

SDE: System Design and Engineering

Lecture – 4.1

Introduction to

Databases and Storage

From Zero to Google: Architecting the Invisible Infrastructure

by

Aatiz Ghimire

Sections

- **Database Fundamentals and Internals**
- **Scaling, Sharding, and High Availability**
- **Storage Systems and Large-Scale Data Management**
- Data Structures and Internals
- Messaging and Streaming Architectures
- Data Management and Migration
- Time Series and Special-Purpose Databases
- PostgreSQL and Open Source Trends

What is Database?

A **database** is a structured collection of data stored electronically.

Enables users and applications to **store, retrieve, update, and manage information efficiently**.

Data may include: Text, Numbers, Images, Videos and Other digital content

Managed by a **Database Management System (DBMS)**:

- Specialized software for organizing and controlling data access.
- Ensures **consistency, security, and integrity**.

Key Role in Modern Computing:

- Supports applications like online shopping, banking, and social media.
- Widely used in business, government, healthcare, and research.
- Enables **data-driven decisions** and **streamlined operations**.

What Does ACID Mean?

ACID describes the **four key properties** that ensure reliable processing of database transactions:

Atomicity

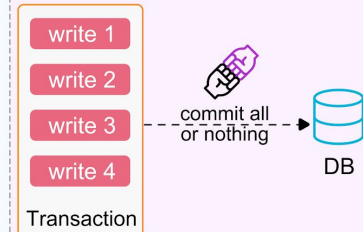
- *All or nothing.*
- A transaction's operations are executed as a **single unit**.
- If any part of the transaction fails, **all changes are rolled back**.

Consistency

- Ensures that a transaction **preserves all database rules and constraints**.
- The database transitions from one **valid state to another valid state**.
- (Note: This is different from *consistency* in the CAP theorem.)*

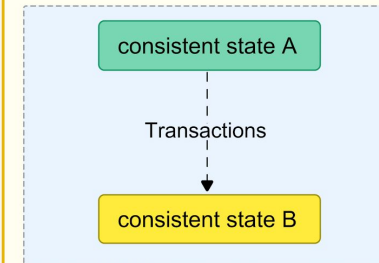
Atomicity

All or nothing



Consistency

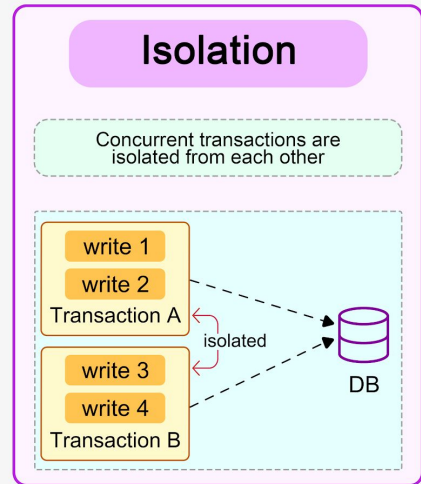
Preserving database invariants



What Does ACID Mean?

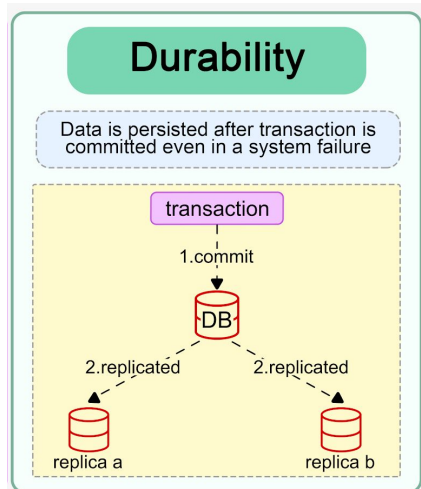
Isolation

- Transactions execute **independently** of each other.
- Concurrent transactions do not interfere.
- The strictest level is **serializability**, which makes transactions behave as if they ran sequentially.
- In practice, databases often use **weaker isolation levels** to improve performance.



Durability

- Once a transaction is committed, its changes are **permanently recorded**.
- Data remains intact even in the event of a **system failure or crash**.
- In distributed systems, this often involves **replicating data across nodes**.

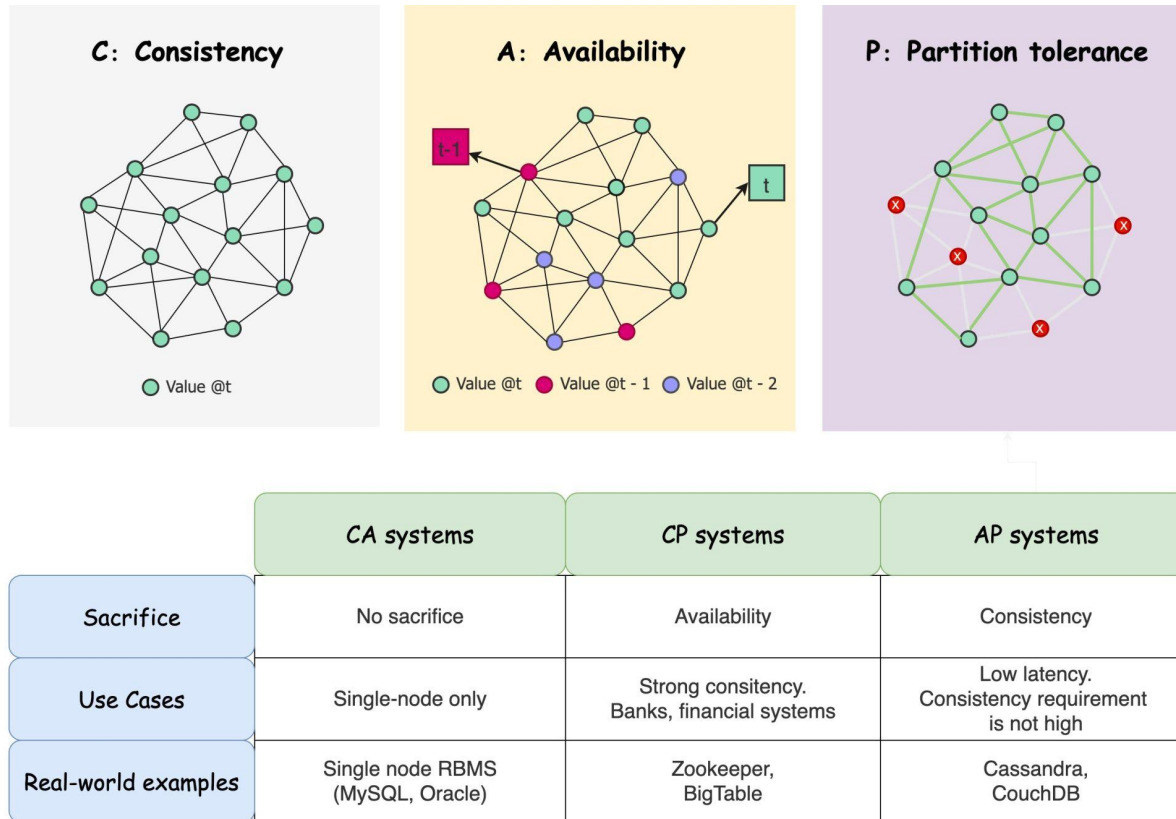


What Is the CAP Theorem?

States that **a distributed system cannot simultaneously guarantee all three of these properties:**

- **Consistency**
 - All clients see the same data at the same time, regardless of which node they connect to.
- **Availability**
 - Every request receives a response (success or failure), even if some nodes are down.
- **Partition Tolerance**
 - The system continues to function despite network partitions.

CAP Theorem



The “2 of 3” Rule in CAP Theorem

- Often simplified as “choose any two,” but this can be **misleading**.
- The theorem mainly **prohibits perfect consistency and perfect availability when a partition occurs**, which is uncommon.

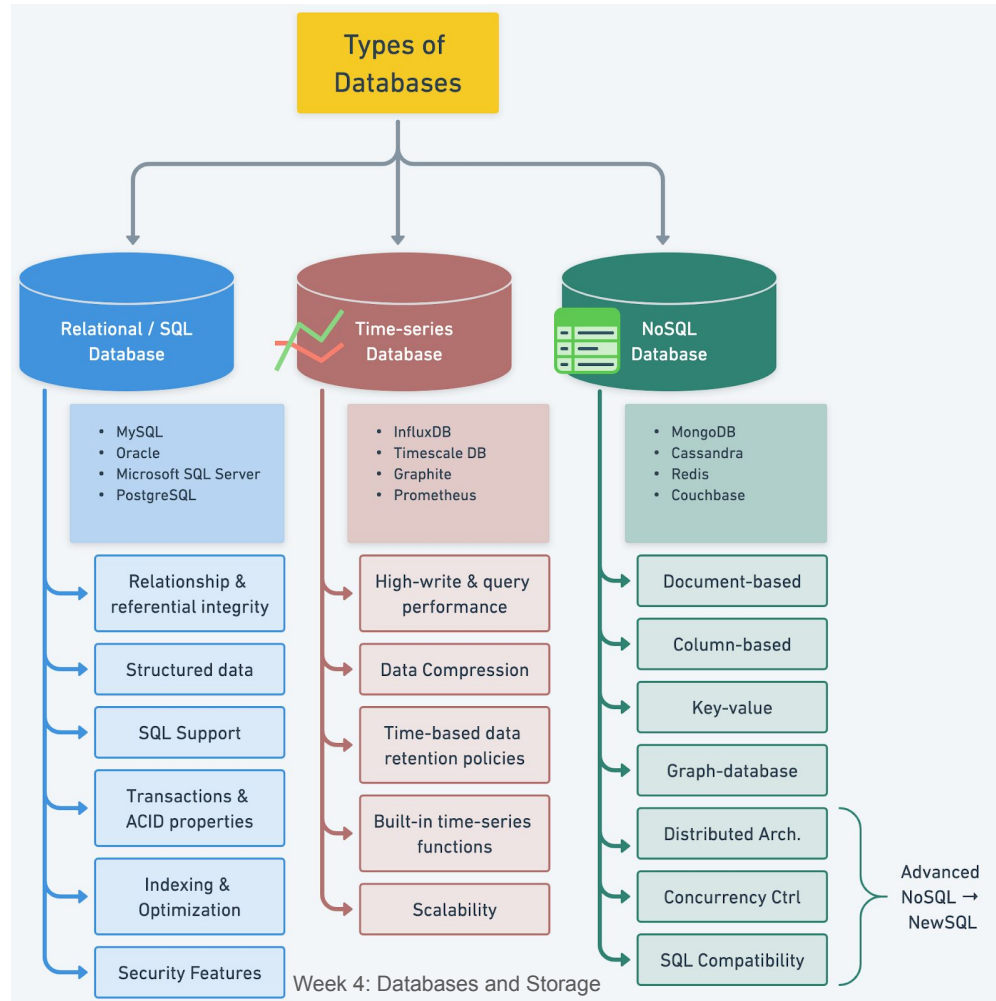
Why Is This Often Misunderstood?

- Real systems rarely experience continuous partitions.
- Systems often **trade off latency and consistency** even when there is no partition (explored by the PACELC theorem).
- Simply labeling a database as CP or AP **does not explain why it is chosen** for a specific application.

Practical Considerations

- **CAP is only part of the picture.**
- Choosing a database requires evaluating:
 - Performance under normal conditions
 - Operational complexity
 - Latency vs. consistency trade-offs
 - Application requirements (e.g., throughput, fault tolerance)
- For example, Cassandra is popular for chat systems **not only** because of CAP trade-offs, but also for:
 - High write availability
 - Tunable consistency levels
 - Operational maturity

Database Types



Data Models


1. Flat Model

- **Description:**
 - Simplest model: a single table of rows and columns (like a spreadsheet).
- **Strengths:**
 - Easy to understand and implement.
- **Limitations:**

Cannot efficiently represent complex relationships between data entities.

Flat Model

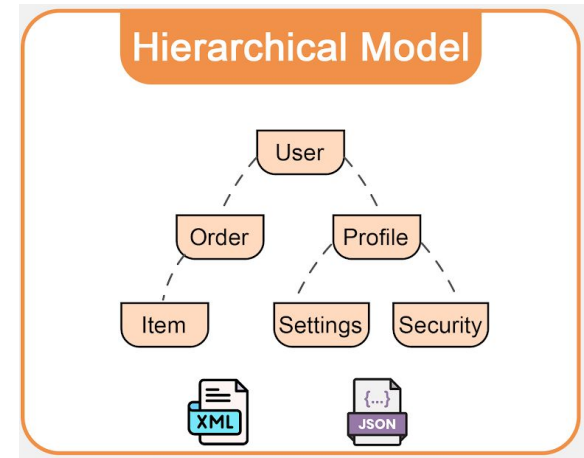
	User No.	User Name
User 1	123	Bob
User 2	345	Alice
User 3	456	Carrie

 Excel

Data Models

2. Hierarchical Model

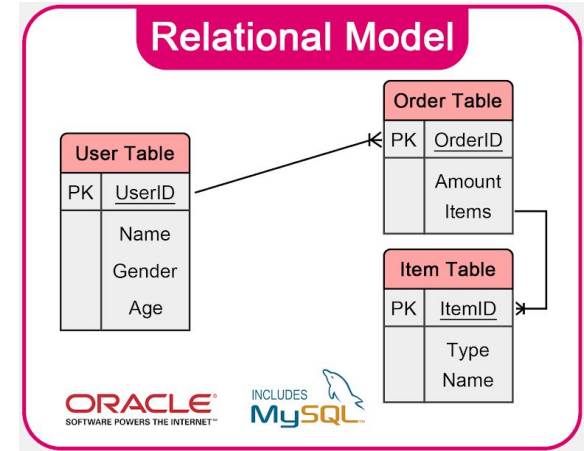
- **Description:**
 - Organizes data in a **tree structure** (each record has one parent and multiple children).
- **Strengths:**
 - Efficient for parent-child relationships.
- **Limitations:**
 - Rigid structure; does not support many-to-many relationships well.



Data Models

3. Relational Model

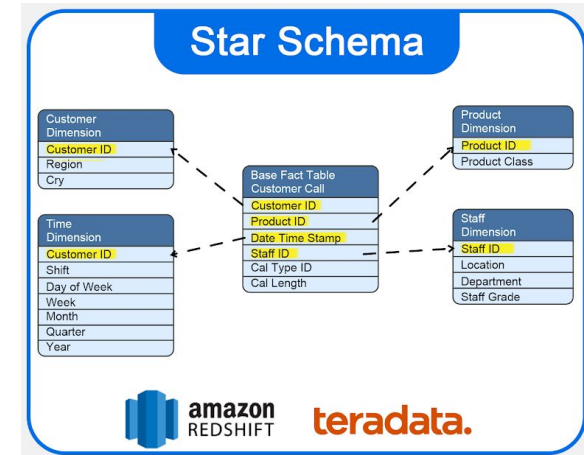
- **Description:**
 - Proposed by **E.F. Codd (1970)**.
 - Data is stored in **tables (relations)** with rows (tuples) and columns (attributes).
- **Strengths:**
 - Supports **keys**, normalization, and **SQL** querying.
 - Flexible, supports many-to-many relationships and complex queries.
- **Limitations:**
 - May require careful indexing and optimization at scale.



Data Models

4. Star Schema

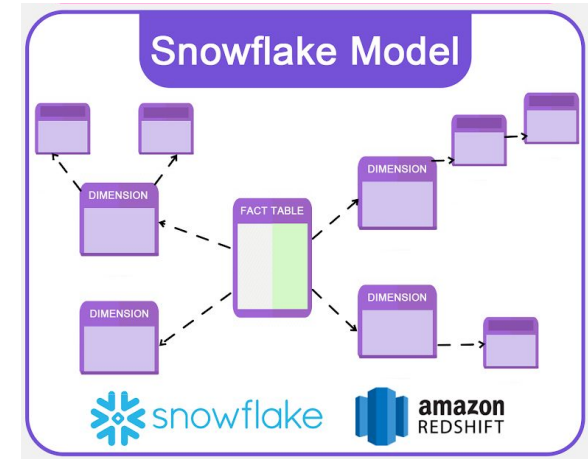
- **Description:**
 - Common in **data warehousing** and OLAP.
 - Central **fact table** connected to multiple **dimension tables**.
- **Strengths:**
 - Simple structure optimized for fast analytical queries.
- **Limitations:**
 - Some redundancy in dimension data.



Data Models

5. Snowflake Model

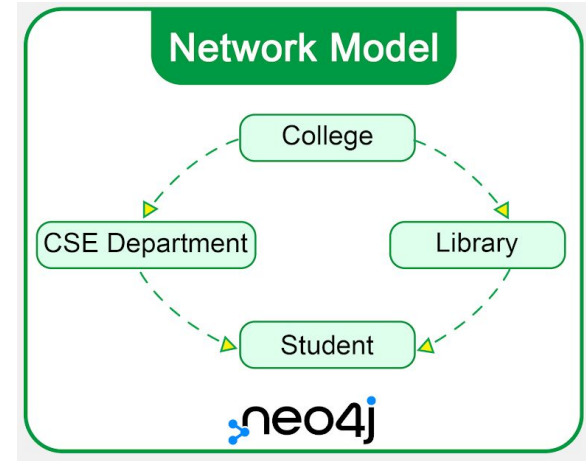
- **Description:**
 - A normalized variation of the star schema.
 - Dimension tables are split into multiple related tables.
- **Strengths:**
 - Reduces redundancy; improves data integrity.
- **Limitations:**
 - More complex queries due to additional joins.



Data Models

6. Network Model

- **Description:**
 - Forms a **graph-like structure** where records can have **multiple parents and children**.
- **Strengths:**
 - Efficiently models complex, many-to-many relationships.
- **Limitations:**
 - More complicated to design and maintain than relational or hierarchical models.



Database Isolation Levels

What Are They?

- **Isolation** ensures that each transaction executes **as if no other transactions are running concurrently**.
- Prevents conflicts and inconsistencies when multiple users access data at the same time.

Isolation Levels (From Lowest to Highest)

Read Uncommitted

- Transactions can read data **modified by other uncommitted transactions**.
- Allows **dirty reads**.

Database Isolation Levels

Read Committed

- A transaction only sees data **committed before the read begins**.
- Prevents dirty reads but allows **non-repeatable reads**.

Repeatable Read

- Data read during the transaction **remains the same** for the duration of that transaction.
- Prevents non-repeatable reads.
- May still allow **phantom reads** (new rows appearing in subsequent queries).

Serializable

- **Strictest isolation level.**
- Ensures that concurrent transactions behave **as if executed sequentially**.
- Prevents dirty reads, non-repeatable reads, and phantom reads.

Database Isolation Levels

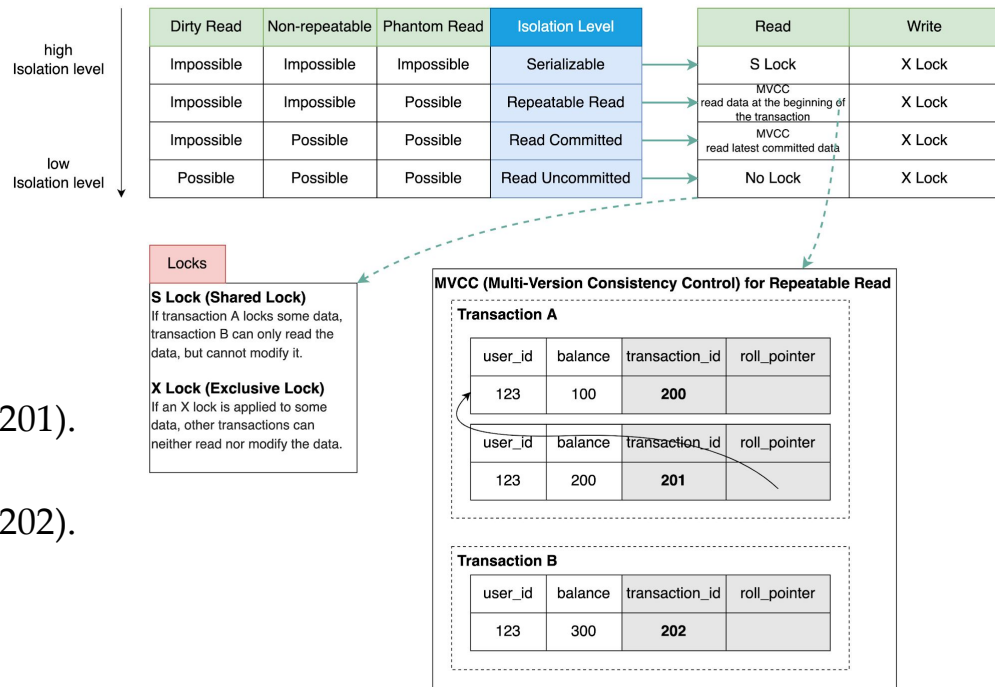
How Is Isolation Achieved?

- **MVCC (Multi-Version Concurrency Control)**
 - Each transaction gets a **snapshot** of the database.
 - Older versions of rows are retained for consistent reads.
- **Locks**
 - Shared locks for reading.
 - Exclusive locks for writing.

Database Isolation Levels

MVCC Example (Repeatable Read)

- Two transactions: **A** and **B**.
- Transaction A starts:
 - Creates a **Read View** (transaction_id=201).
- Transaction B starts:
 - Creates a **Read View** (transaction_id=202).
- Transaction A updates data:
 - New row created.
 - roll_pointer** points to the old version.
- Transaction B reads data:
 - Sees the **committed version prior to A's update**.
 - Even if A commits, B continues reading its consistent snapshot.



Database Isolation Levels

Why Do Isolation Levels Matter?

- They balance:
 - **Data consistency**
 - **System performance**
 - **Concurrency**
- Higher isolation:
 - **Stronger consistency guarantees**
 - **More locking and potential contention**

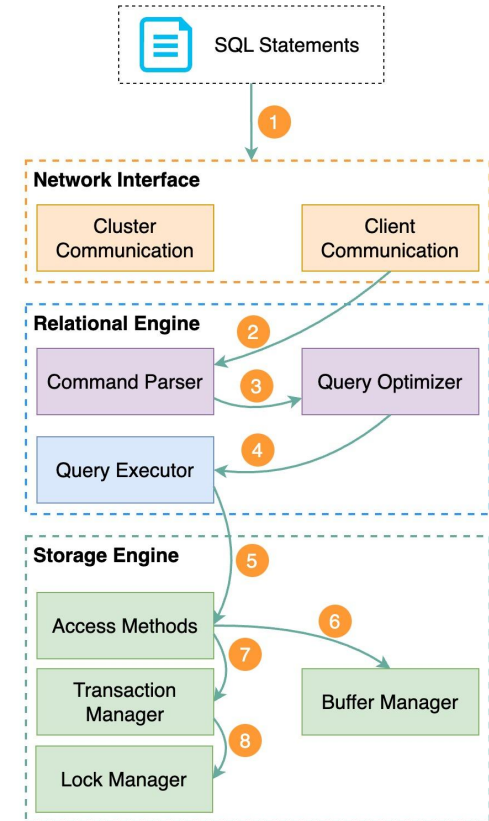
SQL Statement Execution in a Database

Executing an SQL statement involves multiple coordinated components of a database system.

While specific implementations vary across database engines, the following steps illustrate the **common processing flow**:

Step 1 – Transport Layer

- The SQL statement is **sent to the database** using a network protocol (e.g., TCP).
- Responsible for **communication and authentication** between the client and server.



SQL Statement Execution in a Database

Step 2 – Command Parser

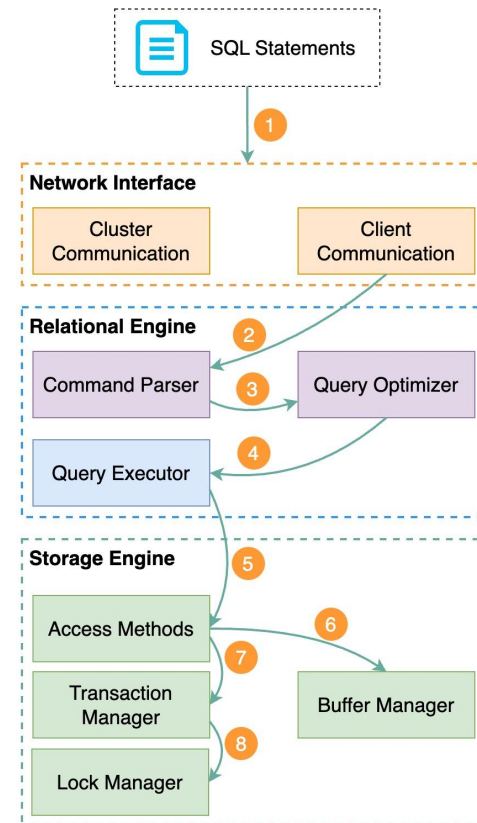
- The parser performs:
 - **Syntactic analysis** (validates SQL grammar)
 - **Semantic analysis** (ensures correctness of references and operations)
- Produces an **internal query tree representation**.

Step 3 – Optimizer

- The optimizer receives the query tree and:
 - Analyzes possible execution strategies.
 - Estimates **costs (I/O, CPU, memory)**.
 - Chooses the most efficient **execution plan**.

Step 4 – Executor

- The execution plan is passed to the **executor**.
- The executor coordinates actual **data retrieval or modification**.



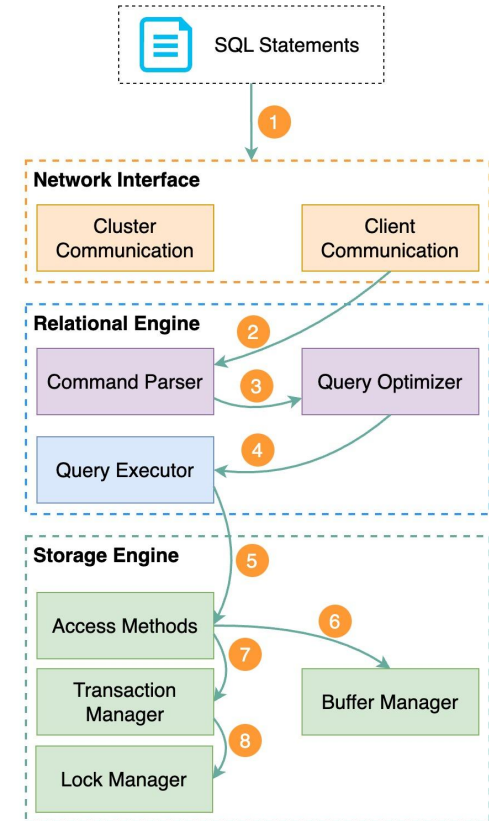
SQL Statement Execution in a Database

Step 5 – Access Methods

- Access methods define **how data is fetched**:
 - Scans (sequential, index)
 - Joins (nested loop, hash join)
- Interact with the **storage engine** to access physical data.

Step 6 – Buffer Manager (Read-Only Queries)

- For **read-only operations** (e.g., **SELECT**):
 - The buffer manager checks whether requested pages are **in memory (cache)**.
 - If not cached, pages are **loaded from disk**.



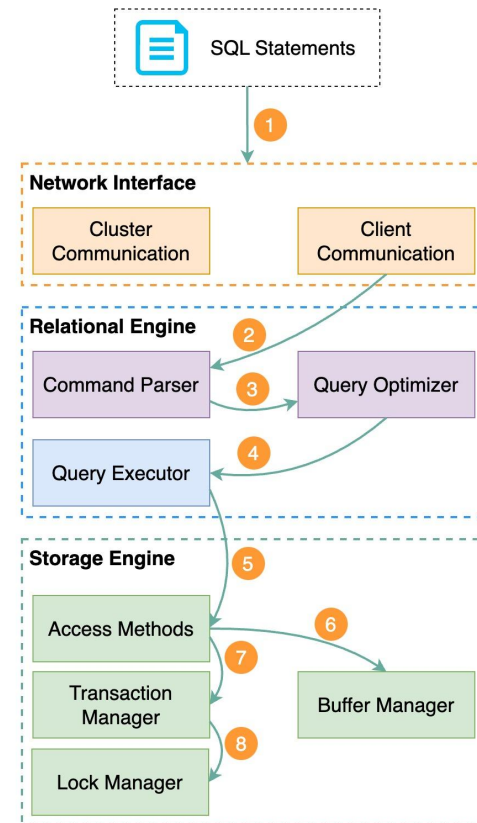
SQL Statement Execution in a Database

Step 7 – Transaction Manager (Update/Insert)

- For **write operations** (**INSERT**, **UPDATE**, **DELETE**):
 - The transaction manager handles:
 - **Atomicity** (ensuring all or nothing)
 - **Durability** (writing to logs)
 - Coordinates transaction commit or rollback.

Step 8 – Lock Manager

- The lock manager:
 - Acquires necessary **locks** to enforce isolation.
 - Prevents conflicts from concurrent transactions.
 - Ensures **ACID guarantees**.



Database Locks

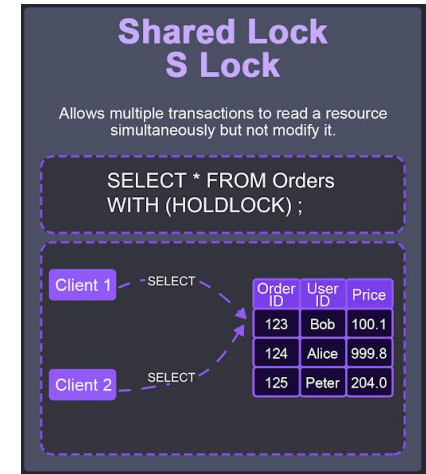
What Are Locks?

- **Locks** are mechanisms used in database systems to **control concurrent access to data**, ensuring:
 - Data integrity
 - Consistency
 - Isolation among transactions

Common Types of Locks

Shared Lock (S Lock)

- **Purpose:**
 - Allows **multiple transactions to read** a resource at the same time.
- **Behavior:**
 - No transaction can modify the resource while a shared lock is held.



Common Types of Locks

Exclusive Lock (X Lock)

- **Purpose:**
 - Allows a transaction to **read and modify** a resource.
- **Behavior:**
 - No other transaction can acquire any lock on the same resource concurrently.

Update Lock (U Lock)

- **Purpose:**
 - Used to **prevent deadlocks** when a transaction intends to upgrade from reading to writing.
- **Behavior:**
 - Compatible with shared locks, but only one update lock allowed per resource.

Exclusive Lock X Lock

- Allows a transaction to both read and modify a resource.
- No other transaction can acquire any type of lock on the same resource while an exclusive lock is held.

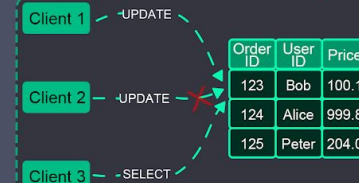
```
UPDATE Orders SET Price = 99.0  
WHERE OrderID = 123;
```



Update Lock U Lock

Used to prevent a deadlock scenario when a transaction intends to update a resource.

```
SELECT * FROM Orders WITH  
(UPDLOCK) WHERE OrderID = 123;
```



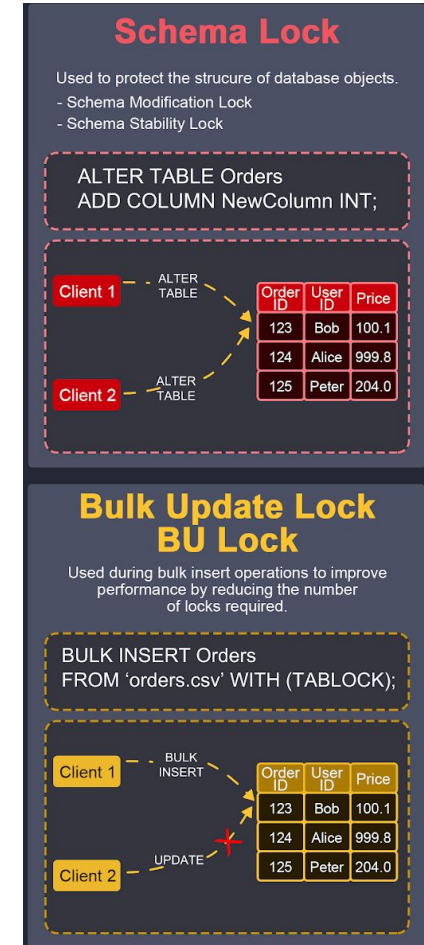
Common Types of Locks

Schema Lock

- **Purpose:**
 - Protects the **structure** of database objects (e.g., table definitions).
- **Use Case:**
 - Schema changes, such as ALTER TABLE operations.

Bulk Update Lock (BU Lock)

- **Purpose:**
 - Optimizes performance during **bulk insert or update operations**.
- **Behavior:**
 - Reduces the number of individual row locks.



Common Types of Locks

Key-Range Lock

- **Purpose:**
 - Prevents **phantom reads** when transactions query ranges of indexed data.
- **Behavior:**
 - Locks a range of keys, preventing inserts into that range by other transactions.

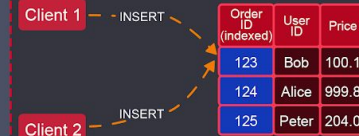
Row-Level Lock

- **Purpose:**
 - Locks a **specific row**, allowing concurrent access to other rows.
- **Benefit:**
 - Maximizes concurrency while protecting individual records.

Key Range Lock

Used in indexed data to prevent phantom reads (inserting new rows into a range that a transaction has already read).

```
SELECT * FROM Orders WHERE  
OrderID BETWEEN 100 AND 200  
WITH (HOLDLOCK, ROWLOCK);
```



Row-Level Lock

Locks a specific row in a table, allowing other rows to be accessed concurrently.

```
UPDATE Orders SET Price = 99.0  
WHERE OrderID = 123 WITH  
(ROWLOCK);
```



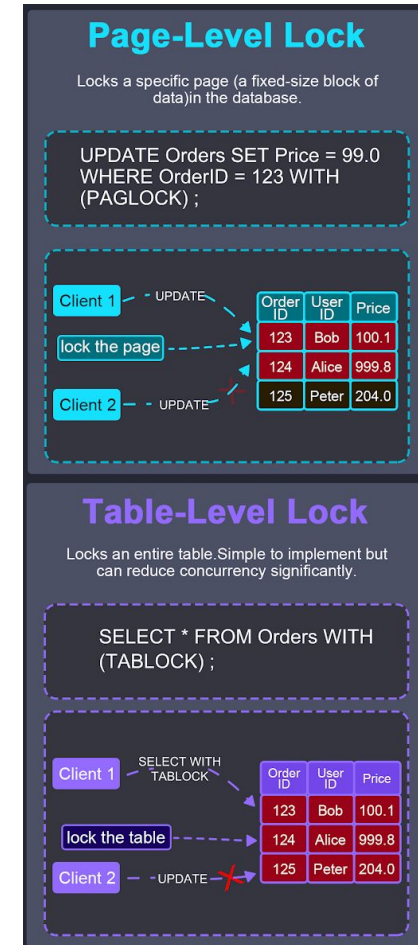
Common Types of Locks

Page-Level Lock

- **Purpose:**
 - Locks a **fixed-size block of data** (a “page”).
- **Trade-off:**
 - Fewer locks than row-level but less concurrency.

Table-Level Lock

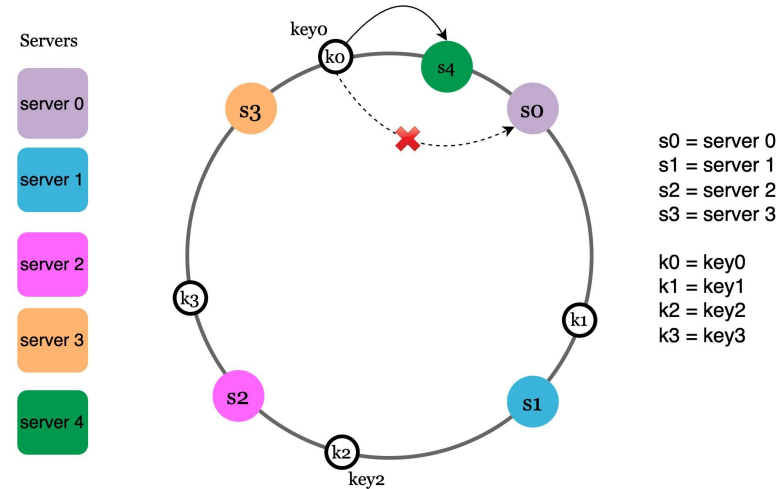
- **Purpose:**
 - Locks the **entire table**.
- **Benefit:**
 - Simple to implement.
- **Drawback:**
 - Greatly reduces concurrency.



Consistent Hashing

What Is Consistent Hashing?

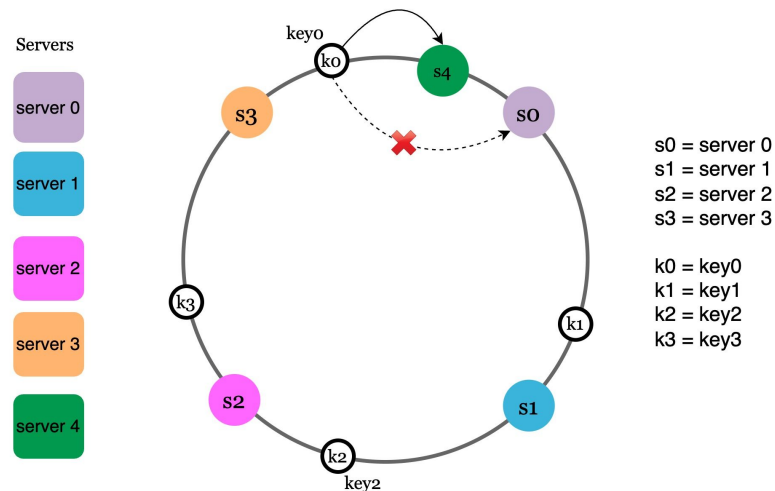
- **Consistent hashing** is a technique to **minimize the number of keys that must be moved** when the cluster membership changes.
- **Key goals:**
 - Keep most keys mapped to the same server.
 - Maintain even distribution of data.



Consistent Hashing

What Problem Does It Solve?

- In **distributed systems**, data must be **distributed evenly** across many servers.
- **Simple hashing** (e.g., $\text{hash}(\text{key}) \% N$) works *only if* the number of servers (N) is fixed.
- **When servers are added or removed:**
 - Simple hashing causes **large-scale data movement** (many keys get remapped).
 - This disrupts caches and increases load.



Consistent Hashing

How Does It Work?

1. Hash the servers:

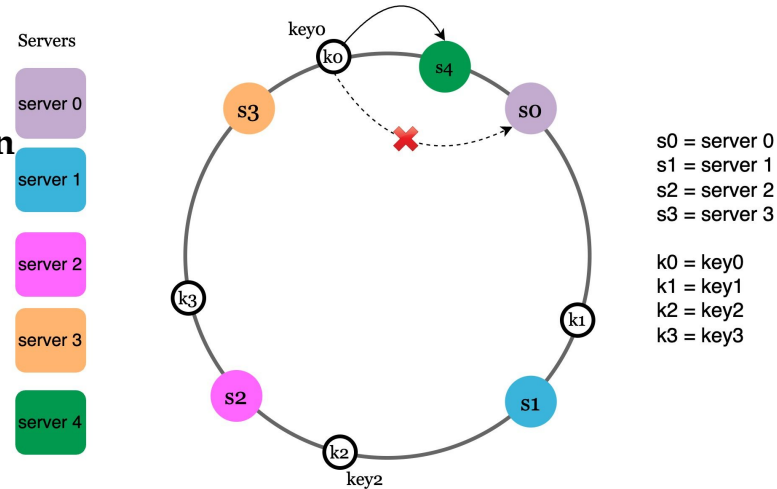
- Each server is hashed (by name or IP) and **placed on a logical ring**.

2. Hash the keys:

- Each object key is also hashed onto the same ring.

3. Locate the server:

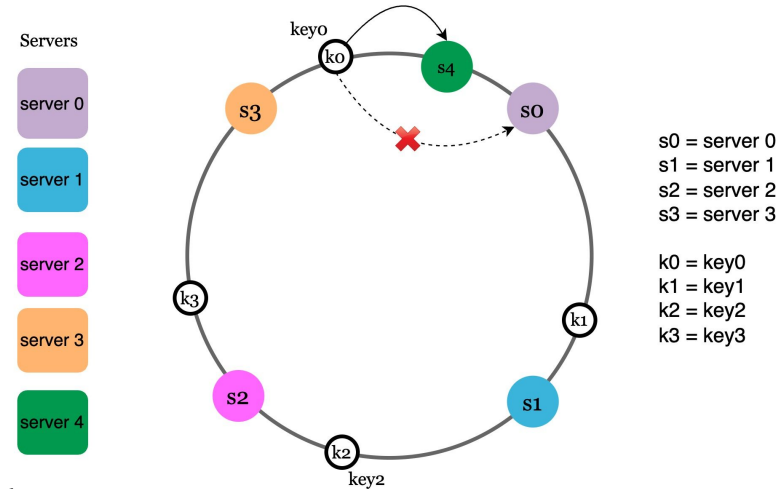
- To find where a key belongs:
 - Start from the key's position.
 - Move **clockwise** on the ring.
 - Assign the key to the **first server encountered**.



Consistent Hashing

Adding or Removing a Server

- **Example:**
 - A new server **s4** is added to the ring.
 - Only keys **falling between s4 and the previous server on the ring** must be moved.
 - All other keys remain unchanged.
- **Benefit:**
 - Minimal disruption (most keys stay on their original servers).



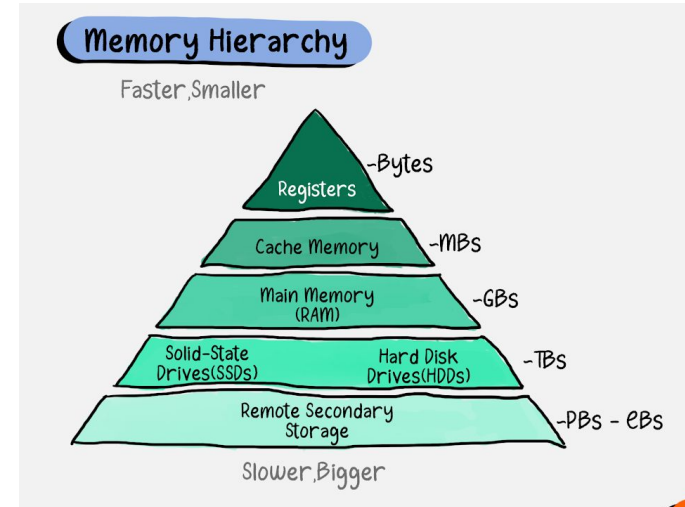
Types of Memory and Storage

Memory in computing systems forms a **multi-layered architecture**, balancing:

- Speed
- Capacity
- Cost

Understanding these layers allows system designers to:

- Optimize performance
- Improve efficiency
- Enhance user experience



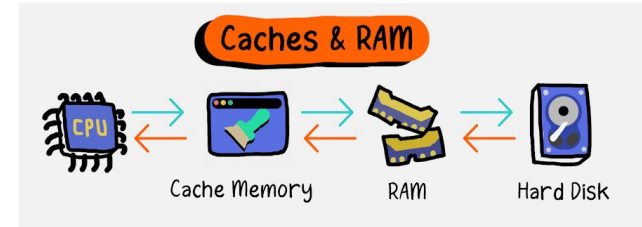
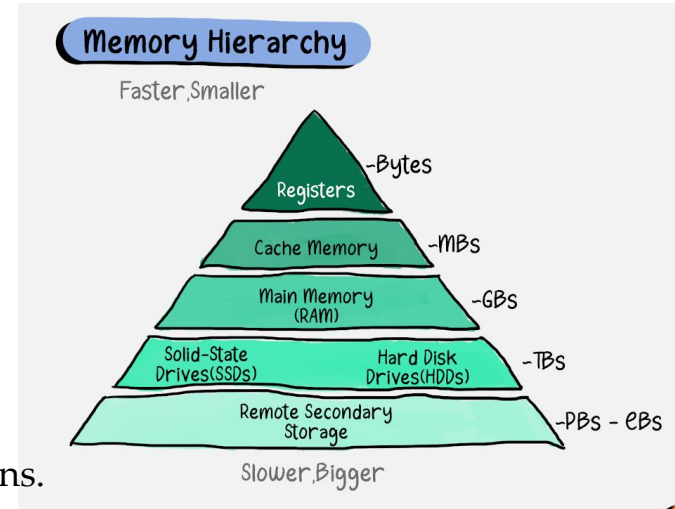
Common Memory Types

Registers

- **Description:**
 - Smallest and fastest memory, located **inside the CPU**.
- **Use:**
 - Holds immediate values for arithmetic and logic operations.

Cache Memory

- **Description:**
 - Small, high-speed memory **close to the CPU**.
- **Levels:**
 - L1, L2, L3 caches.
- **Use:**
 - Stores frequently accessed instructions and data.



Common Memory Types

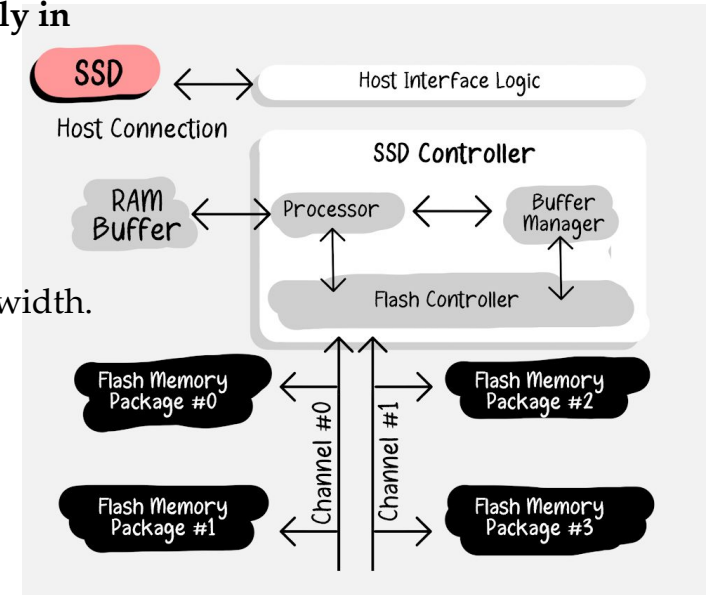
** Others are Solid-State Drives (SSDs), Hard Disk Drives (HDDs), Removable Storage, etc.*

Main Memory (RAM)

- **Description:**
 - **Volatile** memory used to store programs and data **currently in use**.
- **Types:**
 - **SRAM** (Static RAM) — faster, used for cache.
 - **DRAM** (Dynamic RAM) — slower, used for main system memory.
 - **DDR4/DDR5** generations provide higher speed and bandwidth.

Read-Only Memory (ROM)

- **Description:**
 - Non-volatile memory containing firmware.
- **Examples:**
 - **BIOS**, embedded system code.



Strategies to Scale Your Database

1. **Indexing**

Analyze your application's query patterns and create appropriate indexes to accelerate data retrieval.

2. **Materialized Views**

Precompute the results of complex queries and store them as materialized views to enable faster access.

3. **Denormalization**

Simplify data structures by reducing joins, trading some redundancy for improved query performance.

4. **Vertical Scaling**

Enhance the capacity of your database server by adding additional CPU cores, memory, or storage resources.

Strategies to Scale Your Database

5. Caching

Store frequently accessed data in a faster storage layer (e.g., in-memory cache) to reduce load on the primary database.

6. Replication

Create one or more replicas of your primary database on separate servers to scale read operations.

7. Sharding

Partition your database tables into smaller, independent segments distributed across multiple servers, enabling the scaling of both reads and writes.

What is Database Sharding?

- **Database sharding** is a form of **horizontal partitioning** where a large database is split into **smaller, faster, and more manageable parts called shards**.
- Each shard holds a **portion of the data**, and together, all shards represent the full dataset.

Why Use Sharding?

Too Much Data on One Machine

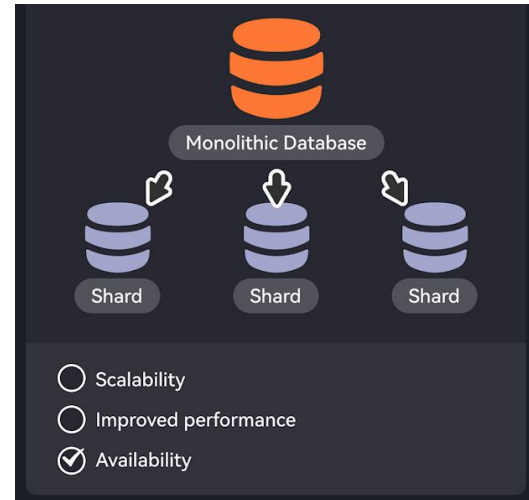
- A single server cannot efficiently store very large datasets.

Too Many Requests

- One database instance cannot handle all concurrent queries.

High Latency

- As data volume grows, query performance degrades.



How Sharding Works

- The dataset is divided into multiple **independent shards**.
- Each shard is stored on a **separate server or node**.
- The sharding strategy determines **which data belongs to which shard**.

Key Concepts

Sharding Key

- A column (or set of columns) used to decide **how data is distributed**.
- Example: `user_id`

Sharding Algorithm

- Logic that maps a sharding key to a shard.

How Sharding Works

Common Algorithms:

- **Range-Based Sharding**
 - Data is split by ranges (e.g., IDs 1–1000).
- **Hash-Based Sharding**
 - A hash function distributes data evenly.
 - Example: `shard_id = hash(user_id) % num_shards`
- **Directory-Based Sharding**
 - A lookup table maintains mappings between keys and shards.

Sharding Approaches

Application-Level Sharding

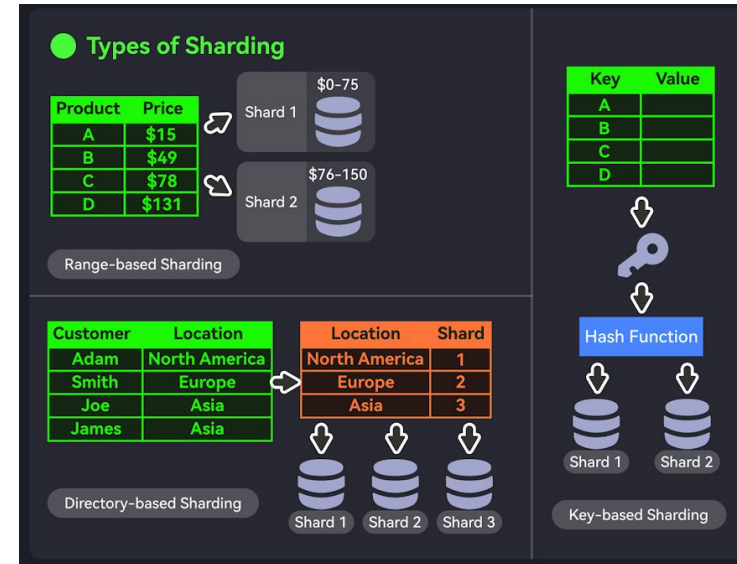
- The **application code** determines which shard to query or write to.

Middleware Sharding

- A **middleware layer** sits between the application and database, handling shard routing transparently.

Database-Native Sharding

- Sharding is built into the **database system itself** (e.g., MongoDB, Vitess).



Challenges of Sharding

Increased Complexity

- More moving parts in infrastructure and application logic.

Data Distribution

- Choosing an effective sharding key is critical to avoid hotspots and imbalance.

Joins and Transactions

- Cross-shard joins and transactions are harder to implement and less efficient.

Resharding

- Changing the sharding scheme later requires careful planning and data migration.

Read Replica Pattern with Middleware

- A **scaling strategy** where:
 - **Writes** go to a **primary database**.
 - **Reads** are served by **replica databases**.
- Improves:
 - **Read performance**
 - **Availability**
 - **Fault tolerance**

Read Replica Pattern with Middleware

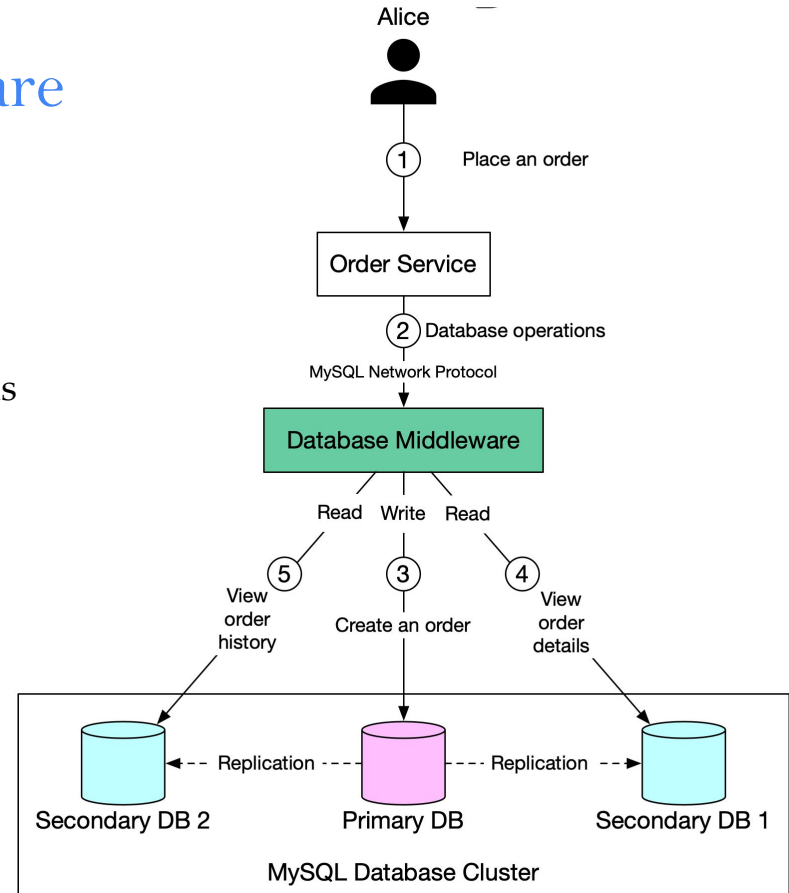
Implementation Approaches

1. Application-Level Routing

- The application **contains logic** to send reads to replicas and writes to the primary.

2. Database Middleware (Focus of This Module)

- A **middleware layer** transparently manages routing between the application and databases.

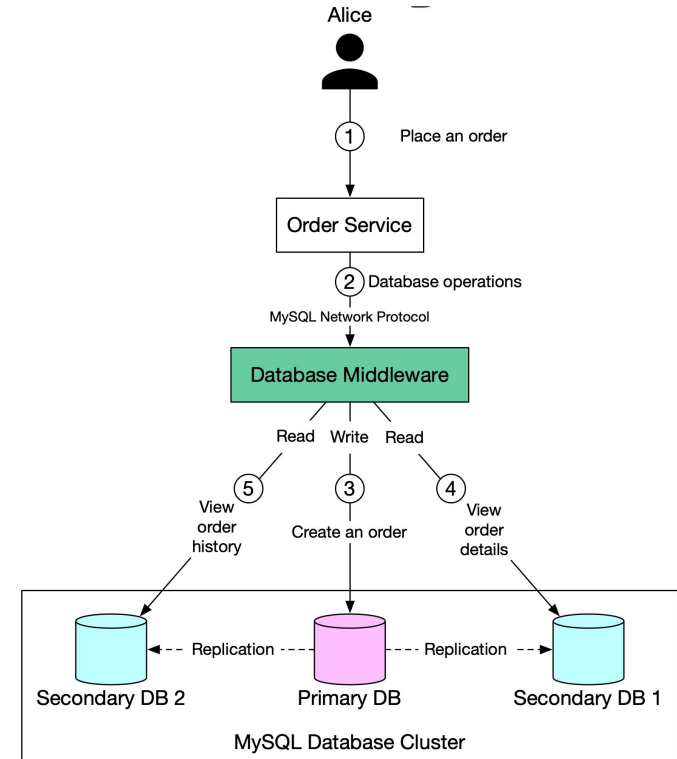


How Middleware-Based Read Replicas Work

1. **Application Query Flow:**
 - The application sends **all SQL statements to the middleware.**
2. **Routing Logic:**
 - The middleware **examines each query:**
 - **Writes** → routed to the **primary database.**
 - **Reads** → routed to **replica databases.**
 - Routing can be customized based on:
 - User
 - Schema
 - SQL statement type
3. **Replication:**
 - The primary **replicates data** to replicas asynchronously.
4. **Communication:**
 - Middleware communicates using the **standard MySQL network protocol**

Example Scenario

- **Alice places an order (write):**
 - The request is routed to the **primary database**.
- **Alice views order details (read):**
 - The request is routed to a **replica**.
- **Alice views order history (read):**
 - Again routed to a **replica**.



Read Replica Pattern with Middleware- Advantage

Simplified Application Code

- The application **does not need to be aware of database topology**.
- No need to implement routing logic in every service.

Better Compatibility

- Middleware uses **standard MySQL protocols**.
- Any MySQL-compatible client can connect without modification.
- Facilitates **easier database migration**.

Increase Database Performance

1. Indexing

Purpose:

- Accelerate data retrieval.

Considerations:

- Over-indexing can degrade write performance.
- Regularly review and optimize indexes to match evolving query patterns.

2. Query Optimization

Techniques:

- Use **EXPLAIN** to analyze and improve query execution plans.
- Avoid **SELECT *** to reduce unnecessary data transfer.
- Write precise, efficient **WHERE** clauses.

Increase Database Performance

3. Connection Pooling

Benefits:

- Reduces the overhead of repeatedly establishing database connections.
- Improves overall response times under load.

4. Caching

Levels:

- **Application-level:** In-memory caches like Redis or Memcached.
- **Database-level:** Query caching mechanisms to reduce repeated reads.

Increase Database Performance

5. Sharding

Definition:

- Distribute data across multiple databases or servers.

Use Cases:

- Scaling write throughput.
- Managing very large datasets efficiently.

6. Replication

Types:

- **Master-Slave:** One primary for writes, replicas for reads.
- **Master-Master:** Multiple nodes accepting writes.

Purpose:

- Improve read scalability.
- Enhance availability and fault tolerance.

Increase Database Performance

7. Hardware

Considerations:

- Sufficient RAM to keep working sets in memory.
- Fast storage (e.g., SSDs) for low-latency data access.
- Adequate CPU resources for query processing.

8. Monitoring

Metrics to Track:

- Query response times and throughput.
- CPU and memory utilization.
- Disk I/O performance and latency.

Increase Database Performance

9. Normalization and Denormalization

Normalization:

- Minimizes redundancy and improves data integrity.

Denormalization:

- Reduces join complexity to improve read performance (with some redundancy).

10. Partitioning

Types:

- **Horizontal Partitioning:** Splitting rows into separate tables or databases.
- **Vertical Partitioning:** Splitting columns into separate tables.

Purpose:

- Improve query performance.
- Simplify data management at scale.

Storage Systems (Three Broad Categories)

1. Block Storage

- **Description:**

- Provides **raw storage volumes** as blocks to servers.
- Servers can format these blocks into file systems or let applications (e.g., databases, virtual machines) manage them directly.

- **Key Characteristics:**

- **High performance and flexibility.**
- Each volume is **fully owned by a single server** (not shared).
- Can be attached:
 - **Physically** (HDDs, SSDs).
 - **Over a network** (Fibre Channel, iSCSI).

- **Use Cases:**

- Databases requiring fast random access.
- Virtual machine disk volumes.

Storage Systems (Three Broad Categories)

2. File Storage

- **Description:**
 - Provides a **higher-level abstraction** over block storage.
 - Data is stored as **files organized in hierarchical directories**.
- **Protocols:**
 - Common network protocols: **NFS, SMB/CIFS**.
- **Key Characteristics:**
 - **Simplifies access**—clients do not need to manage raw blocks.
 - Supports **multiple servers accessing shared files**.
- **Use Cases:**
 - Shared drives for teams and organizations.
 - General-purpose file sharing and collaboration.

Storage Systems (Three Broad Categories)

3. Object Storage

- **Description:**
 - Designed for **scalability, durability, and cost-efficiency**, especially for large volumes of unstructured data.
 - Stores data as **objects in a flat namespace** (no folders).
- **Access:**
 - Primarily via **RESTful APIs**.
- **Trade-offs:**
 - **Slower performance** compared to block and file storage.
 - Optimized for **cold or archival data**.
- **Examples:**
 - AWS S3, Google Cloud Storage, Azure Blob Storage
- **Use Cases:**
 - Backups and archives.
 - Media repositories.
 - Storing logs and large datasets.

Erasure Coding

- A **data protection technique** widely used in **object storage systems** such as **Amazon S3**.
- Unlike replication (copying full datasets), erasure coding:
 - Splits data into **chunks**.
 - Calculates **parity blocks** for redundancy.
 - Allows lost data to be **mathematically reconstructed**.

Example: (4 + 2) Erasure Coding

1. **Chunking**
 - Original data is divided into **4 data chunks**:
 - d1, d2, d3, d4

Erasure Coding

2. Parity Calculation

- **2 parity chunks** (p1 and p2) are computed using formulas.
- Simplified example:
 - $p1 = d1 + 2 \cdot d2 - d3 + 4 \cdot d4$
 - $p2 = -d1 + 5 \cdot d2 + d3 - 3 \cdot d4$

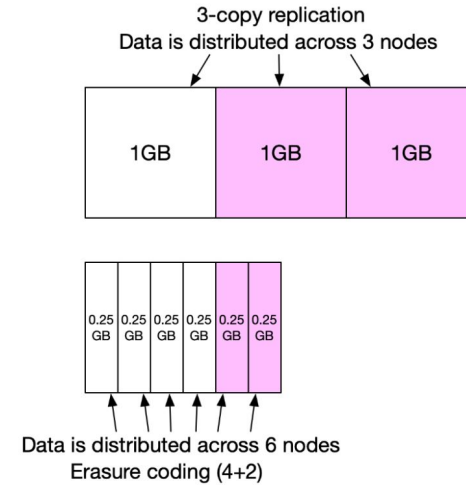
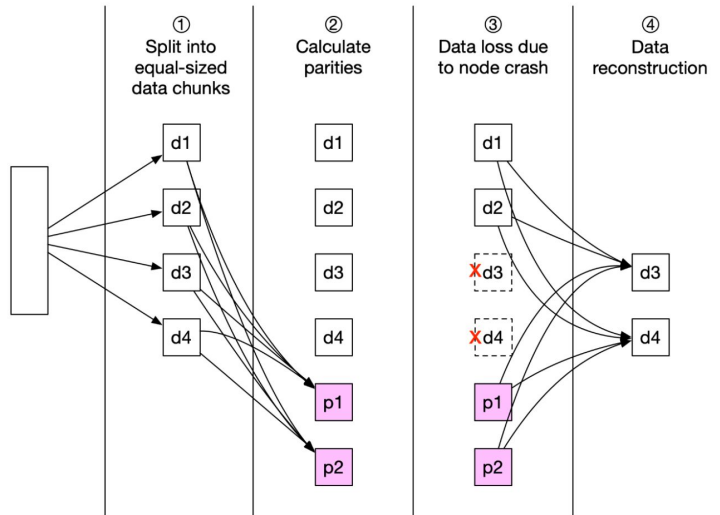
3. Failure Scenario

- Suppose **d3 and d4 are lost** due to node failures.

4. Recovery

- Using the **remaining chunks (d1, d2) and parities (p1, p2)**, the system solves equations to **rebuild d3 and d4**.

Erasure Coding / 3-copy replication



Erasure Coding

Technique	Redundancy Example	Storage Overhead
Erasure Coding	2 parity chunks per 4 data	50% (2 extra blocks for 4)
3-Copy Replication	3 full copies	200% (2 extra copies)

Key Advantages of Erasure Coding:

- **Much lower storage overhead** than replication.
- High **durability** across hardware failures.
- Common in cloud-scale systems storing large, less frequently accessed objects.

Where Is It Used?

- **Amazon S3**
- **Azure Blob Storage**
- **HDFS (Hadoop Distributed File System)**
- **Ceph**

Uploading large files Techniques

To **optimize uploading large files** (e.g., to S3), use **multipart upload**:

- **Why?**
 - Large files (GBs) are slow to upload and prone to failure.
 - Multipart upload allows splitting the file into parts, uploading them in parallel, and resuming if interrupted.

How it works:

1. **Initiate multipart upload** to get an upload ID.
2. **Split the file** into smaller parts (e.g., 200 MB each).
3. **Upload each part independently** with the upload ID.
4. **Verify each part** using ETags (checksums).
5. **Complete the upload** to assemble all parts into the final object.

Thank you!
Any Questions?