

---

---

# Mastering System Design

Section 7: Database and Storage

Storage

---

---

# Scalability - Section Agenda

1. Introduction to Storage in System Design
2. Understanding Database Models: SQL vs. NoSQL
3. Advanced Database Topics: Sharding, Replication & Polyglot Persistence
4. Object Storage in Modern Systems
5. File Systems and Distributed Storage
6. Big Data Fundamentals
7. Choosing the Right Storage Solution
8. Summary & Recap

---

---

# Introduction to Storage in System Design

Storage and Databases

---

---

# Why Storage Matters in System Design

- All systems generate and consume data — storing it is essential
- Storage impacts performance, reliability, cost
- Persistent storage enables everything from user profiles to search history to analytics



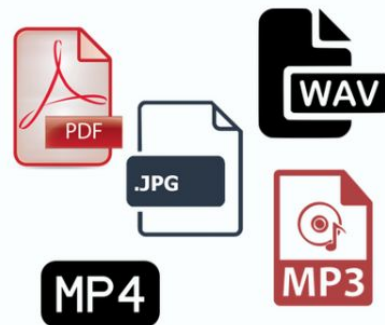
# Structured vs. Unstructured Data

- **Structured:** Rows/columns, predefined schema (e.g., SQL tables)
- **Unstructured:** No schema, flexible format (e.g., images, videos, logs)

STRUCTURED DATA FORMATS



UNSTRUCTURED DATA FORMATS



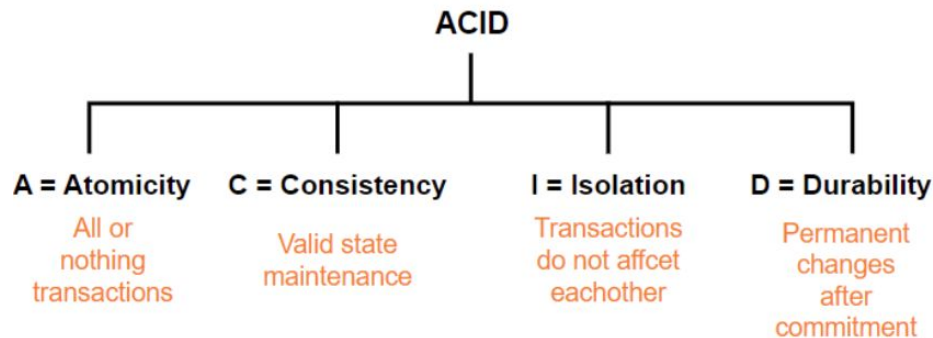
# Categories of Storage

- Database storage (SQL/NoSQL)
- Object storage (e.g., S3)
- File storage (e.g., NFS)
- Block storage (e.g., EBS)



# Storage Properties

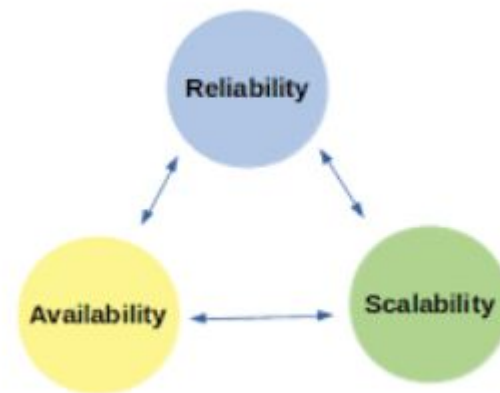
- Durability – Data persists even after failures
- Availability – Data can be accessed when needed
- Consistency – Every read returns the most recent write
- (Optional) Atomicity – Operations are all-or-nothing (relevant in transactional storage)



# The Trade-offs in Storage Design

Scalability vs. Reliability vs.  
Performance

- No perfect solution — must trade off between:
  - Scale (can handle growth)
  - Reliability (resistant to failure)
  - Performance (speed of reads/writes)

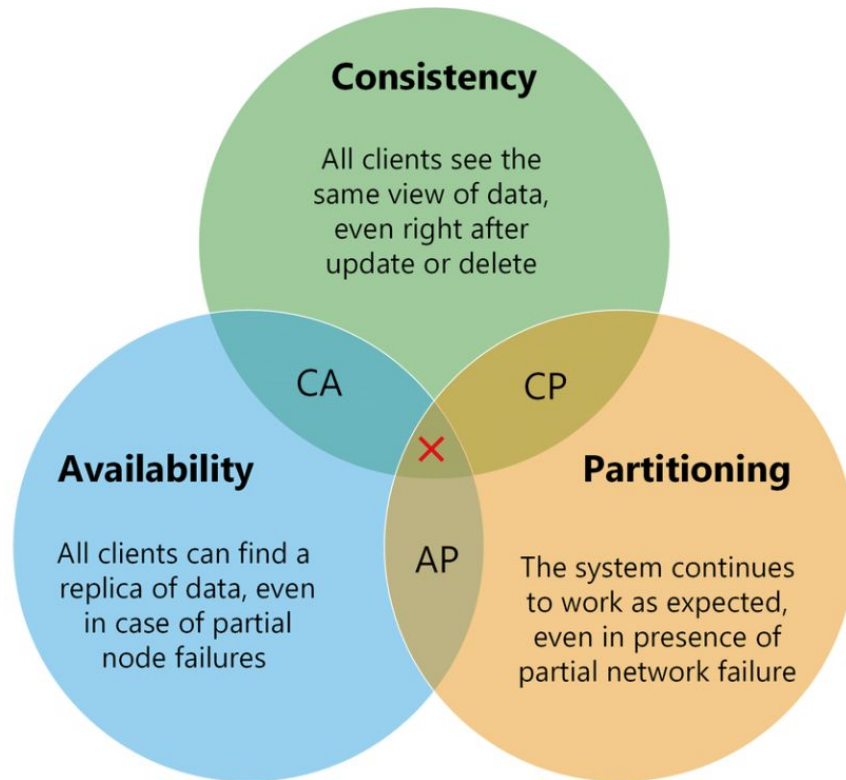




# The CAP Theorem

- In any distributed system, you can only fully guarantee 2 of the 3:
  - Consistency – Every read gets the latest write
  - Availability – Every request receives a response
  - Partition Tolerance – System continues despite network failures
  - No system can have all 3 at the same time

**Note:** Since network partitions are unavoidable, especially at scale, real-world systems must choose between Consistency and Availability during a partition.



# Types of Systems Based on CAP Trade-offs

- CP (Consistency + Partition Tolerance)
  - Prioritizes data correctness over availability
  - During a partition, the system may reject requests to avoid inconsistent reads
  - Not always available, but when it is — data is guaranteed to be correct
  - Example: HBase: Strongly consistent. If a node can't confirm a write across replicas, it won't serve it — even if it means being unavailable briefly.
  - When to use: Financial systems, banking apps, anything where data integrity is critical
- AP (Availability + Partition Tolerance)
  - Prioritizes system uptime over consistency
  - During a partition, the system will serve requests, even if they return stale or eventually consistent data
  - Example: DynamoDB: Inspired by Amazon's Dynamo model, which uses eventual consistency by default for high availability
  - When to use: Social media feeds, product catalogs, content delivery — where being up is more important than perfect accuracy
- CA (Consistency + Availability) — The “Unicorn”
  - Only possible if no network partitions ever occur — i.e., in single-node or tightly coupled systems
  - In practice, not achievable in distributed systems that need to tolerate network faults
  - Example: Relational databases (like PostgreSQL) in standalone mode (not distributed) could be considered CA

# Real-World Use Cases

- E-commerce: Product catalog (structured), images (object)
- Streaming services: Media files (object), user data (NoSQL)
- Log aggregation: Time-series/columnar DB or object storage

# Interview Questions

- Why is storage a critical component in system design?
- How would you differentiate between structured and unstructured data?
- What are the different types of storage systems and their use cases?
- What do durability, availability, and consistency mean in the context of storage?
- What is atomicity, and where is it relevant?
- Can a system be both highly available and strongly consistent? Why or why not?
- What is the CAP theorem? Why is it important in distributed system design?
- Explain the difference between CP and AP systems with examples.
- Why is CA considered rare or impractical in distributed systems?
- How would you decide between consistency and availability when designing a real-world system?
- You're building a photo-sharing app. How would you design storage for photos vs. user metadata?
- What kind of storage system would you choose for an analytics pipeline handling logs and metrics?
- How does object storage differ from file and block storage in terms of access patterns and scalability?

# Summary and Key Takeaways

- Storage is foundational in every system
- Know your data type: structured vs. unstructured
- Choose storage based on properties, access patterns, and scale
- Real-world systems often use a combination of storage types
- What's next:
  - Understanding Database Models: SQL vs. NoSQL

---

---

# Understanding Database Models: SQL vs. NoSQL

Storage and Databases

---

---

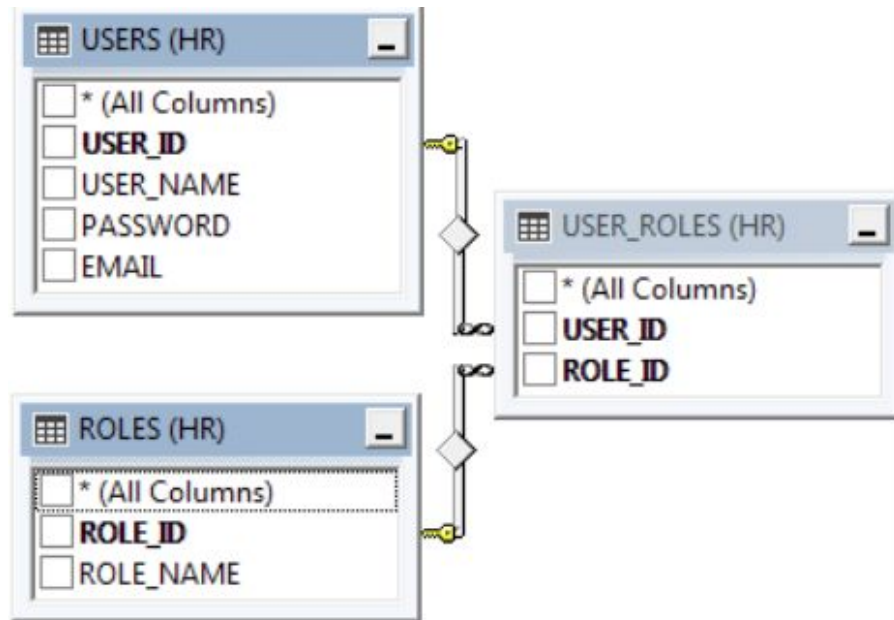
# What is a Database?

- A structured way to store, retrieve, and manage data
- Supports persistent storage and efficient querying
- Core component of backend systems, from small apps to global platforms



# Introduction to Relational Databases (SQL)

- Examples: MySQL, PostgreSQL, Oracle
- Data stored in rows and columns
- Enforces strict schema (structure)
- Uses SQL (Structured Query Language) for queries





# Core Concepts of Relational Databases

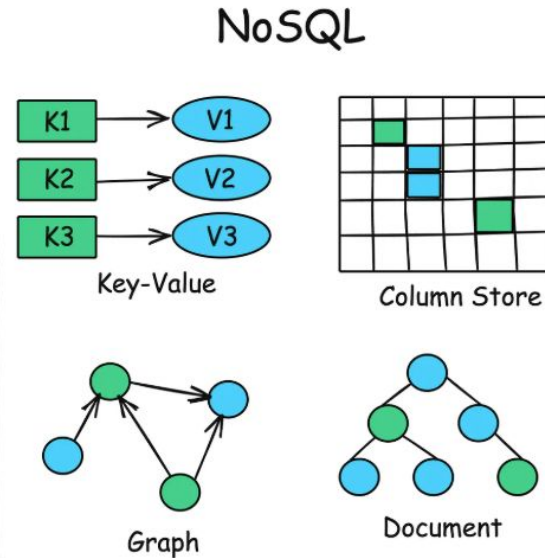
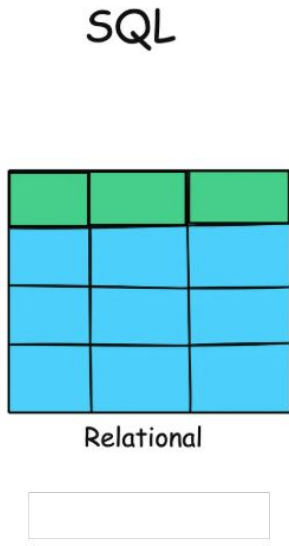
- Schemas: Predefined structure (tables, columns, data types)
- Joins: Combine data across multiple tables
- ACID Properties:
  - Atomicity – all or nothing
  - Consistency – data integrity maintained
  - Isolation – transactions don't interfere
  - Durability – changes survive system failures

# Limitations of Relational Databases

- Not ideal for:
  - Rapidly changing or schema-less data
  - Large-scale horizontal scaling
  - Flexible or nested data (e.g., JSON blobs)

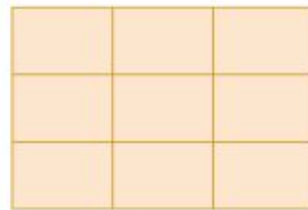
# Introduction to NoSQL

- Designed for flexibility and scale
- Schema-less or dynamic schema
- Types:
  - Document (MongoDB)
  - Key-Value (Redis, DynamoDB)
  - Columnar (Cassandra, HBase)
  - Graph (Neo4j)

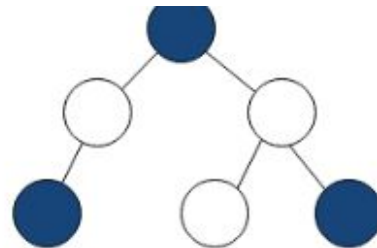


# NoSQL - Deep Dive

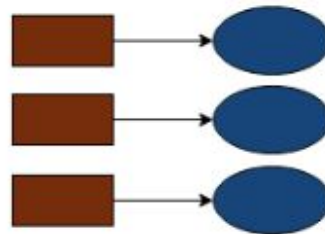
- Document Databases
  - JSON-like structure (key-value pairs, nesting supported)
  - Ideal for content management, user profiles
  - Example: MongoDB
- Key-Value Databases
  - Simple, fast, key-based lookups
  - High performance, low latency
  - Example: Redis, DynamoDB
- Columnar Databases
  - Store data by column, not row
  - Optimized for analytical queries over large datasets
  - Example: Cassandra, HBase
- Graph Databases
  - Store entities and relationships as nodes and edges
  - Efficient for highly connected data (e.g., social networks)
  - Example: Neo4j



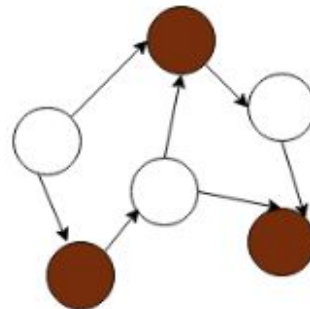
Column



Graph



Key Value



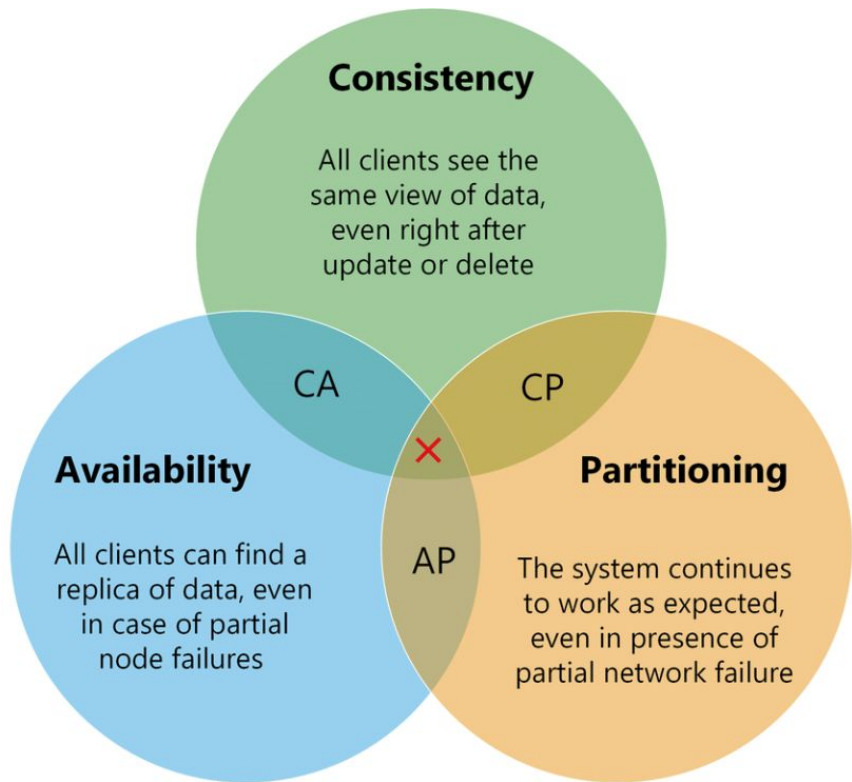
Document

# BASE Properties in NoSQL

- **Basically Available:** Always returns a response (even if stale)
- **Soft state:** System state may change over time
- **Eventually consistent:** Data will be consistent... eventually

# CAP Theorem – Revisited

- Consistency, Availability, Partition Tolerance
- No distributed system can guarantee all 3 simultaneously
- SQL tends toward CP, NoSQL systems often choose AP or BASE



# When to use what

- When to Use SQL
  - Complex queries and relationships
  - Strong consistency needed
  - Structured and well-known schema
  - Example: Banking, ERP, Inventory Systems
- When to Use NoSQL
  - High scalability needed
  - Flexible or evolving data structure
  - Low-latency or high-volume operations
  - Example: IoT, Recommendation Systems, Caching, Logs

# Interview Questions

- What are the key differences between SQL and NoSQL databases?
- Explain ACID vs. BASE.
- What are the different types of NoSQL databases, and when would you use each?
- When would you prefer MongoDB over PostgreSQL?
- What are the trade-offs in the CAP theorem?
- Where do SQL and NoSQL databases fit within the CAP theorem categories?
- What database model would you choose for:
  - a. A financial ledger system
  - b. A product catalog
  - c. A real-time chat app
- What are the limitations of relational databases in modern distributed systems?
- What is polyglot persistence and why is it useful?
- How does data modeling differ between SQL and NoSQL systems?



# Summary and Key Takeaways

- SQL and NoSQL are both essential in modern system design
- Choose based on consistency needs, data structure, and scale
- Many systems today use both (polyglot persistence)
- Next:
  - We'll dive deeper into advanced database topics — sharding, replication, and polyglot persistence

---

---

# Advanced Database Topics

Storage and Databases

---

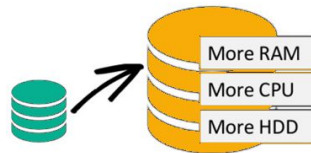
---

# Scaling Strategies – SQL vs. NoSQL

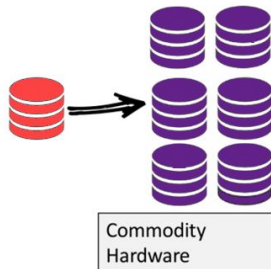
## Vertical Scaling (Scale-Up) – Traditional SQL Databases

- How it works: Add more CPU, RAM, or SSD to a single database server.
- Common with: Relational DBs like MySQL, PostgreSQL, Oracle.
- Pros:
  - Simpler architecture
  - Strong consistency (ACID guarantees)
- Cons:
  - Limited by hardware capacity
  - Cost grows non-linearly
  - Risk of single point of failure

**Scale-Up** (*vertical scaling*):



**Scale-Out** (*horizontal scaling*):

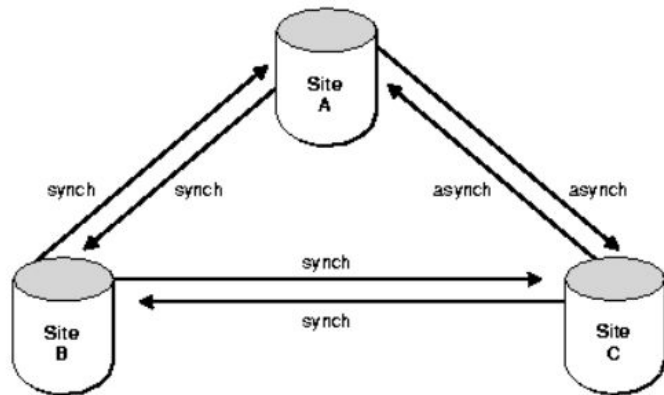


## Horizontal Scaling (Scale-Out) – Modern NoSQL Databases

- How it works: Add more database nodes to distribute data and load.
- Common with: NoSQL DBs like MongoDB, Cassandra, DynamoDB
- Pros:
  - Elastic scalability
  - Handles large-scale traffic and big data
  - Better fault tolerance
- Cons:
  - Complex architecture
  - Weaker consistency (often eventual)

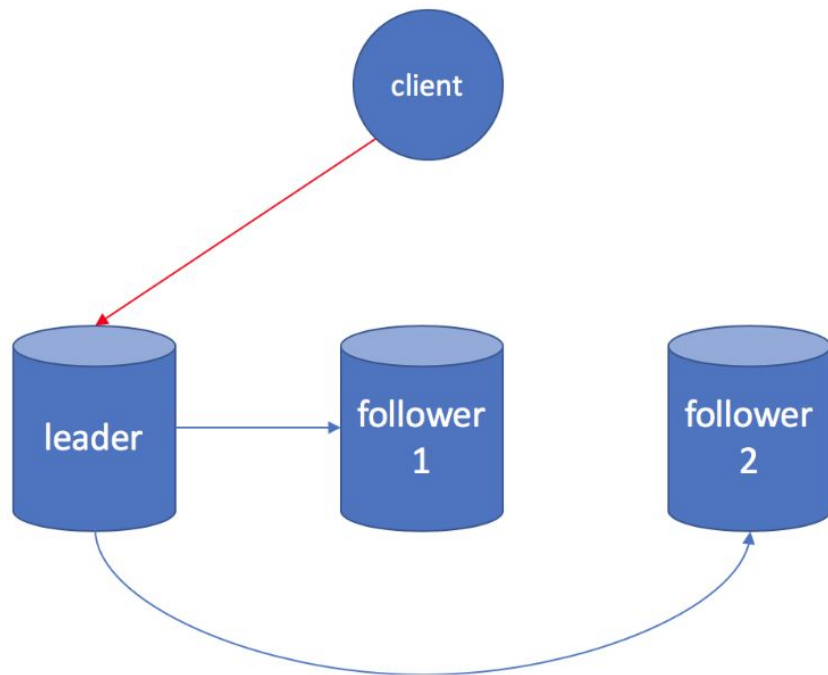
# What is Replication?

- Definition: Copying data from one database node to another for redundancy and performance.
- Benefits:
  - Fault tolerance
  - Read performance improvement
  - Data availability
- Trade-offs in real-world systems(CAP Theorem):
  - Replication may favor availability
  - Strong consistency may sacrifice availability



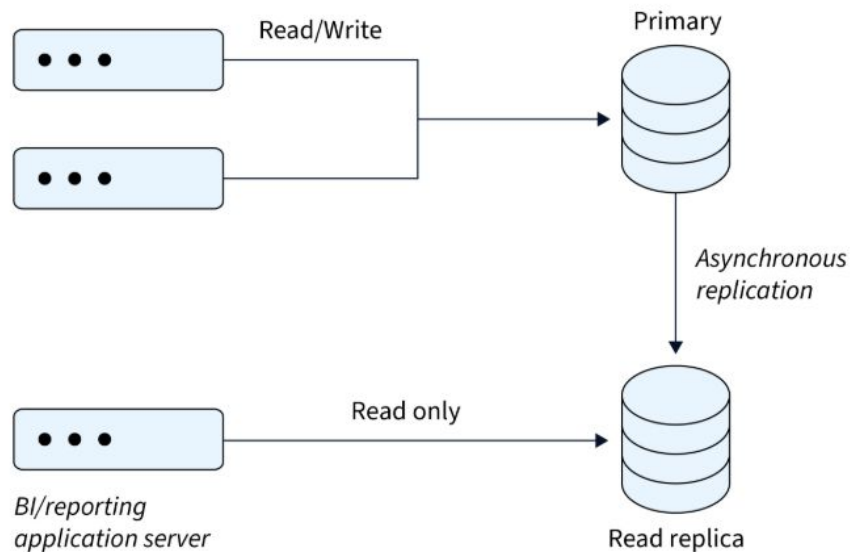
# Leader-Follower Replication

- How it works:
  - Writes go to the Leader
  - Reads from Followers
- Consistency Consideration:
  - Asynchronous replication = possible lag



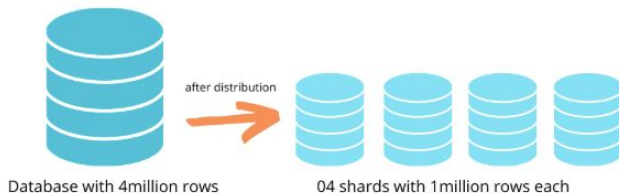
# Read Replicas

- Use case:
  - Scaling read-heavy workloads
- Differences from Leader-Follower:
  - Often used just for load balancing reads
  - Doesn't take part in writes



# What is Sharding?

- Definition: Splitting data across multiple databases (shards) to scale horizontally.
- Why it's needed:
  - Performance and storage limits of a single node
- Types of sharding:
  - Horizontal Sharding: Rows are distributed across shards (most common).
  - Vertical Sharding: Tables or columns are split across shards based on function or access pattern.



# Sharding Strategies

- Range-Based Sharding
  - Split by value ranges (e.g., user\_id 1-1000).
  - Can create hot spots.
- Hash-Based Sharding
  - Distribute using a hash function.
  - Even distribution but harder to query ranges.
- Consistent Hashing for Resilient Hash-Based Sharding
  - Traditional hashing breaks when nodes are added/removed (re-hashing everything)
  - Consistent Hashing solves this: Only a subset of keys need to be re-mapped when topology changes
  - Enables better elasticity and fault tolerance
- Geo-Based Sharding
  - Shard by user region/location.
  - Useful in geo-distributed systems.



# Polyglot Persistence

- Definition: Using different types of databases for different components/services
- Why: Each DB excels at different things (search, analytics, relations)
- Benefits:
  - Better performance
  - Optimized storage and queries

# Interview Questions – Advanced Database Topics

- What is the difference between horizontal and vertical scaling? When would you prefer one over the other?
- Explain leader-follower replication. How does it impact consistency and availability?
- What are the pros and cons of using read replicas?
- Compare range-based and hash-based sharding. What are the trade-offs of each?
- Why is consistent hashing important in distributed databases?
- How does the CAP theorem influence the design of distributed databases? Can you give an example of a CP or AP system?
- What is polyglot persistence? Why might an architecture choose to use multiple types of databases?
- How do systems like Netflix or Uber use a mix of database technologies in production?

# Summary & Key Takeaways

- Database Scaling strategies
- Replication and Replication strategies
- Sharding and Sharding Strategies
- Polyglot persistence boosts performance in complex systems
- What's next:
  - Object Storage in Modern Systems

---

---

# Object Storage in Modern Systems

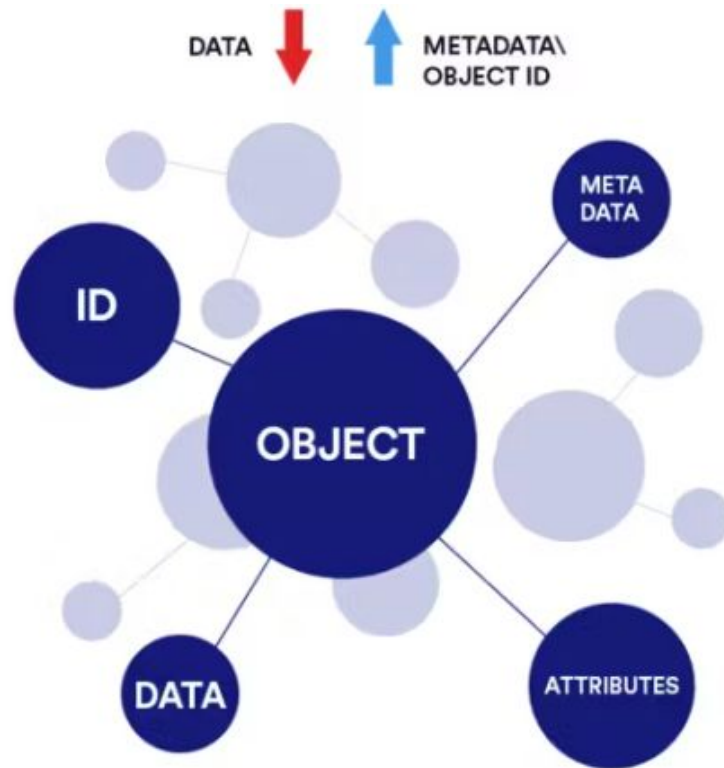
Storage and Databases

---

---

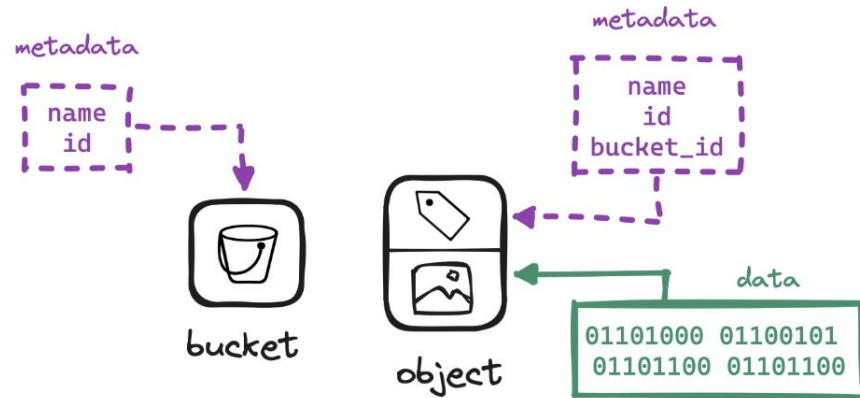
# What is Object Storage?

- A storage architecture that manages data as objects, not files or blocks
- Each object contains:
  - The data itself
  - A unique identifier (key)
  - Metadata
- Unlike traditional storage, it is scalable, distributed, and suited for unstructured data



# Key Concepts in Object Storage






- Object: Self-contained unit of data
- Bucket: Logical container for storing objects (like a folder)
- Metadata: Custom data that describes the object (e.g., MIME type, timestamps, custom tags)

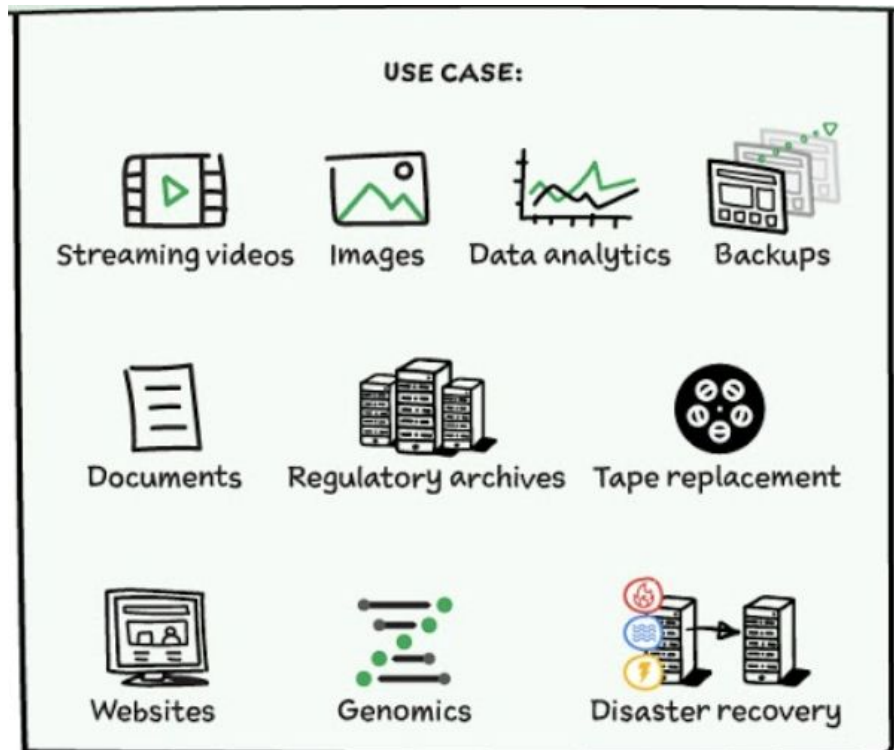


# Popular Object Storage Platforms

- Amazon S3: Industry standard, rich ecosystem, high durability
- Google Cloud Storage (GCS): Simplified tiers, ML-friendly
- Azure Blob Storage: Strong integration with Microsoft stack
- On-prem alternatives (MinIO, Ceph) for hybrid/cloud-native use

# Common Use Cases

-  Media storage (videos, images, large files)
-  Backups and archives
-  Data lakes for analytics
-  Static website hosting (e.g., via S3)
-  IoT and ML data pipelines





# Important Considerations with Object Storage

- Performance Considerations
  - Latency: Object storage has higher latency than block/file
  - Throughput: Designed for massive parallel access
  - Consistency: Eventual consistency in some platforms (e.g., S3)
  - Access patterns: Suited for write-once, read-many workloads
- Cost Considerations
  - Storage class tiers: Standard, Infrequent Access, Archive (e.g., S3 Glacier)
  - Charges: Storage, requests (PUT, GET), egress bandwidth
  - Best practices:
    - Use lifecycle rules for archiving/deletion
    - Monitor usage and optimize classes

# Interview Questions - Object Storage

- Conceptual Questions
  - What is object storage and how is it different from file or block storage?
  - When would you choose object storage over a traditional file system?
  - Explain the structure of an object in object storage. What role does metadata play?
- System Design Scenarios
  - Design a media hosting platform (e.g., YouTube). How would you use object storage for video uploads and streaming?
  - How would you architect a cost-efficient backup system for petabytes of logs using Amazon S3?
  - In a microservices system, how would services securely share large files using object storage?
- Practical & Trade-off Questions
  - What are the performance trade-offs of using object storage for real-time access?
  - How do storage class tiers (e.g., S3 Standard vs. Glacier) influence design decisions?
  - How would you implement access control for user-specific data stored in object storage?

# Summary & Key Takeaways

- Object storage is essential for modern, scalable, unstructured data storage
- Understand buckets, objects, and metadata
- Choose the right storage based on access patterns and cost
- Embrace object storage for media, backups, data lakes, and more
- What's next:
  - File Systems and Distributed Storage

---

---

# File Systems and Distributed Storage

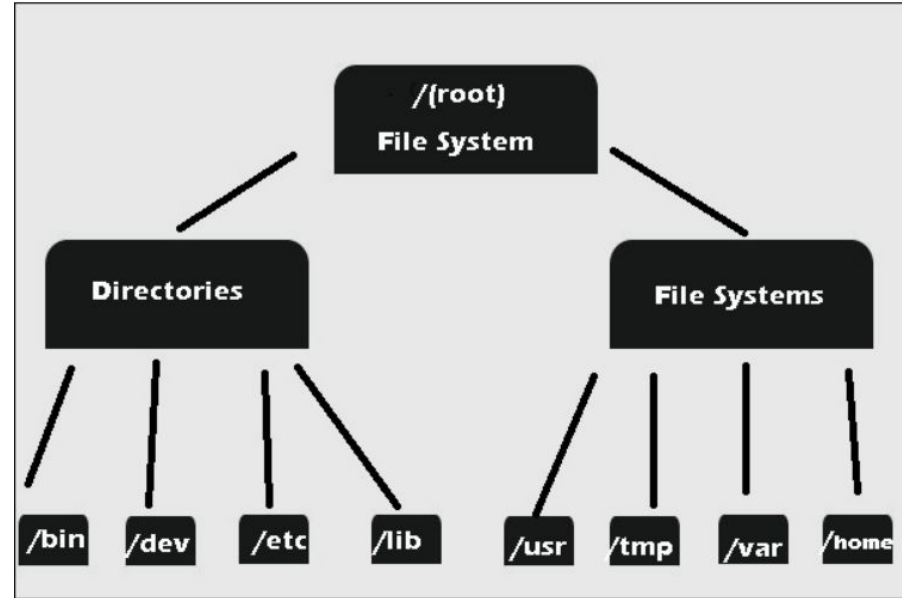
Storage and Databases

---

---

# What is a File System?

- Defines how data is stored and retrieved on disk
- Examples: ext4, NTFS, XFS
- Handles file metadata, directories, permissions



# Key Characteristics of Traditional File Systems

- Hierarchical structure (folders/files)
- Limited scalability across machines
- Suited for single-node systems
- Use cases: Personal devices, small servers

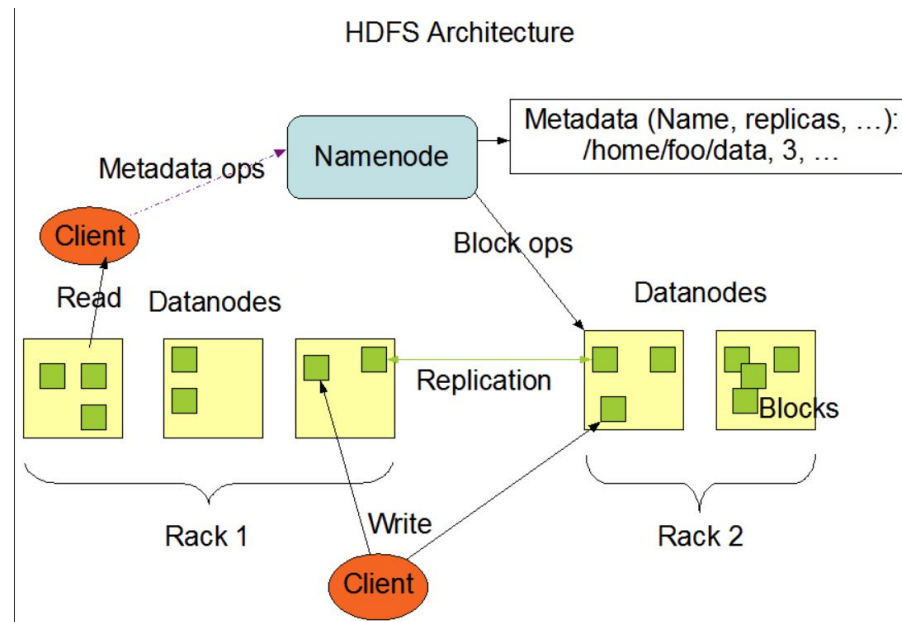
# What is a Distributed File System (DFS)?

- Allows file access across multiple nodes
- Ensures redundancy, fault tolerance
- Appears as a single file system to the user
- Key benefit: Scalability & reliability
- Example:
  - HDFS (Hadoop Distributed File System) – Big data analytics
- Use Cases
  - Big Data Analytics (e.g., Hadoop, Spark)
  - Enterprise storage (shared file systems)
  - Scientific computing
  - Media and backup systems



# DFS Architecture and How Replication Works

- NameNode: Manages metadata, file hierarchy
- DataNodes: Store actual data blocks
- Replication across nodes for fault tolerance
- Data blocks are replicated across nodes
  - Ensures availability during node failures
  - Replication factor configuration
  - Example: HDFS default = 3 copies
- Block size & striping concepts





# Scalability & Fault Tolerance

- DFS can scale horizontally by adding nodes
- Automatic rebalancing of data
- Supports high-throughput workloads
- Built-in failure recovery mechanisms

# Interview Questions

- What is the difference between a file system and a distributed file system?
- How does HDFS ensure fault tolerance and reliability?
- What roles do NameNode and DataNodes play in distributed file systems?
- Explain the trade-offs between latency and throughput in distributed storage systems.
- In what scenarios would you use CephFS or GlusterFS over HDFS?
- How would you handle scaling storage in a high-throughput analytics system?

# Summary & Key Takeaways

- File systems organize how data is stored and accessed
- DFS brings scalability and resilience to large-scale systems
- Choose based on trade-offs: performance, complexity, fault tolerance
- Crucial for data-heavy workloads and analytics platforms
- What's next:
  - Big Data Fundamentals

---

---

# Big Data Fundamentals - 30000 Feet Overview

Storage and Databases

---

---

# What is Big Data?

- Big Data refers to datasets too large or complex for traditional data-processing tools.
- Emerged due to growth in data generation from web, sensors, apps, and machines.
- Requires new storage, processing, and analysis strategies.

## What is Big Data?



**Large amounts of data**









**collected passively from digital interactions**



**with great variety and a high rate of velocity.**

# The 6 V's of Big Data

- **Volume**
  - Refers to the massive amount of data generated and stored.
  - Example: Terabytes to exabytes of data from social media, IoT devices, transaction logs, etc.
- **Velocity**
  - The speed at which data is generated, collected, and processed.
  - Example: Real-time data from stock markets, clickstreams, or sensor feeds.
- **Variety**
  - The diversity of data formats and types.
  - Includes structured (databases), semi-structured (JSON, XML), and unstructured (videos, emails, images, etc.).
- **Veracity**
  - The accuracy, quality, and trustworthiness of data.
  - Challenges include noise, inconsistencies, missing values, and ambiguity.
- **Value**
  - The usefulness of the data — i.e., the insights, decisions, or business impact derived from it.
  - Without value, data is just storage overhead.
- **Variability**
  - The inconsistency or unpredictability in data meaning, format, or flow.
  - Example: Changing user behavior, evolving language (e.g., slang), or fluctuating sensor readings.

VOLUME	VARIETY	VELOCITY
The amount of data from myriad sources.	The types of data: structured, semi-structured, unstructured.	The speed at which big data is generated.
		
VERACITY	VALUE	VARIABILITY
The degree to which big data can be trusted.	The business value of the data collected.	The ways in which the big data can be used and formatted.
		

# Why Traditional Storage Fails at Scale

- Limited scalability – not built for petabyte-scale data.
- Performance bottlenecks – not optimized for parallel reads/writes.
- Cost inefficiencies – expensive to scale up (vertical scaling).
- Lack of fault tolerance – no data replication or distributed recovery.

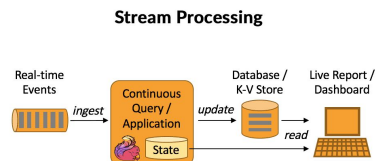
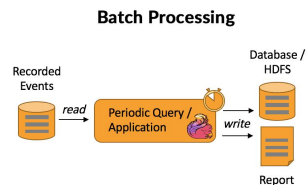
# Common Big Data Workloads

- Logs & Events: Application logs, system logs, server metrics
- Clickstreams: User navigation paths on websites and apps
- IoT Data: Sensor streams, smart devices
- ML Pipelines: Feature extraction, model training, data versioning



# Batch vs. Stream Processing

- Storing Big Data is only the first step — the real value comes from processing it efficiently.
- Batch Processing
  - Processes large data chunks at once
  - High throughput, higher latency
  - Examples: Hadoop, Spark (batch mode)
- Stream Processing
  - Processes data in real-time as it arrives
  - Low latency, continuous processing
  - Examples: Apache Kafka, Spark Streaming, Flink
- When to Use Batch vs. Stream
  - **Batch:** Historical analysis, ETL jobs, nightly aggregations
  - **Stream:** Real-time monitoring, fraud detection, recommendation systems



# Interview Questions

- What are the 5 V's of Big Data? Why are they important?
- Why do traditional databases struggle with Big Data workloads?
- Compare HDFS and S3. In what scenarios would you choose one over the other?
- What types of workloads qualify as Big Data problems?
- What is the difference between batch and stream processing?
- What is Delta Lake and how does it improve upon traditional data lakes?
- How would you design a system to process terabytes of log data daily?
- What storage and processing frameworks would you use and why?

# Summary and Key Takeaways

- Big Data is defined by the 5 V's.
- Traditional storage struggles with scale and velocity.
- Distributed systems like HDFS, S3, and Delta Lake address these challenges.
- Big Data workloads span multiple industries and domains.
- Batch and Stream processing are key paradigms for handling big data.
- What's next:
  - Choosing the Right Storage Solution & Final Recap of storage section

# Section Summary - Storage and Databases

- Introduction to Storage in System Design
  - 👉 Use when: Deciding foundational storage strategy based on data type, access patterns, and reliability needs.
- Understanding Database Models: SQL vs. NoSQL
  - 👉 Use SQL: For structured data, complex queries, and strong consistency.
  - 👉 Use NoSQL: For high scalability, flexible schemas, and distributed environments.
- Advanced Database Topics: Sharding, Replication & Polyglot Persistence
  - 👉 Use when: Scaling read/write loads, improving fault tolerance, or handling multi-model requirements.
- Object Storage in Modern Systems
  - 👉 Use when: Storing unstructured data (images, videos, backups, logs) that grows at scale and requires metadata tagging.
- File Systems and Distributed Storage
  - 👉 Use when: You need large-scale, high-throughput data access (e.g., analytics, big data pipelines) with reliability.
- Big Data Fundamentals
  - 👉 Use when: Working with high-volume, high-velocity datasets and need distributed, scalable storage + real-time/analytical processing.
- What's Next: Performance - Concepts, Tools & Techniques