# General Architecture

**Two-Tier Architecture – Client & Server 🖥️➡️📊**

**Initially, database systems used a two-tier client-server model, where:**

* **The client (you, the data engineer) directly communicates with the database server to run queries.**
* **Example: Running SELECT \* FROM customers; on your SQL tool directly talks to the DBMS server to fetch data.**

**🚨 Problem?  
If multiple users access the database at once, performance degrades, and security is harder to manage.**

**Three-Tier Architecture – Adding a Middleman 🤝**

**To fix the issues of direct client-server communication, we introduced a middle layer:**

1. **Client (User Interface) – You write queries from an app or tool.**
2. **Application Server (Middle Layer) – Handles requests, security, caching before sending to DBMS.**
3. **Database Server – Actually stores and retrieves data.**

**🛠 Example:**

* **Instead of every user hitting the database directly, a web app or API acts as the middleman, improving scalability.**

**🚀 Why not four layers? Because three layers provide the right balance between flexibility, security, and performance without unnecessary complexity.**

**Three-Schema Architecture – A More Organized Library 📚**

**To manage databases better, we introduced three layers of abstraction:**

| **Schema Level** | **What it Represents** | **Analogy** |
| --- | --- | --- |
| **View Level (External)** | **What users see** | **Different bookshelves for different readers** |
| **Conceptual Level (Logical)** | **The database structure** | **The library’s master catalog** |
| **Physical Level** | **How data is stored** | **How books are arranged in storage rooms** |

**🛠 Example (as a Data Engineer)**

* **When you run SELECT \* FROM table; you’re in the View Level.**
* **When you create a table, you’re working in the Conceptual Level.**
* **When the database stores that table efficiently (indexes, partitions), it’s in the Physical Level.**

**Logical & Physical Data Independence – Why It Matters 🎯**

1. **Logical Data Independence → If we change the table structure (add a column), old queries should still work.**
2. **Physical Data Independence → If we change storage methods (heap → index), queries should still behave the same.**

**Why is this useful?  
Imagine if a library moved books to a better storage room, but readers could still find them without knowing the internal changes—that’s data independence!**

## Library Management System – Integrity Constraints in Action

**Imagine a library system where we store data about books, members, and borrowed books. Without proper constraints, the system can become inconsistent and unreliable. Let's walk through real-world issues and how integrity constraints fix them.**

* + 1. **Domain Constraint – Ensuring Valid Data**

**❌ Problem: What if someone enters a book with negative pages or a due date in the past?**

**✅ Solution: Domain constraints enforce valid data types and values.**

**Example:**

* **The Pages field must be a positive number.**
* **The Due\_Date must be a future date.**

**📝 SQL Code:**

CREATE TABLE Books (

Book\_ID INT PRIMARY KEY,

Title VARCHAR(255) NOT NULL,

Pages INT CHECK (Pages > 0), -- Prevents negative pages

Published\_Year INT CHECK (Published\_Year >= 1800) -- Books should have a valid year

);

**✅ Now, no book can have invalid pages or unrealistic publication years.**

* + 1. **Entity Integrity – Ensuring Uniqueness & Non-null Data**

**❌ Problem: What if a book is added without a Book\_ID, or two books have the same ID?**

**✅ Solution: Entity Integrity ensures each row is uniquely identifiable.**

**💡 Example:**

* **Each book must have a unique Book\_ID and cannot be NULL.**
* **Each member must have a unique Member\_ID.**

**📝 SQL Code:**

CREATE TABLE Members (

Member\_ID INT PRIMARY KEY, -- Ensures uniqueness

Name VARCHAR(100) NOT NULL -- Name cannot be NULL

);

**✅ Prevents duplicate or missing IDs, ensuring every record is unique.**

* + 1. **Referential Integrity – Maintaining Relationships**

**❌ Problem: What if a book is borrowed by a non-existent member?**

**✅ Solution: Referential Integrity ensures related records exist.**

**💡 Example:**

* **A borrowed book must belong to an existing member.**
* **If a member is deleted, their borrowed books must also be removed (CASCADE DELETE).**

**📝 SQL Code:**

CREATE TABLE BorrowedBooks (

Borrow\_ID INT PRIMARY KEY,

Book\_ID INT,

Member\_ID INT,

Borrowed\_Date DATE NOT NULL,

Due\_Date DATE NOT NULL,

FOREIGN KEY (Book\_ID) REFERENCES Books(Book\_ID),

FOREIGN KEY (Member\_ID) REFERENCES Members(Member\_ID) ON DELETE CASCADE

);

**✅ This prevents orphan records, ensuring every borrowed book is linked to a valid member.**

* + 1. **Key Constraints – Avoiding Duplicates**

**❌ Problem: What if two books have the same ISBN number?**

**✅ Solution: Key constraints ensure unique identifiers.**

**💡 Example:**

* **Each book should have a unique ISBN.**
* **No two members should have the same email.**

**📝 SQL Code:**

CREATE TABLE Books (

Book\_ID INT PRIMARY KEY,

Title VARCHAR(255) NOT NULL,

ISBN VARCHAR(13) UNIQUE -- ISBN should be unique

);

**✅ Now, no two books can have the same ISBN.**

* + 1. **Business Rule Constraints – Enforcing Library Rules**

**❌ Problem: What if a member borrows more than 5 books at a time?**

**✅ Solution: Business rule constraints enforce policies.**

**💡 Example:**

* **A member cannot borrow more than 5 books at a time.**

**📝 SQL Code:**

ALTER TABLE BorrowedBooks

ADD CONSTRAINT chk\_max\_books CHECK (

(SELECT COUNT(\*) FROM BorrowedBooks WHERE Member\_ID = BorrowedBooks.Member\_ID) <= 5

);

**✅ This ensures members cannot borrow more books than allowed.**

**📌 Summary: Why Are Integrity Constraints Important?**

**🚀 Data Accuracy → No negative pages or future-born books.  
🔑 Uniqueness → Every Book\_ID & Member\_ID is distinct.  
🔗 Relationships Intact → No missing members or books in borrowing records.  
📏 Business Rules Enforced → Library rules (like max borrow limit) are maintained.**

## Keys

| **Key Type** | **Definition** | **Example in Library System** | **Key Properties** |
| --- | --- | --- | --- |
| **Super Key** | A set of one or more columns that can uniquely identify a row. | {Book\_ID}, {ISBN}, {Book\_ID, Title} | 🔹 **Can have extra columns**  🔹 **Includes all possible unique keys** |
| **Candidate Key** | The minimal subset of a Super Key that uniquely identifies a row. | {Book\_ID}, {ISBN} | 🔹 **No unnecessary columns**  🔹 **Multiple Candidate Keys possible** |
| **Primary Key** | A chosen Candidate Key that uniquely identifies a row in the table. | {Book\_ID} | 🔹 **Must be unique**  🔹 **Cannot be NULL**  🔹 **Only one Primary Key per table** |
| **Alternate Key** | A Candidate Key that was **not selected** as the Primary Key. | {ISBN} (if {Book\_ID} is chosen as Primary Key) | 🔹 **Unique but not the Primary Key** |
| **Composite Key** | A key made of multiple columns to ensure uniqueness. | {Member\_ID, Book\_ID} in BorrowedBooks | 🔹 **Combination of columns**  🔹 **Used when a single column isn’t unique** |
| **Foreign Key** | A column that references the Primary Key in another table. | Book\_ID in BorrowedBooks referencing Book\_ID in Books | 🔹 **Establishes relationships**  🔹 **Ensures referential integrity** |
| **Unique Key** | Ensures column values are unique but allows NULL values. | Email in Members | 🔹 **Like a Primary Key but allows NULLs** |

## ACID Properties (Critical for Transactional Databases)

Transactional systems (like banking and e-commerce) require strict **ACID compliance**:

**a. Atomicity**

* A transaction is **all or nothing**.
* Example: Transferring money between accounts—both debit and credit operations must succeed together.

**b. Consistency**

* Ensures that the database remains in a valid state before and after a transaction.
* Example: A new Order cannot reference a non-existent Customer\_ID.

**c. Isolation**

* Transactions should not interfere with each other.
* Example: Two users booking the same airline seat should not cause conflicts.

**d. Durability**

* Once a transaction is committed, it must persist even if the system crashes.
* Example: A completed online order should not disappear after a server restart.

# Why to Data Model?

## 1. Introduction

Data modelling is a crucial technique in database design and architecture that helps **structure, organize, and define relationships** between data elements. It ensures **data consistency, integrity, and efficiency** in storage and retrieval.

In real-world scenarios, businesses handle vast amounts of structured and unstructured data. A well-structured data model is essential for scalability, performance, and analytical insights.

## 2. Problem Statement: The Client’s Challenge

Imagine a **retail company** that is expanding into e-commerce. The client wants to manage **product catalogues, customer orders, inventory, and payments** efficiently. However, they face the following challenges:

**Pain Points:**

1. **Data Redundancy** – Customer details are repeated in multiple order records.
2. **Data Inconsistency** – Different spellings of the same customer name exist.
3. **Performance Issues** – Queries take too long due to poorly designed tables.
4. **Scalability Constraints** – As the business grows, the current system cannot handle the increasing data volume.
5. **Lack of Relationships** – No clear linkage between products, customers, and transactions.

To solve these challenges, a **Data Engineer** proposes **Data Modelling** as the foundation for designing a scalable and efficient database.

## 3. Solution: Data Modelling

Data modelling involves defining:

* **Entities (Tables)** – Objects or concepts (e.g., Customers, Orders, Products).
* **Attributes (Columns)** – Characteristics of entities (e.g., Customer Name, Order Date).
* **Relationships** – How entities interact (e.g., A customer places multiple orders).

**Types of Data Models**

1. **Conceptual Data Model** – High-level business perspective of data.
2. **Logical Data Model** – Detailed structure with attributes, relationships, and constraints.
3. **Physical Data Model** – Implementation-ready model with specific database features (indexes, partitions).

## 4. Example: Retail E-Commerce Data Model

Let’s create a **simplified database model** for our client’s e-commerce business.

**Entities & Attributes**

**Customer Table**

|  |  |  |  |
| --- | --- | --- | --- |
| **Customer\_ID** | **Name** | **Email** | **Phone Number** |
| 1001 | Alice Doe | alice@email.com | 1234567890 |
| 1002 | Bob Smith | bob@email.com | 9876543210 |

**Product Table**

|  |  |  |  |
| --- | --- | --- | --- |
| Product\_ID | Name | Price | Stock Quantity |
| 2001 | Laptop | 700.00 | 20 |
| 2002 | Phone | 300.00 | 50 |

**Order Table**

|  |  |  |  |
| --- | --- | --- | --- |
| Order\_ID | Customer\_ID | Order\_Date | Total\_Amount |
| 5001 | 1001 | 2024-02-10 | 1000.00 |
| 5002 | 1002 | 2024-02-11 | 300.00 |

**Order\_Items Table**

|  |  |  |
| --- | --- | --- |
| Order\_ID | Product\_ID | Quantity |
| 5001 | 2001 | 1 |
| 5002 | 2002 | 1 |

**Relationships:**

* **One-to-Many:** A **Customer** can place **multiple Orders**.
* **Many-to-Many:** An **Order** can contain **multiple Products**, and a **Product** can be in multiple Orders (Handled using **Order\_Items table**).

## 5. Benefits of Data Modelling

✅ **Removes Redundancy** – Customer details stored in a single table. ✅ **Ensures Data Consistency** – No duplicate or conflicting information. ✅ **Improves Query Performance** – Indexed and normalized structure speeds up queries. ✅ **Enhances Scalability** – Supports business growth without performance degradation. ✅ **Facilitates Better Decision Making** – Structured data enables advanced analytics.

## 6. Conclusion

Data Modelling is a foundational step for any data-driven solution. By **identifying pain points, structuring data efficiently, and defining relationships**, a Data Engineer can create a **scalable, optimized, and future-proof** database for businesses.

This structured approach ensures that data remains **accurate, accessible, and ready for analytics** to support business decisions.