03 - Moving Beyond the Relational Model

Benefits of the Relational Model

- Mostly) Standard Data Model and Query Language
- ACID Compliance (Atomicity, Consistency, Isolation, Durability) ensures reliable transactions.
- Highly structured data support.
- Scalability for large amounts of data.
- Widespread adoption with extensive tooling and expertise available.

Relational Database Performance

- Many ways that a RDBMS (like a super-smart librarian that organizes and speeds up the way we find and store information) increases efficiency:
    - Indexing (the topic we focused on)
        - Instead of flipping through every page to find a word, you go to the index at the back, find the word, and jump straight to the right page.
    - Directly controlling storage
        - A database carefully manages where and how data is stored on a computer, making it quicker to access.
    - Column oriented storage vs row oriented storage
    - Query optimization
        - When you ask a database a question (a query), it figures out the fastest way to get the answer instead of checking everything one by one.
    - Caching/Prefetching
        - Caching saves frequently used data so the database doesn't have to look it up again and again.
        - Prefetching predicts what data you might need next and loads it early to speed things up.
    - Materialized views
        - A materialized view is a saved result of a complicated query, so the database doesn't have to recalculate it every time someone asks.
    - Precompiled stored procedures
        - A stored procedure is a saved piece of code that runs quickly when called, making repeated tasks faster.
    - Data replication and partitioning
        - Replication copies data across multiple servers so more people can access it quickly.
        - Partitioning splits data into smaller chunks and stores them separately so searches don't have to go through everything at once.

Transaction Processing

- Transaction: a sequence of one or more of the CRUD (create, read, update, delete) operations performed as a single, logical unit of work
    - Either the entire sequence succeeds (COMMIT)
    - OR the entire sequence fails (ROLLBACK or ABORT)
- Help ensure
    - Data integrity
    - Error recovery
    - Concurrency control
    - Reliable data storage
    - Simplified error handling

ACID Properties

- Atomicity: the transaction is treated as an atomic unit - it is fully executed or no parts of it are executed
- Consistency: a transaction takes a database from one consistent state to another consistent state, consistent state - all data meets integrity constraints
- Isolation: transactions do not interfere with each other.
- Durability: once committed, changes persist even after system failures.

Isolation & Concurrency Issues

Problems That Arise Due to Concurrent Transactions:

1. Dirty Reads: a transaction reads uncommitted changes from another transaction.
2. Non-Repeatable Reads: a transaction reads the same data twice but gets different results because another transaction modified it.
3. Phantom Reads: a transaction retrieves a set of records, but another transaction adds/deletes rows in that set before it finishes.

Example SQL Transaction: Money Transfer

```
DELIMITER //

CREATE PROCEDURE transfer(
   IN sender_id INT,
   IN receiver_id INT,
   IN amount DECIMAL(10,2)
)
BEGIN
```

```
    DECLARE rollback_message VARCHAR(255) DEFAULT 'Transaction rolled back: Insufficient
funds';
    DECLARE commit_message VARCHAR(255) DEFAULT 'Transaction committed successfully';

    -- Start the transaction
    START TRANSACTION;

    -- Attempt to debit money from sender
    UPDATE accounts SET balance = balance - amount WHERE account_id = sender_id;

    -- Attempt to credit money to receiver
    UPDATE accounts SET balance = balance + amount WHERE account_id = receiver_id;

    -- Check for insufficient funds
    IF (SELECT balance FROM accounts WHERE account_id = sender_id) < 0 THEN
        ROLLBACK;
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = rollback_message;
    ELSE
        COMMIT;
        SELECT commit_message AS 'Result';
    END IF;
END //

DELIMITER ;
```

Challenges of the Relational Model

Why RDBMS May Not Always Be the Best Choice:

- Evolving schemas – Some applications require flexible structures.
- ACID compliance overhead – Not all applications need strict transaction guarantees.
- Expensive joins – Queries involving multiple tables can slow down performance.

- Semi-structured & unstructured data – JSON, XML, and other flexible formats are hard to fit into rigid schemas.
- Horizontal scaling limitations – Traditional relational databases struggle with distributed architectures.
- Real-time & low-latency applications – Some use cases require databases optimized for speed over consistency.

Scalability: Vertical vs. Horizontal Scaling

Vertical Scaling (Scaling Up)

- Increases system capacity by upgrading hardware (CPU, RAM, SSDs).
- Pros: Easier to implement, no need to modify software.
- Cons: Expensive and has hardware limitations.

Horizontal Scaling (Scaling Out)

- Expands capacity by distributing the workload across multiple machines.
- Pros: More scalable, better redundancy, lower hardware costs.
- Cons: Requires changes in database architecture, introduces complexities like data replication and partitioning.

Distributed Databases & CAP Theorem

What is a Distributed System?

A collection of independent computers that appear as one system to the user.

Key Characteristics:

- Concurrent operations across multiple machines.
- Independent failures (one machine can crash without affecting others).
- No shared global clock.

Distributed Data Stores:

- Data is stored on multiple nodes and often replicated.
- Examples:
    - Relational (MySQL, PostgreSQL with sharding & replication).
    - NoSQL (MongoDB, Cassandra, CockroachDB).

Network Partitioning is Inevitable!

- Failures can occur at any time.
- Systems must be designed to handle partition tolerance (P) gracefully.

CAP Theorem: Trade-offs in Distributed Databases

The CAP Theorem states that a distributed system cannot achieve all three of the following properties simultaneously:

1. Consistency (C) – Every read gets the most recent write or an error. In other words, all nodes always return the same up-to-date data.
2. Availability (A) – Every request receives a response, but the response may not contain the most recent data. The system remains operational even if some nodes fail.
3. Partition Tolerance (P) – The system continues operating despite network failures that separate nodes into isolated groups.

Understanding CAP Theorem in Practice

- Consistency and Availability (CA): A system that guarantees both consistency and availability will fail if a network partition occurs. For example, traditional relational databases running on a single machine provide strong consistency and availability but cannot tolerate network failures across multiple servers.
- Consistency and Partition Tolerance (CP): A system that prioritizes consistency and partition tolerance will sacrifice availability. This means that if a network partition occurs, some requests will be rejected to ensure that all nodes return the most up-to-date data. Google Spanner is an example of a CP system.
- Availability and Partition Tolerance (AP): A system that prioritizes availability and partition tolerance may return outdated (eventually consistent) data but will always respond to requests, even during network failures. NoSQL databases like Cassandra and DynamoDB follow this model.

Since network partitions are inevitable in distributed systems, databases must make trade-offs between Consistency (C) and Availability (A) based on their use case requirements.

Real-World Implications of CAP Theorem

- Distributed databases must choose trade-offs based on application needs.
- No system is fully CP, CA, or AP – trade-offs depend on workload demands.
- Example:
    - Google Spanner prioritizes C + P (strong consistency, partitions tolerated).
    - Cassandra prioritizes A + P (high availability, eventual consistency).

Key Takeaways

- Relational databases excel at structured, high-integrity data storage.
- Distributed systems introduce trade-offs between consistency, availability, and partition tolerance.
- Choosing the right database depends on performance needs, scalability, and data consistency requirements.