Moving Beyond the Relational Model

Benefits of the Relational Model

- Provides a (mostly) standard data model and query language
- ACID compliant (Atomicity, Consistency, Isolation, Durability)
- Well-suited for highly structured data
- Can handle large volumes of data
- Mature technology with extensive tooling and industry experience

Relational Database Performance Enhancements

Multiple ways RDBMSs increase efficiency:

- Indexing (a focus in the course)
- Direct storage control
- Column-oriented vs row-oriented storage
- Query optimization
- Caching/prefetching
- Materialized views
- Precompiled stored procedures
- Data replication and partitioning

Transaction Processing

- Transactions are sequences of CRUD operations performed as a single logical unit
- All operations in a transaction either succeed (COMMIT) or fail (ROLLBACK/ABORT)
- Transactions help ensure:
    - Data integrity
    - Error recovery
    - Concurrency control
    - Reliable data storage
    - Simplified error handling

ACID Properties

1. **Atomicity**: Transactions are treated as indivisible units - either fully executed or not executed at all

2. **Consistency**: Transactions take databases from one consistent state to another (where all data meets integrity constraints)
3. **Isolation**: Concurrent transactions don't interfere with each other. Problems can arise when transactions interact:
    - Dirty Read: Reading uncommitted data from another transaction
    - Non-repeatable Read: Getting different values in same transaction when reading same data twice (because another transaction committed changes)
    - Phantom Reads: When records appear/disappear during a transaction due to another transaction's actions
4. **Durability**: Once committed, transactions are permanent, even if system failure occurs

Example Transaction

The presentation includes a comprehensive SQL example of money transfer between accounts with transaction safeguards:

- The transaction starts with `START TRANSACTION`
- Updates sender and receiver account balances
- Checks for sufficient funds
- Either commits with `COMMIT` or rolls back with `ROLLBACK` if conditions aren't met
- Includes logging transaction details in a separate table

Limitations of Relational Databases

Despite their benefits, relational databases may not be ideal for all scenarios:

- Schema evolution can be challenging
- Not all applications need full ACID compliance
- Join operations can be expensive
- Many modern data types are semi-structured or unstructured (JSON, XML)
- Horizontal scaling presents challenges
- Some applications need higher performance (real-time, low-latency systems)

Scaling Options

- **Vertical Scaling** (scaling up): Using bigger, more powerful systems
    - Easier as it requires minimal architecture changes
    - Has practical and financial limits
    - Often preferred initially until high-availability needs dictate otherwise
- **Horizontal Scaling** (scaling out): Distributed computing approach
    - More complex but potentially more cost-effective
    - Modern systems making this more manageable

Distributed Systems

- Definition: "A collection of independent computers that appear to users as one computer" (Andrew Tennenbaum)
- Key characteristics:
    - Computers operate concurrently
    - Computers fail independently
    - No shared global clock

Distributed Storage Approaches

Two main directions:

1. **Replication**: Creating complete copies of data across multiple nodes
2. **Sharding**: Dividing data into subsets across multiple nodes

Distributed Data Stores

- Data stored on multiple nodes, typically replicated (available on N nodes)
- Can be relational or non-relational
    - MySQL, PostgreSQL support replication and sharding
    - CockroachDB is noted as a newer player
    - Many NoSQL systems support distributed models
- Network partitioning is inevitable in distributed systems
    - Must have partition tolerance to keep running during network issues

The CAP Theorem

States that distributed data stores cannot simultaneously provide more than two of:

- **Consistency**: Every read receives most recent write or error
- **Availability**: Every request receives non-error response (but not necessarily most recent data)
- **Partition Tolerance**: System continues operating despite network issues

Database systems typically emphasize two of these properties:

- **CA Systems**: Traditional RDBMS (MySQL, PostgreSQL)
- **CP Systems**: MongoDB, HBase, Redis
- **AP Systems**: CouchDB, Cassandra, DynamoDB

CAP Trade-offs

- **Consistency + Availability**: System always has latest data but may struggle with network partitions
- **Consistency + Partition Tolerance**: Always provides latest data or drops request if unavailable
- **Availability + Partition Tolerance**: Always responds but may not have the most recent data

Practical Interpretation of CAP

The presentation points out that CAP is often oversimplified:

- Real meaning: "If you cannot limit faults, requests go to any server, and you insist on serving every request, consistency is impossible"
- Common interpretation: "Always sacrifice one: consistency, availability, or partition tolerance"