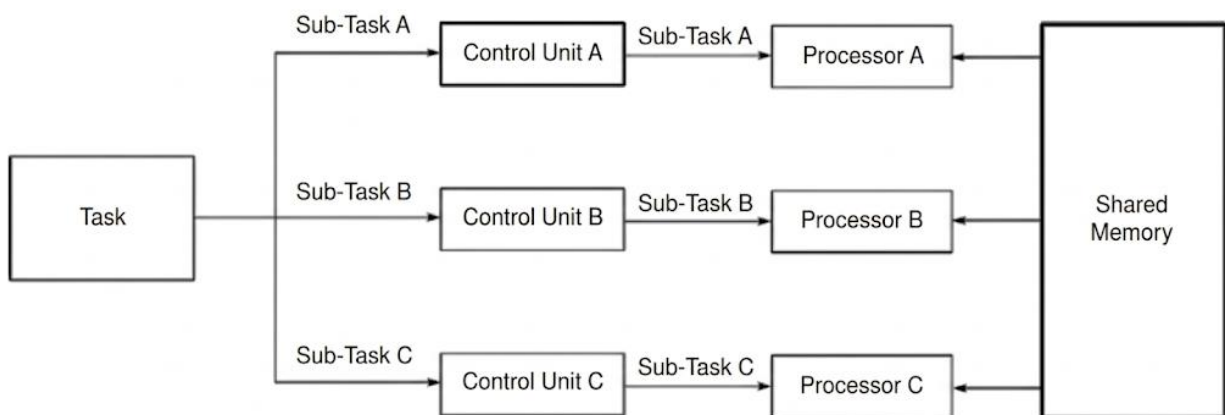
B. Distributed Computing

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:

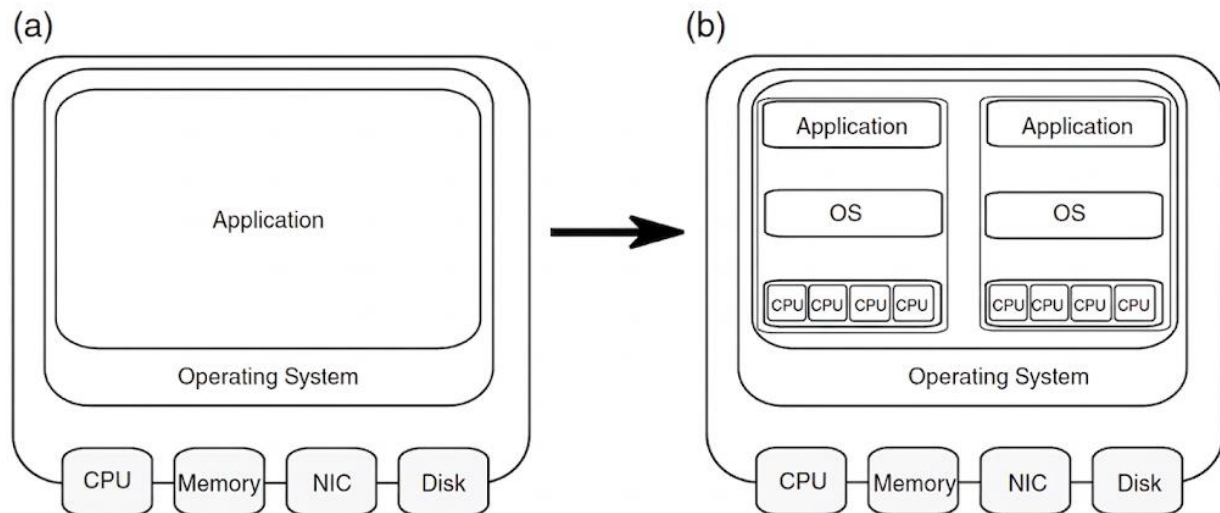
- Virtual CPUs
- Virtual memory
- Virtual network interface cards (NICs)
- Virtual machines (VMs)

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).

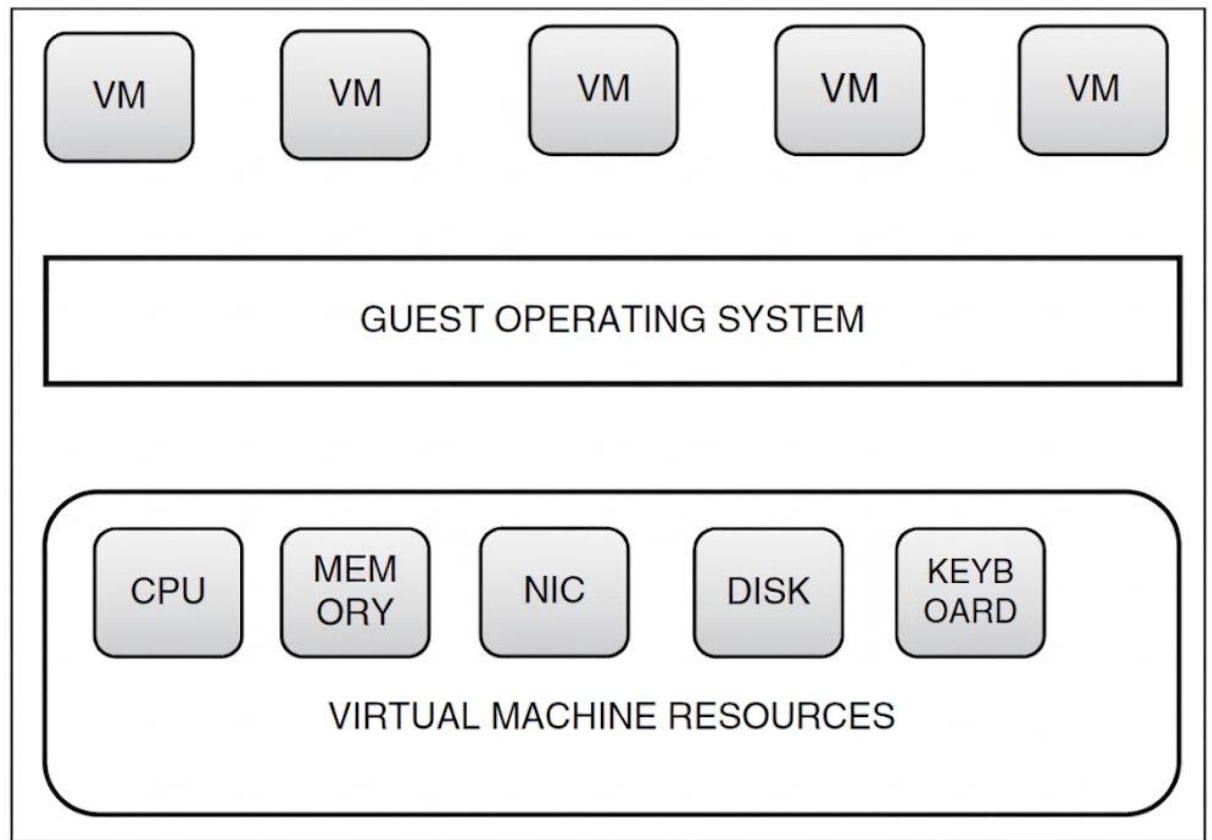
Isolation

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

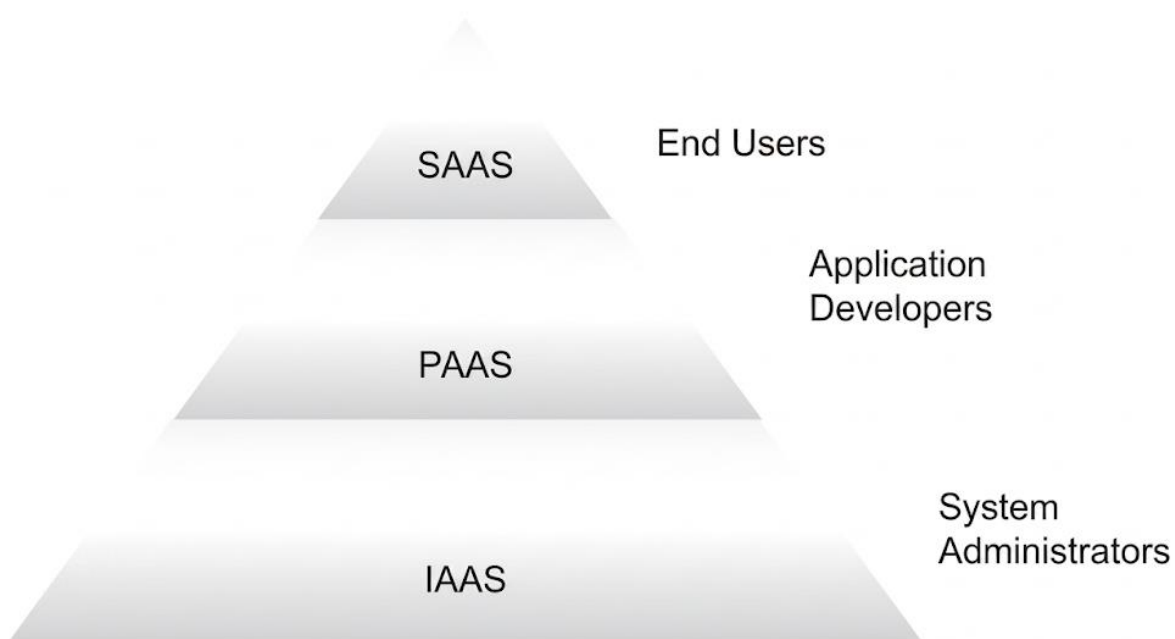
- Oracle Cloud Infrastructure
- Fujitsu
- Hewlett-Packard

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

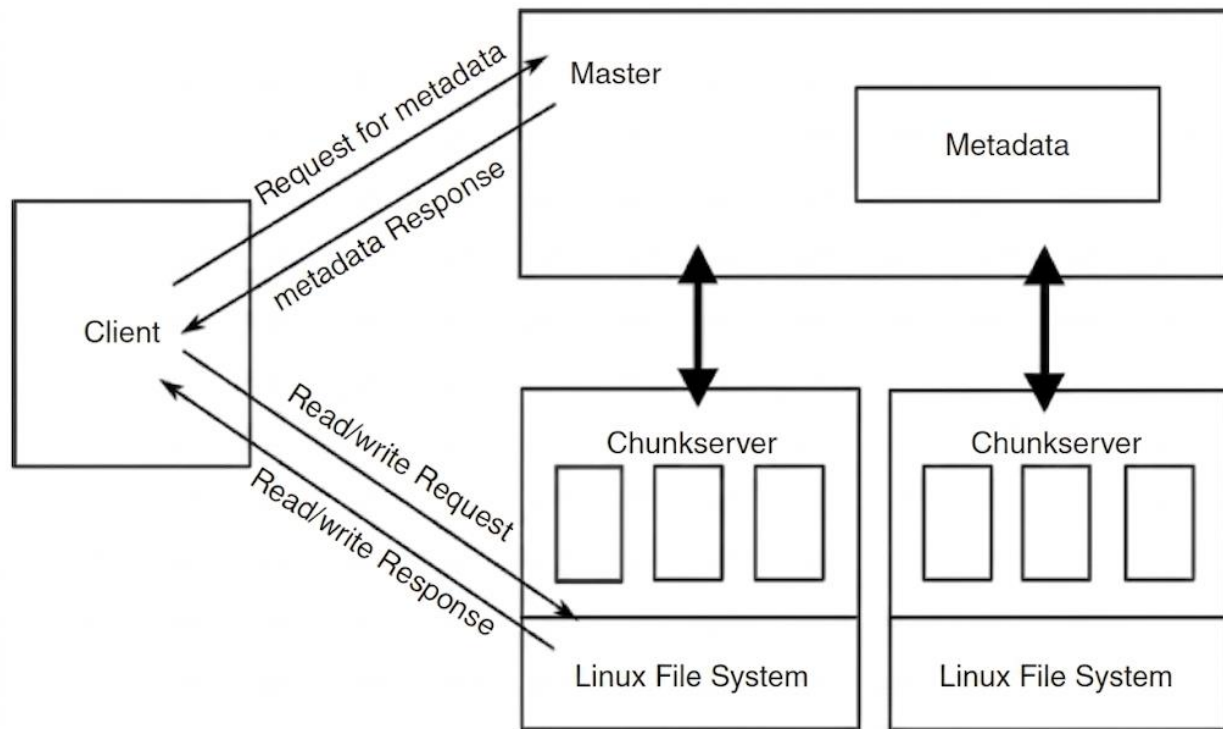
Evolution of Cloud Storage Technologies

- 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

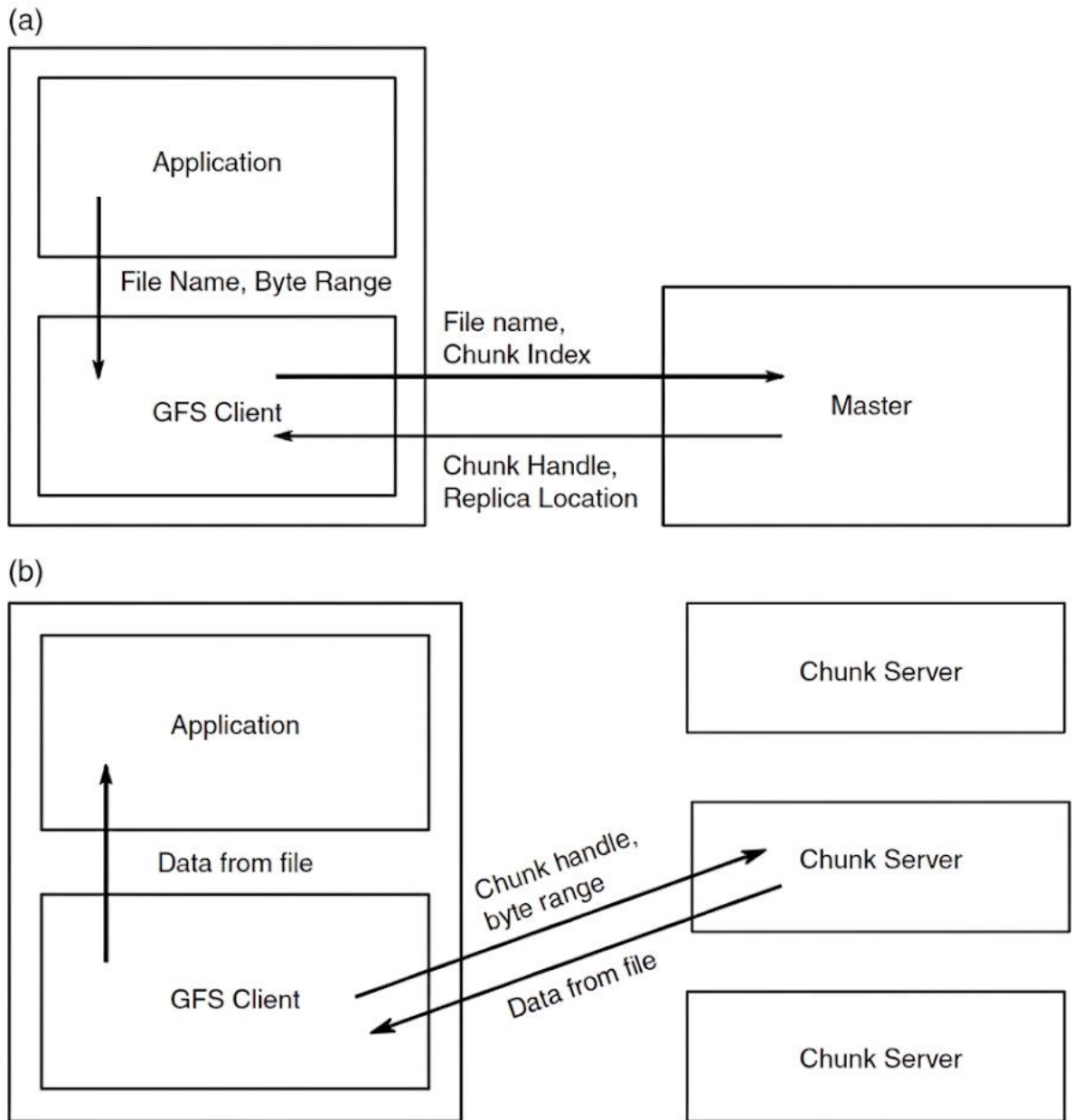
- Chunks are similar to file system blocks but:
 - Much larger in size
- Comparison:
 - Traditional blocks → KBs
 - GFS chunks → **64 MB**
- Advantages of large chunk size:
 - Reduces metadata size
 - Reduces number of master accesses
 - Improves scalability
- Example benefit:
 - Larger chunks → fewer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

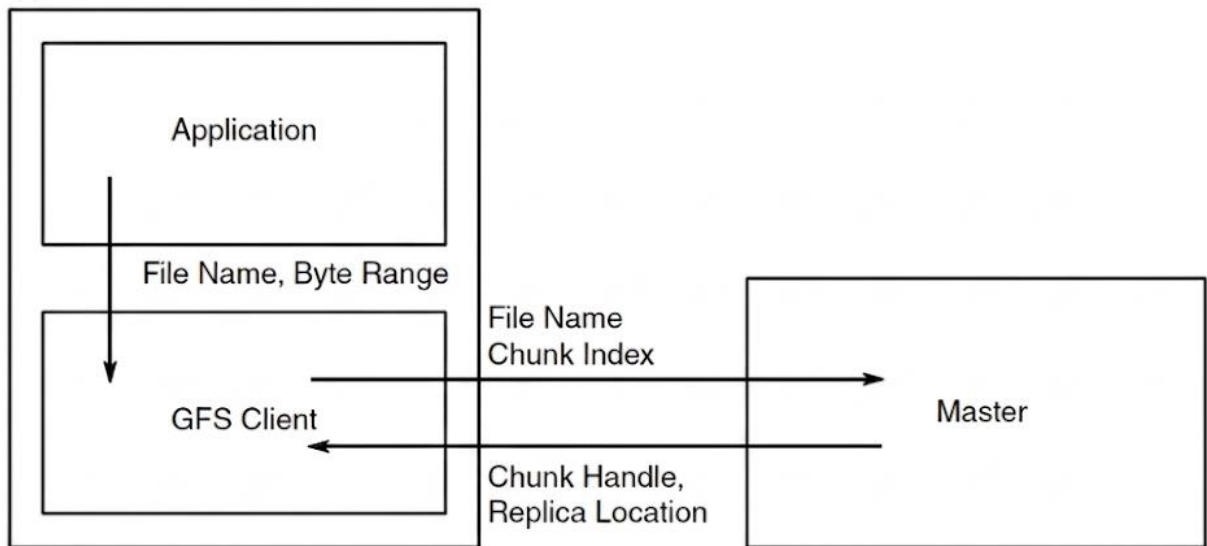
Write Algorithm in GFS

Write Operation Steps

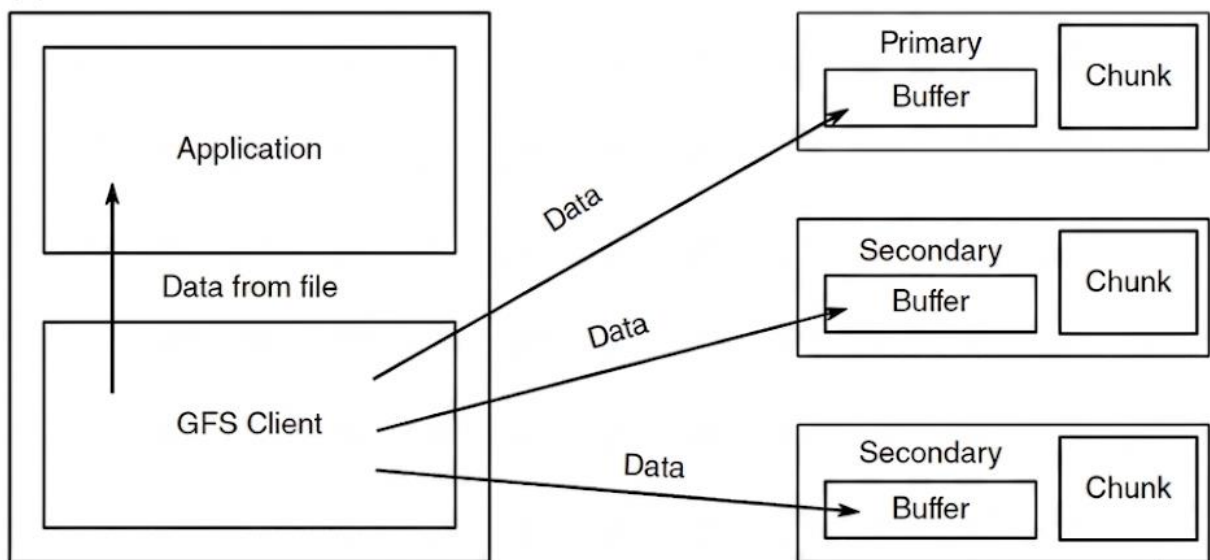
1. Application initiates a write request.
2. GFS client sends filename and data metadata to master.

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.

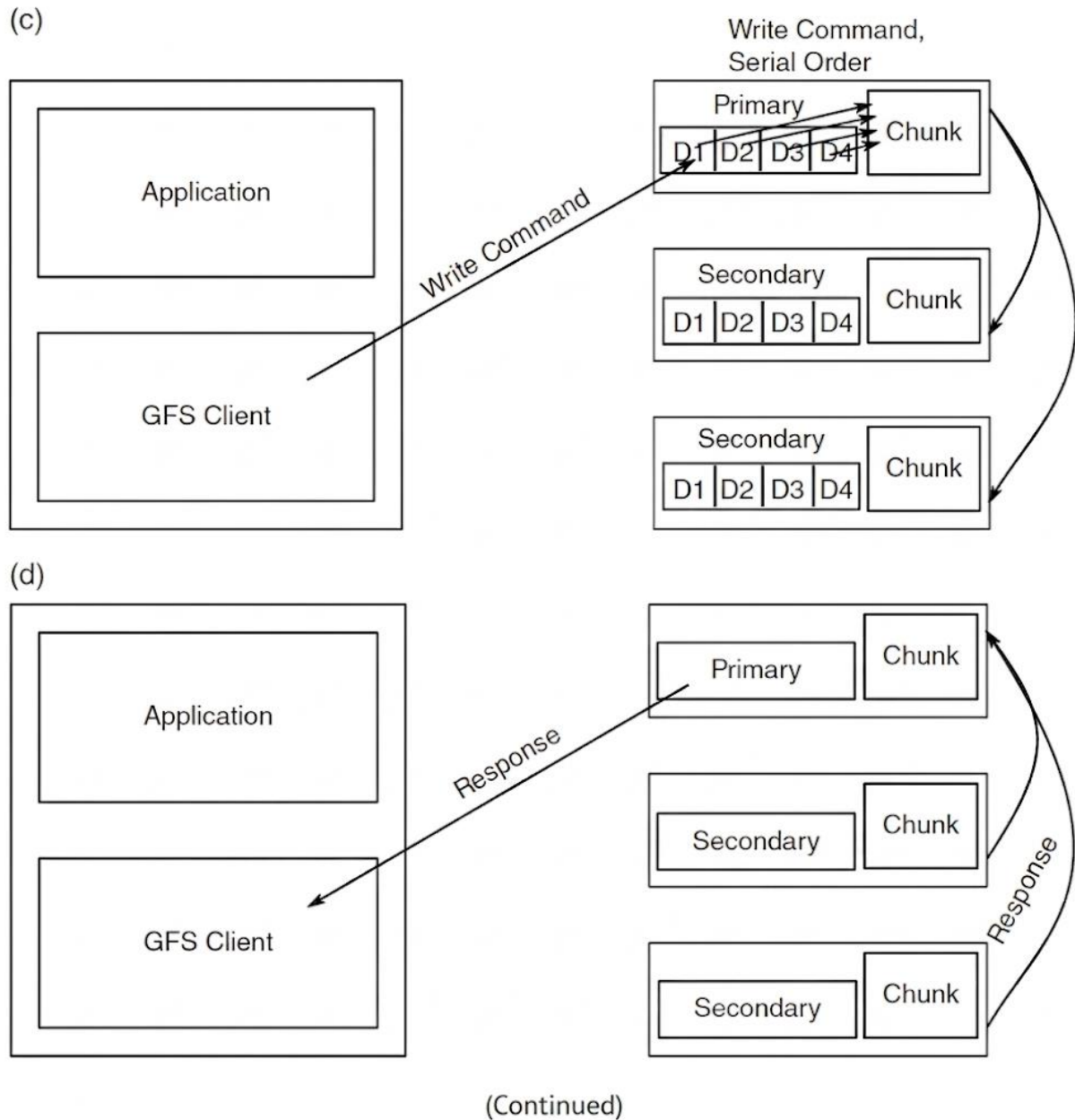
(a)



(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

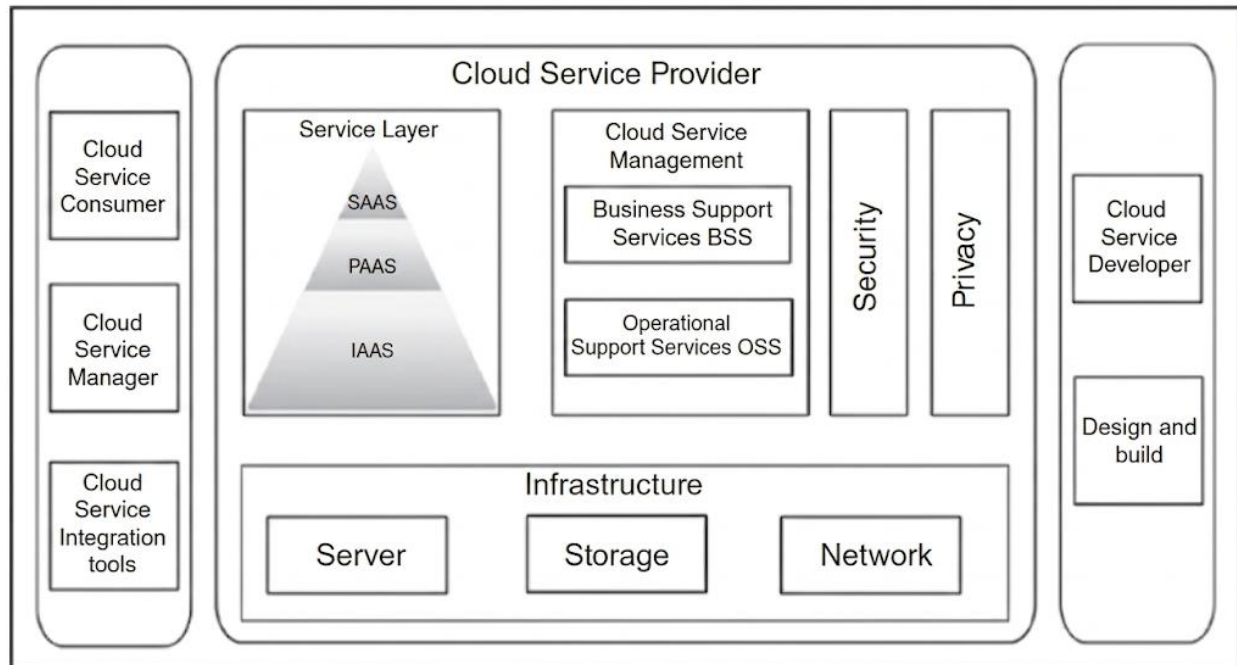
- Fault detection through heartbeat messages
- High scalability through:
 - Large chunk sizes
 - Distributed storage
- Reduced bottleneck at master by:
 - Direct client–chunkserver communication

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