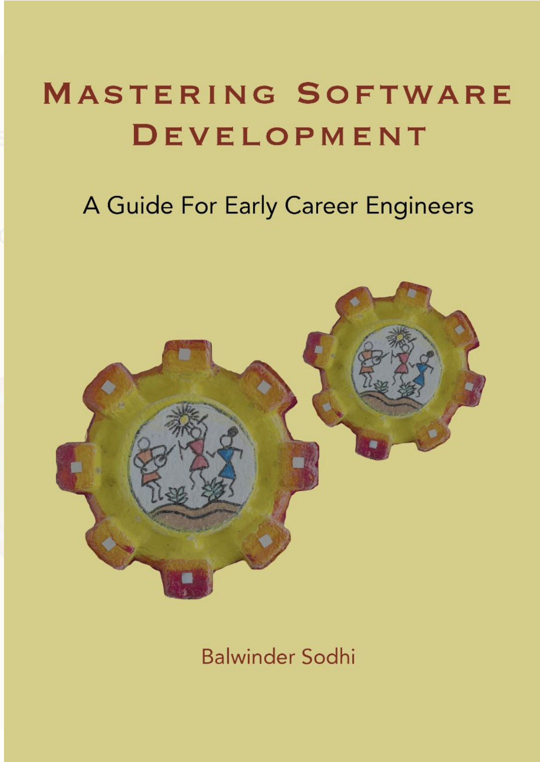


Database Design and Management

Balwinder Sodhi

Why Database Design Shapes System Quality

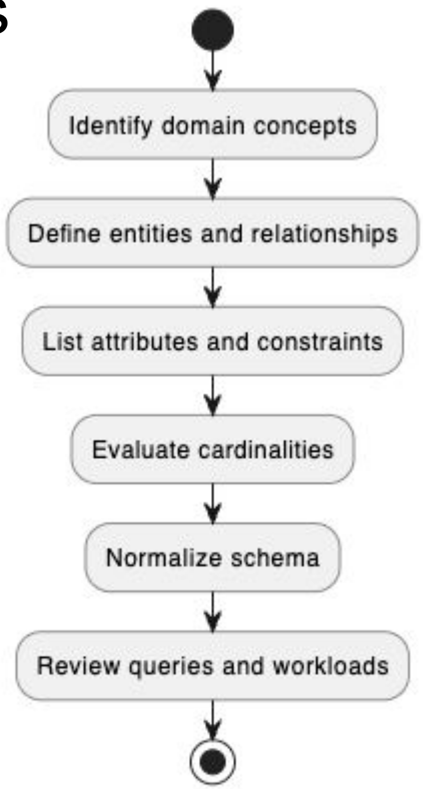
- Databases persist the most long-lived state of the system.
- Structural mistakes accumulate compound interest:
  - Poor joins → systemic latency
  - Unbounded growth tables → storage & performance bottlenecks
  - Incorrect relationships → inconsistent user flows
  - Wrong normalization → slow writes or slow reads

Data Modeling: Sketching Before Building

- Domain modeling identifies concepts, rules, boundaries.
- Data modeling transforms domain concepts into entities, attributes, constraints.
- Good modeling avoids premature optimization:
    - Keep schema expressive but simple.
    - Avoid leaking UI and API concerns into schema.

Example Data Modeling Thought Process

- **Domain:** Online Bookstore
- **Identify entities:** Users, Books, Orders, Payments, Inventory, Reviews
- **Identify rules:**
  - A user can have multiple orders
  - A book can have multiple reviews
  - An order can contain many books
- **Identify constraints:**
  - Stock quantity cannot go negative
  - Order total must equal sum of items



A Simple Online Bookstore

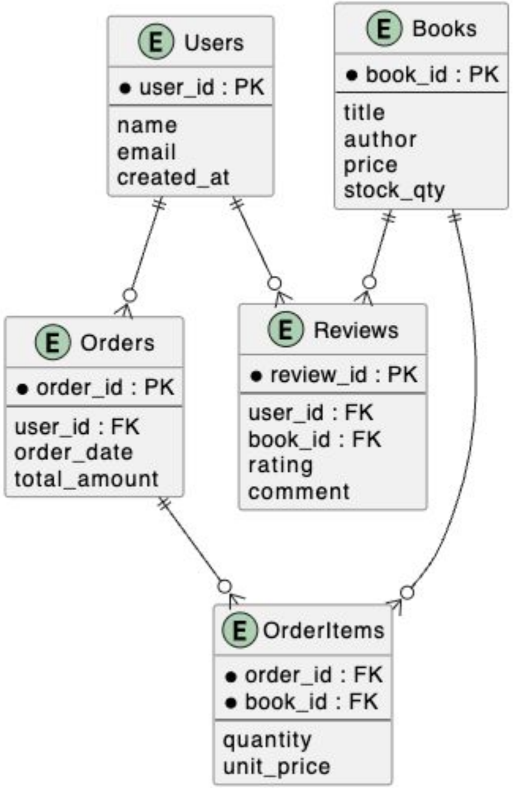
Core Entities

- Users
- Books
- Orders
- Payments
- Reviews
- Inventory

Typical Workflows

- User browses books → search/indexing considerations
- Creates an order → transactions, locking, consistency
- Reduces stock → constraints & triggers
- Writes reviews → preventing duplicates, spam protection

Online Bookstore Entities



Types of Relationships

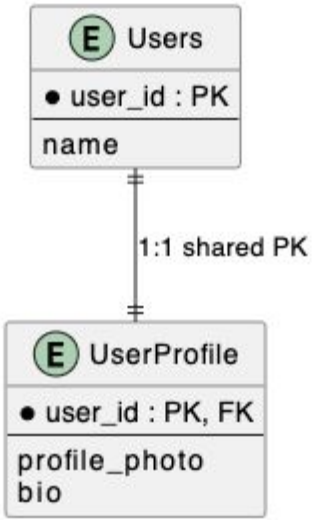
- One-to-One (1:1)
- One-to-Many (1:M)
- Many-to-Many (M:N)

One-to-One (1:1)

- Definition
  - Each row in Table A corresponds to at most one row in Table B.
- When to Use
  - Entities conceptually distinct but tightly coupled.
  - Sensitive or rarely accessed data separated for security/performance.
  - Very wide tables broken up into logical modules.
- Common Real-World Uses
  - Users → UserProfiles
  - Orders → OrderPayments
  - Employees → EmployeeConfidentialInfo

One-to-One (1:1)

- Implementation Patterns
  - **Shared Primary Key (strongest constraint)**
    B’s primary key is also a foreign key to A.
  - **Foreign key in one direction (soft 1:1)**
    B has a unique constraint on FK to A.
- Trade-offs
  - Shared PKs strictly enforce cardinality but add migration complexity.
  - Soft 1:1s offer flexibility but risk violating true uniqueness unless constraints in place.

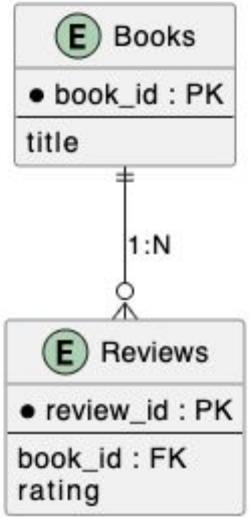


One-to-Many (1:N)

- Definition: One row in Table A connects to zero or more rows in Table B.
- It fits perfectly when you have a “most natural” relationship in business domains:
  - A customer → multiple orders
  - A book → many reviews
  - A device → many sensor readings
- Implementation
  - B contains a foreign key pointing to A.

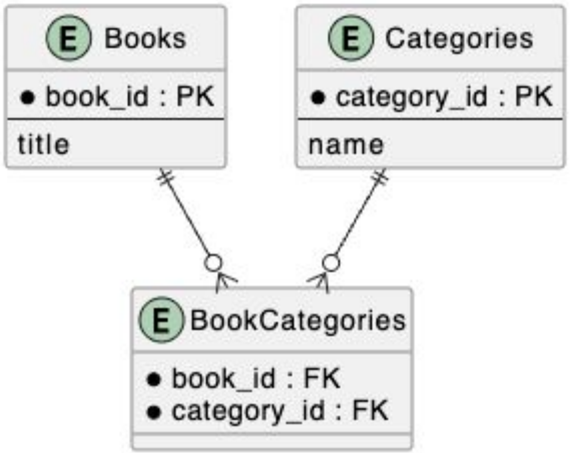
One-to-Many (1:N)

- Performance Considerations
  - FK lookup on B→A is fast if indexed.
  - Querying all children for parent is straightforward.
  - Cascading deletes require careful reasoning in production workloads.
- When It Becomes a Problem
  - Child table grows unbounded (e.g., logs, metrics, events).
  - FK introduces locking contention on writes.



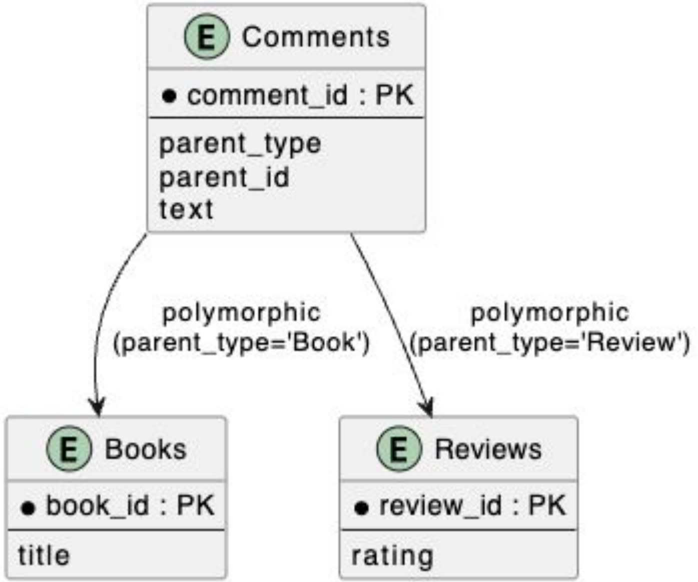
Many-to-Many (M:N)

- Definition: A row in A relates to multiple rows in B, and vice versa.
- Always implemented through a junction table (a.k.a. join table or association table).
- Examples:
  - `Students ↔ Courses`
  - `Books ↔ Categories`
  - `Users ↔ Roles`
- Advanced Practical Considerations
  - The join table can grow extremely large → impacts join performance.
  - Queries often require composite indexes (e.g., `(A_id, B_id)`).
  - Cascades on many-to-many relationships must be thought through carefully.



Polymorphic Relationships

- One table references multiple other tables via a “type” + “id” pair.
- Pros
  - Flexible
  - Reduces table explosion for similar concepts
- Cons
  - No FK constraints possible → referential integrity risk
  - Harder to query efficiently
  - Requires type-safe checks in app logic



Heuristics for Deciding Relationship Types

Ask these questions:

- Is this object dependent on another object to exist?
  - Yes → likely 1:N or 1:1 (composition)
  - No → maybe M:N or separate domain
- How does the domain describe the relationship?
  Natural language helps:
  - “A user has many orders”
  - “An order contains multiple items”
  - “A product belongs to a category”
  - These map 1:1 to schema choices.

Heuristics for Deciding Relationship Types

Ask these questions:

- What is the anticipated query pattern?
  - If you always query books with authors → embedding or denormalizing may help.
  - If you frequently need category filters → M:N join table with proper indexing.
- What is the data growth pattern?
  - Logs/events → enormous 1:N → consider vertical partitioning
  - Many-to-many between large domains → massive join tables → consider caching or search engine indexing

Hard Referential Integrity (Database-Enforced)

- Using foreign key constraints.
- Pros:
    - Guaranteed correctness
    - Catches bugs early
    - Maintains domain consistency
    - Enables cascading rules
- Cons:
    - Slows write-heavy systems
    - Causes locking issues
    - Hard to bulk-update or run large migrations
    - Multi-region DB setups complicate FK guarantees

Soft Referential Integrity (App-Enforced)

- Application code checks relationships; database stores plain IDs without FK constraints.
- Pros:
  - Faster writes
  - Easier sharding
  - Lower lock contention
  - Ideal for event-driven and append-only systems
- Cons:
  - Requires robust tests + observability
  - Risk of broken references
  - Developers must build cleanup tasks & orphan detection
- When Used in Practice
  - Twitter, Meta, etc. with massive write loads
  - Systems using Kafka-like async flows
  - Federated architectures with independent services

Cardinality Impacts Query Efficiency

- 1:1 → predictable joins, simplest queries
- 1:N → often OK, but watch child table size
- M:N → worst-case complexity increases dramatically
- Polymorphic → kills most optimizers

Normalization: Reducing Redundancy and Improving Integrity

Why Normalize?

- Prevent duplicate/contradictory data
- Make updates consistent by default
- Ensure schema represents domain constraints
- Reduce bugs from hidden transitive dependencies

Step-by-Step Normalization Example (Book + Author)

Imagine a CSV imported from legacy system (unnormalized data):

StudentID	StudentName	Courses	Instructor
1	Arun	Math, Physics	Prof. Singh
2	Barun	Math	Prof. Singh
3	Charlie	Physics, Chemistry	Prof. Singh, Prof. Nath

- Issues:
  - Multi-valued Courses field (violates 1NF)
  - Instructor data repeated for every enrolment
  - Instructor depends on Course (transitive dep)

1NF: Ensure Atomic Columns

To transform the table to First Normal Form (1NF) we ensure that the columns contain only atomic values as in table below:

StudentID	StudentName	Course	Instructor
1	Arun	Math	Prof. Singh
1	Arun	Physics	Prof. Singh
2	Barun	Math	Prof. Singh
3	Charlie	Physics	Prof. Singh
3	Charlie	Chemistry	Prof. Nath

Second Normal Form (2NF)

- We make sure it is in 1NF and that the non-key attributes must depend on the whole primary key.
- Current primary key is (`StudentID, Course`).
- To fix the problem, we decompose the table into two:

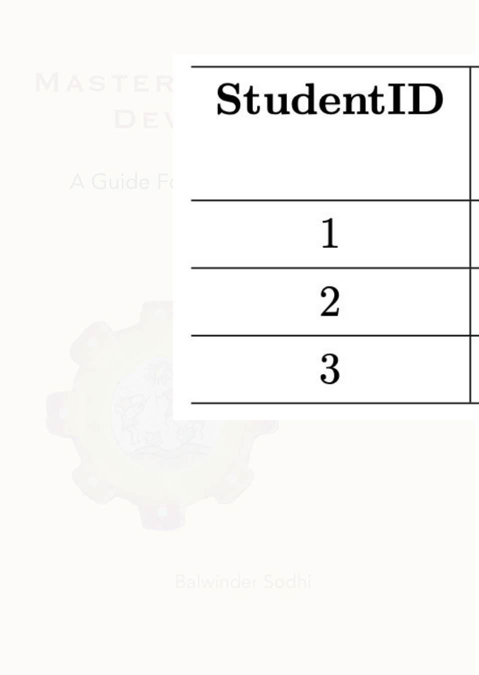
StudentID	Student Name
1	Arun
2	Barun
3	Charlie

StudentID	Course	Instructor
1	Math	Prof. Singh
1	Physics	Prof. Singh
2	Math	Prof. Singh
3	Physics	Prof. Singh
3	Chemistry	Prof. Nath

Third Normal Form (3NF)

- The problem with the above (2NF) tables is that `Instructor` depends on `Course`, not directly on the key (`StudentID, Course`).
- To fix it, we have to have the tables in 2NF **and** no transitive dependencies (i.e., non-key attributes shouldn't depend on other non-key attributes) in the table columns.
- Enforcing this idea gives us the Third Normal Form (3NF) as shown in the decomposed tables below.

Third Normal Form (3NF)



StudentID	Student Name
1	Arun
2	Barun
3	Charlie

StudentID	Course
1	Math
1	Physics
2	Math
3	Physics
3	Chemistry

Course	Instructor
Math	Prof. Singh
Physics	Prof. Singh
Chemistry	Prof. Nath

A table is in 1NF when

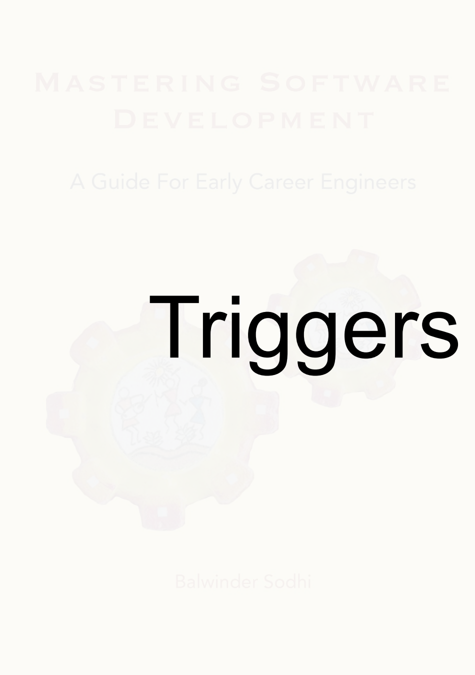
- All values are atomic (no repeating groups, arrays, or composite values).
- Each record is unique (no duplicate rows).
- Each column contains values of a single data type.
- The order of rows and columns does not matter.

A table is in 2NF when

- It is already in 1NF.
- Every non-key attribute is fully functionally dependent on the entire primary key.
  - This applies only when the primary key is composite.
  - No partial dependencies (i.e., a non-key attribute depending on only part of a composite key).

A table is in 3NF when

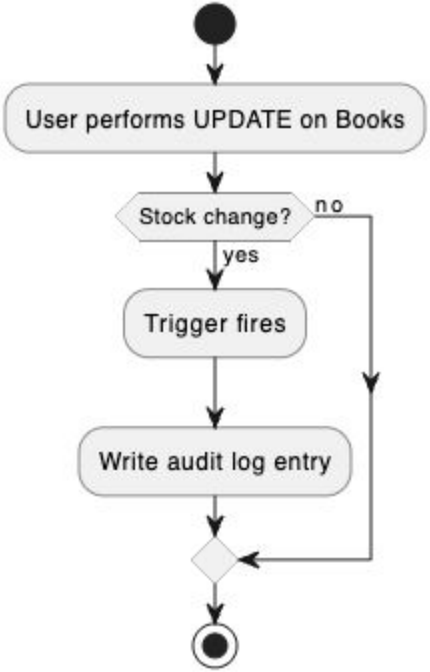
- It is already in 2NF.
- There are no transitive dependencies:
  - No non-key attribute depends on another non-key attribute.
  - All non-key attributes must depend directly and only on the primary key.



Triggers and Stored Procedures/Functions

What Triggers Are

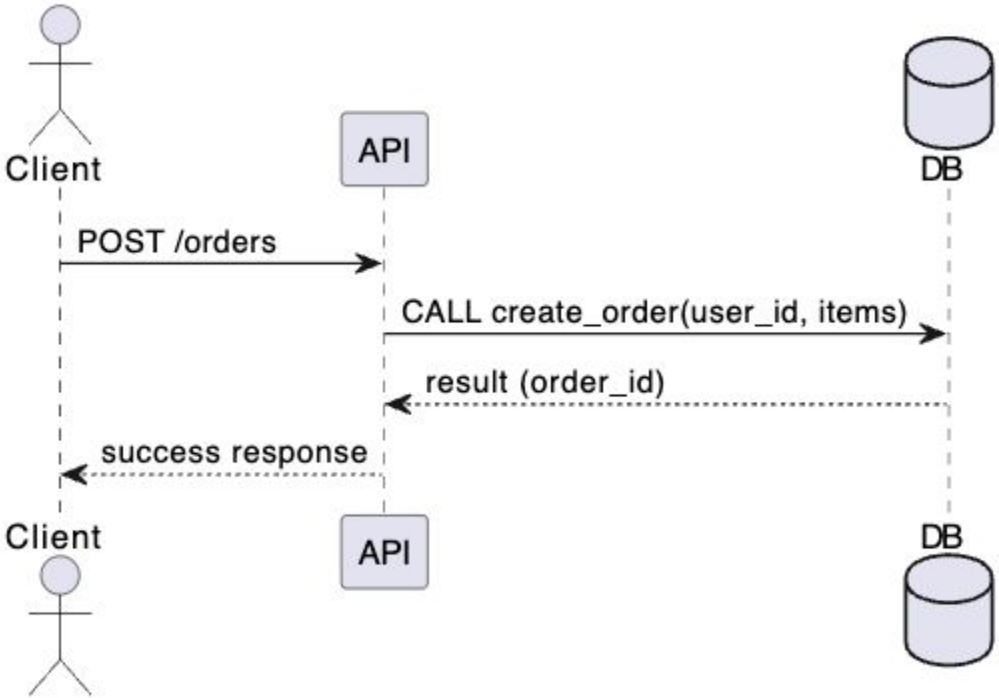
- Database-side automation reacting to INSERT/UPDATE/DELETE.
- Use cases:
  - Audit trails
  - Maintaining derived values
  - Enforcing complex constraints
  - Cascading behavior
- Pitfalls:
  - Harder to debug
  - Hidden logic → surprising for developers
  - Performance impact if overused
  - Avoid business logic in triggers



Stored Procedures & Functions: Reusable Database Code

- When to Use Them
  - Encapsulate multi-step operations (e.g., creating an order).
  - Improve performance by reducing network round trips.
  - Enforce security boundaries (role-based access).
  - Simplify batch updates / heavy data transforms.
- Best Practices
  - Keep them small and purpose-driven.
  - Version-control them like application code.
  - Avoid embedding business logic that should live in services.
  - Prefer idempotent operations where possible.

Stored Procedure Call Example



Advanced Trigger Use Cases

- Materialized view maintenance
- Change Data Capture (CDC) pipelines
- Audit history + temporal tables
- Cache invalidation (Redis, CDN)

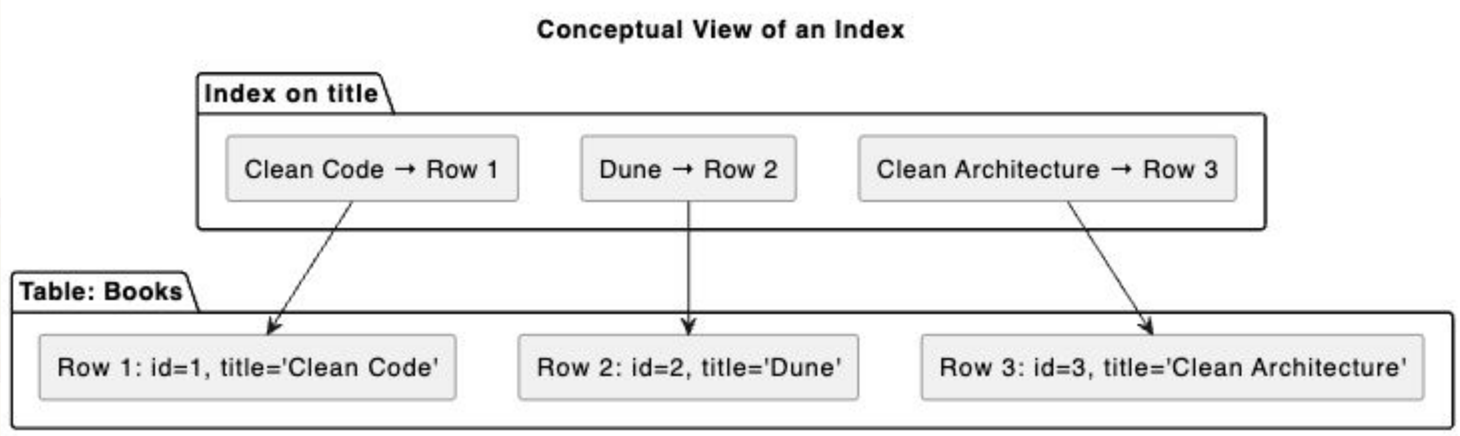
Avoid in Triggers:

- Calling external APIs (timeouts kill transactions)
- Heavy computation
- Cross-row loops (N^2 behavior)

What Is an Index?

- An index is a data structure that helps the database find rows faster, just like an index in the back of a book:
  - Without an index → the database reads the whole table (full scan).
  - With an index → the database jumps directly to relevant rows.
  - Indexes trade faster reads for:
    - slower writes (because index updates are needed)
    - extra storage
- Real-World Analogy
  - Think of a phone directory: names sorted alphabetically.
  - To find “Alice”, you don’t read every page — you jump directly to the A section.

Conceptual view



Why Indexes Make Queries Faster

- Indexes allow the database to avoid scanning irrelevant rows.

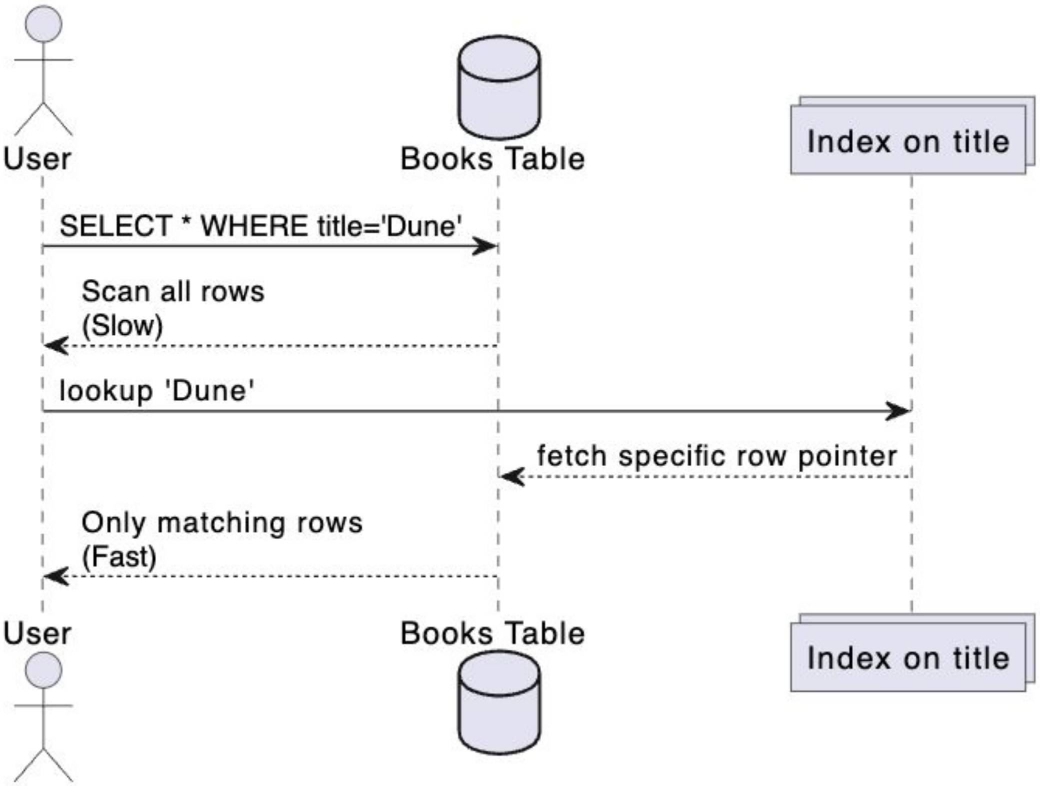
<u>Two Common Query Paths:</u>

- Without Index (Full Table Scan)
  - DB reads every row → checks the condition
  - Slow on large tables (O(n))
- With Index (Index Seek)
  - DB uses a sorted/searchable structure
  - Quickly locates matching values (O(log n))
  - Fetches only required rows

When Indexes Help

- Searching by a column (e.g., ISBN, email, username)
- Sorting (ORDER BY)
- Filtering (WHERE)
- Joins on foreign keys
- Partial match text search (GIN/GiST)

Full Scan vs Index Seek



Commonly Used Types of Indexes

- B-Tree indexes
  - Default in most RDBMS
  - Great for ranges, sorting, equality
- Hash indexes
  - Equality only
- GIN / GiST indexes
  - Document/JSONB search
  - Full-text search
  - Array membership queries
- Composite indexes
  - Multi-column indexing
  - Order matters
- Covering indexes
  - Index contains all fields needed → no table read

Transactions & Isolation Levels

ACID Principles

- Atomicity — all or nothing
- Consistency — valid state transitions
- Isolation — concurrent operations behave independently
- Durability — data survives crashes

Transaction Isolation Levels

Level	Prevents	Allows
Read Uncommitted	nothing	dirty reads
Read Committed	dirty reads	non-repeatable reads
Repeatable Read	dirty + non-repeatable reads	phantom reads (varies)
Serializable	all anomalies	slowest, uses locks/predicates

Anomaly Examples

- Dirty Read
  - Transaction A updates row
  - Transaction B reads uncommitted data
  - A rolls back → B saw ghost data
- Non-Repeatable Read
  - Transaction A reads row
  - Transaction B updates same row
  - A reads again → gets different result
- Phantom Read
  - Transaction A queries range
  - Transaction B inserts new matching rows
  - A re-queries → sees extra rows

Performance Tuning

Where Performance Problems Usually Come From

- Missing indexes
- Bad cardinality estimates
- Too many joins
- Unbounded scans
- Large transactional locks
- Poor schema normalization
- Hotspot rows (e.g., "global counters")

Query-Level Optimization

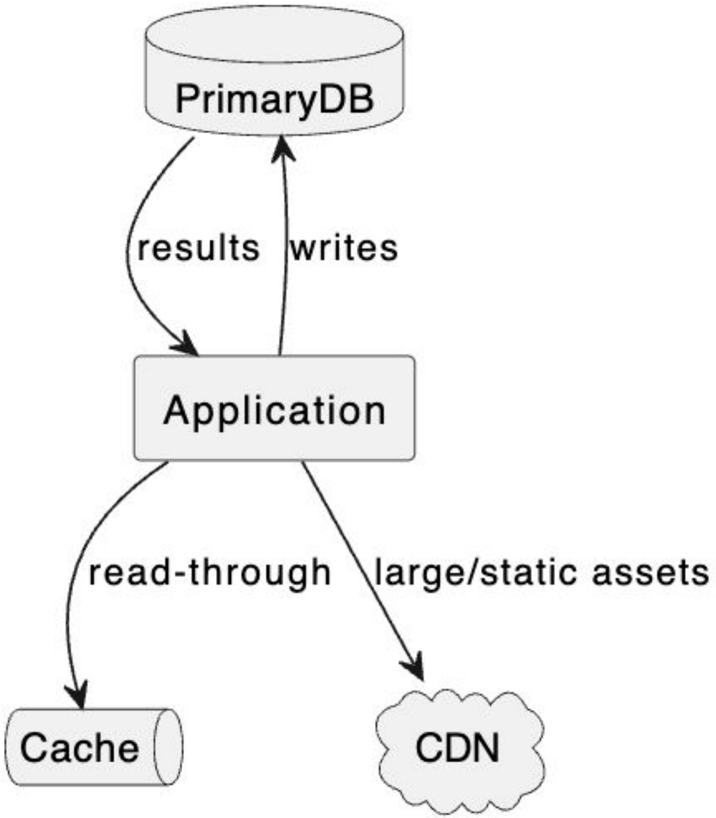
- Avoid SELECT *
- Filter early (in SQL, not app)
- Limit result sets
- Avoid unnecessary ORDER BY or DISTINCT
- Use materialized views for heavy analytic queries

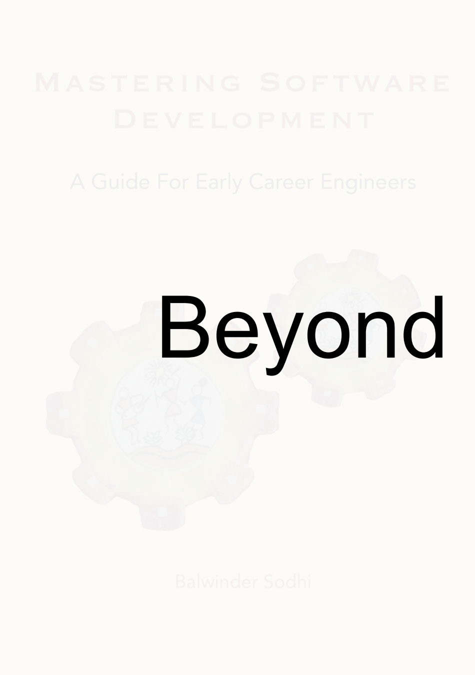
Schema-Level Optimization

- Use proper data types
- Partition large tables
- Avoid storing large blobs inline
- Introduce summary/rollup tables
- Use foreign key indexes

Application-Level Optimization

- Use caching layers (Redis)
- Batch writes
- Apply optimistic concurrency where possible
- Use read replicas for heavy read traffic





Beyond Relational: A Glimpse into NoSQL

Why NoSQL Exists

- Scaling relational systems horizontally is difficult.
- Modern applications store semi-structured and large-volume data.
- NoSQL trades strict schema + ACID for flexibility + scalability.

Major NoSQL Types

- Document Stores (MongoDB)
  - Great for nested data, variable schemas
  - Natural for JSON-based workflows
- Key-Value Stores (Redis)
  - Fast caching & ephemeral data
- Wide-Column Stores (Cassandra)
  - High write throughput
  - Tunable consistency
- Graph Databases (Neo4j)
  - Relationship-heavy domains (routing, recommendations)

Document vs Relational Example (Book example)

- Relational: `Books` table, `Authors` table, `BookCategory` join table.
- Document store equivalent:

```
{
  "book_id": "B1",
  "title": "Clean Code",
  "authors": ["Robert C. Martin"],
  "categories": ["software", "engineering"],
  "reviews": [
    { "user": "U1", "rating": 5, "comment": "Excellent." }
  ]
}
```

Choice of SQL vs NoSQL

- Choose NoSQL when
    - Schema flexibility needed.
    - Large-scale reads/writes.
    - Data naturally hierarchical or graph-like.
    - Event ingestion, logs, time-series workloads.
- When Not To
    - Strong consistency & transactions crucial.
    - Complex multi-table relationships that benefit from joins.

Key Takeaways

- Start with modeling → schema-first approach reduces long-term cost.
- Normalize first; denormalize carefully based on real workloads.
- Understand relationships deeply; they shape query performance.
- Use triggers & stored procedures thoughtfully.
- NoSQL offers flexibility and scale, but with different trade-offs.
- Database design is not static—schemas evolve with the product.