# Moving Beyond the Relational Model

Benefits of the Relational Model:
- **Standardized Data Model and Query Language**:
  Relational databases typically use Structured Query Language (SQL), providing a universal method to manage and query data.

- **ACID Compliance**:
  - **Atomicity**: Transactions complete entirely or not at all.
  - **Consistency**: Database remains in a valid state before and after transactions.
  - **Isolation**: Concurrent transactions don't interfere with each other.
  - **Durability**: Once transactions commit, changes are permanent even after system failures.

- Works efficiently with highly structured data, capable of handling large datasets.
- Well-established technology with extensive tooling and deep industry expertise.

---

Relational Database Performance:

RDBMSs improve efficiency through:
- **Indexing**: Speeds data retrieval operations.
- **Storage Management**: Direct control over data storage enhances performance.
- **Column vs. Row-Oriented Storage**: Optimizes data access based on usage patterns.
- **Query Optimization**: Automatically optimizes SQL queries for best performance.
- **Caching/Prefetching**: Reduces data access latency.
- **Materialized Views**: Stores query results for frequent or expensive queries.
- **Precompiled Stored Procedures**: Faster execution through precompiled logic.
- **Data Replication & Partitioning**: Enhances availability, fault tolerance, and parallelism.

---

Transaction Processing:

A **Transaction** is a group of database operations performed as a single logical unit, adhering to:
- **COMMIT**: Transaction succeeds entirely.
- **ROLLBACK** or **ABORT**: Transaction fails completely, undoing all changes.

Transactions ensure:
- Data Integrity
- Error Recovery
- Concurrency Control
- Reliable Data Storage
- Simplified Error Handling

---

ACID Properties (Detailed):

- **Atomicity**:
  Transactions are indivisible—either fully completed or fully aborted.

- **Consistency**:
  Transactions transition databases from one consistent state to another, maintaining integrity constraints.

- **Isolation**:
  Concurrent transactions don't adversely affect each other, preventing issues like:
  - **Dirty Reads**: Reading uncommitted changes.
  - **Non-repeatable Reads**: Inconsistent reads within the same transaction due to another committed transaction.
  - **Phantom Reads**: Different results in successive reads caused by concurrent insertions or deletions.

- **Durability**:
  Committed transactions persist through failures.

---

Isolation Issues Explained:

- **Dirty Read**:
  Occurs when a transaction reads data modified but not committed by another transaction. This can lead to inconsistencies if the uncommitted transaction is rolled back.

- **Non-repeatable Read**:
  A transaction reads data twice, getting different results because another transaction modified and committed the data between the two reads.

- **Phantom Reads**:

Rows appear or disappear between two reads within the same transaction due to inserts or deletes by concurrent transactions.

---

Example Transaction (Money Transfer):

Transactions exemplified by a bank transfer, demonstrating checks for sufficient funds, account debit, credit, logging of transactions, and error handling through rollback mechanisms if conditions fail.

---

Beyond Relational Databases:

Relational databases aren't ideal for:
- Rapid schema evolution.
- Applications where full ACID compliance is unnecessary.
- Expensive JOIN operations.
- Handling semi-structured/unstructured data (JSON, XML).
- Efficient horizontal scaling.
- Real-time or low-latency applications.

---

Scalability – Vertical vs. Horizontal:
- **Vertical Scaling** (scaling up): Adding resources to existing nodes, easy but limited.
- **Horizontal Scaling** (scaling out): Adding more nodes, offers better scalability and fault tolerance but adds complexity.

---

Distributed Systems:

Defined as a collection of independent computers appearing as one system.
Characteristics:
- Concurrent operations
- Independent failures
- Lack of a global clock

---

Distributed Data Storage:

Two main strategies:
- **Replication**: Same data stored on multiple nodes for redundancy and fault tolerance.
- **Sharding**: Data divided across nodes to improve scalability.

Relational and non-relational databases both offer replication and sharding (e.g., MySQL, PostgreSQL, CockroachDB, various NoSQL systems).

Network partitions are inevitable, making partition tolerance essential for system resilience.

---

CAP Theorem:

States a distributed data store cannot simultaneously guarantee all three:
- **Consistency**: Latest data on every read.
- **Availability**: Always returns a non-error response.
- **Partition Tolerance**: Continues operation despite network partitions.

Systems typically prioritize two at the cost of the third.

---

CAP Theorem in Practice:

- **Consistency + Availability (CA)**: Prioritizes latest data but struggles with network issues.
- **Consistency + Partition Tolerance (CP)**: Always returns latest data or no response during partitions.
- **Availability + Partition Tolerance (AP)**: Always responds, though data might be outdated.

Practical interpretation: You must prioritize which guarantees to sacrifice depending on application needs and fault tolerance requirements.