Introduction to Databases for Backend Development (MySQL)

---

1. Introduction to Databases

What is a Database?

A database is an electronic system used to store, organize, manage, and retrieve data efficiently. Instead of keeping information in scattered files, a database structures data in a way that makes it easy for software applications to access, update, and process it reliably. From banking to hospitals to e-commerce and social media, almost every digital system depends on databases to handle large volumes of user, transaction, and operational data.

Why Databases?

We need databases because modern applications generate massive, continuous streams of data that must be stored safely, accessed quickly, and updated efficiently. Databases allow organizations to maintain accurate records, build relationships between different data types, filter and search information instantly, support millions of users, and ensure secure, reliable data operations. Without databases, today's digital services—banking, healthcare, online shopping, social media, and IoT devices—could not function.

Where Are Databases Stored?

- **Dedicated Database Machine**
    - Special hardware only for database usage
    - High performance and security
    - Used in large enterprises
- **Cloud Hosting**
    - Database stored over the internet
    - Scalable, cheaper, reliable
    - Examples: AWS RDS, Firebase, MongoDB Atlas

Types of Databases

1. Relational Databases (RDBMS)

- Store data in tables (rows + columns)
- Uses SQL
- Uses Primary & Foreign Keys
- Examples: MySQL, PostgreSQL, SQL Server, Oracle

2. Object-Oriented Databases (OODBMS)

- Data stored as objects
- Supports OOP concepts
- Example hierarchy:
    - Person → Author
    - Person → Customer

3. Graph Databases

- Data stored in nodes & edges
- Used for highly connected data
- Examples: Neo4j, Amazon Neptune

4. Document Databases

- JSON-like documents
- Schema-less
- Examples: MongoDB, CouchDB

Alternative / Modern Database Types

1. NoSQL Databases

Designed for big data, high scalability, and flexibility.
Why NoSQL?

- Fast
- Flexible schema
- Handles large unstructured data
- Cloud friendly

Types:

- Document DBs → MongoDB
- Key-Value → Redis
- Wide-Column → Cassandra
- Graph DB → Neo4j

2. Big Data Databases

- For large-scale, distributed data
- Tools: Hadoop, Spark, HBase

3. Cloud Databases

- Hosted & managed online
- Auto-scaling, backups
- Examples: DynamoDB, Firebase

4. Business Intelligence Systems

- Used to analyse and visualize data
- Tools: Tableau, Power BI, Looker

Database Evolution (History Overview)

Timeline

- 1970s–1990s → Flat Files, Hierarchical, Network
- 1980s–Present → Relational Databases
- 1990s–Present → Object-Oriented
- 2000s–Present → NoSQL

Flat File Databases

- Simple text files
- No relationships
- Example: CSV

Hierarchical Databases

- Tree structure (one-to-many)
- Example: 1 student → many courses

Network Databases

- Graph structure (many-to-many)
- Example: teacher ↔ course

Relational Databases

- Tables
- SQL
- Primary & foreign keys
- Example:
    - PROFESSOR
    - COURSE
    - CLASS (linking both)

Object-Oriented Databases

- Data stored as objects
- Supports inheritance

NoSQL Databases

- Schema-less
- For large web-scale systems

---

2. Introduction to SQL

What is SQL?

SQL (Structured Query Language) is used to store, update, retrieve, and manage data in relational databases.

How SQL Works with DBMS?

When a SQL statement is executed, the **DBMS** interprets, reads, and processes it internally.

Role of DBMS:

- Parses SQL commands
- Checks syntax
- Executes operations

- Manages users, storage, and security

Advantages of SQL

- Easy to learn
- Fast and efficient
- Portable
- Case insensitive.
- Supports multiple queries and joins

Common Uses of SQL

SQL is used to perform various tasks in a database. Some of the most common operations include:

- **Creating databases**

```sql
CREATE DATABASE database_name;
```

- **Inserting data into tables**

```sql
INSERT INTO table_name(column1, column2) VALUES (value1, value2);
```

- **Updating existing records**

```sql
UPDATE table_name
SET column_name = 'value'
WHERE id = 1;
```

- **Filtering, sorting, and retrieving data**

```sql
SELECT * FROM students
ORDER BY marks DESC;
```

- **Joining multiple tables**

```sql
SELECT e.name, d.department_name
FROM employees e
INNER JOIN departments d ON e.dept_id = d.dept_id;
```

SQL Subsets

- **DDL (Data Definition Language)** → CREATE, ALTER, DROP

  Key Commands

  - **CREATE** – Create a database or table

    ```sql
    CREATE TABLE table_name (
        column1 datatype,
        column2 datatype
    );
    ```

  - **DROP** – Delete a table

    ```sql
    DROP TABLE table_name;
    ```

  - **ALTER** – Add, modify, or delete columns

    ```sql
    ALTER TABLE table_name ADD column_name datatype;
    ```

  - **TRUNCATE** – Remove all rows from a table

    ```sql
    TRUNCATE TABLE table_name;
    ```

- **DML (Data Manipulation Language)** → INSERT, UPDATE, DELETE

  Key Commands

  - **INSERT** – Add new records

    ```sql
    INSERT INTO table_name (col1, col2) VALUES (val1, val2);
    ```

  - **UPDATE** – Modify existing records

    ```sql
    UPDATE table_name SET col1 = val WHERE id = 1;
    ```

  - **DELETE** – Remove records

    ```sql
    DELETE FROM table_name WHERE id = 1;
    ```

- **DQL (Data Query Language)** → SELECT

Command: `SELECT`

- **SELECT** – Fetch data

```sql
SELECT * FROM table_name;
```

- **DCL (Data Control Language)** → GRANT, REVOKE

Commands: `GRANT`, `REVOKE`

- **GRANT** – Give access

```sql
GRANT SELECT ON table_name TO user_name;
```

- **REVOKE** – Remove access

```sql
REVOKE SELECT ON table_name FROM user_name;
```

- **TCL (Transaction Control Language)** → COMMIT, ROLLBACK

Commands: `COMMIT`, `ROLLBACK`

- **COMMIT** – Save changes permanently

```sql
COMMIT;
```

- **ROLLBACK** – Undo changes

```sql
ROLLBACK;
```

---

3. Basic Database Structure

Key Components of Database Structure

- **Tables** → store data
- **Fields (Columns)** → attributes

- **Records (Rows)** → individual data entries
- **Attributes** → properties describing entities
- **Primary Key** → unique row identifier

Types of Keys

Primary Key → Unique & Not Null

A **Primary Key** uniquely identifies each record in a table and cannot contain NULL values. It ensures every row can be uniquely referenced.

SQL Example

```sql
CREATE TABLE Students (
  student_id INT PRIMARY KEY,
  name VARCHAR(50)
);
```

Foreign Key → References Another Table

A **Foreign Key** is a column in one table that refers to the **Primary Key** of another table. It creates a relationship between tables.

SQL Example

```sql
CREATE TABLE Orders (
  order_id INT PRIMARY KEY,
  student_id INT,
  FOREIGN KEY (student_id) REFERENCES Students(student_id)
);
```

Composite Key → Multiple Columns Form PK

A **Composite Key** uses **two or more columns** together to uniquely identify a record when one column alone is not enough.

SQL Example

```sql
CREATE TABLE Enrollment (
  student_id INT,
  course_id INT,
  PRIMARY KEY (student_id, course_id)
);
```

Candidate Key → Can Qualify as Primary Key

A **Candidate Key** is any column that has **unique** values and **could** be chosen as the Primary Key.

SQL Example

sql Copy code

```sql
CREATE TABLE Staff (
  staff_id INT UNIQUE,
  email VARCHAR(100) UNIQUE,
  contact_no VARCHAR(20),
  PRIMARY KEY (staff_id)   -- email can also be a candidate key
);
```

Alternate Key → Candidate Key Not Selected

An **Alternate Key** is a **candidate key that was not chosen** as the primary key.

SQL Example

sql Copy code

```sql
CREATE TABLE Staff (
  staff_id INT PRIMARY KEY,
  email VARCHAR(100) UNIQUE  -- this becomes an alternate key
);
```

Integrity Constraints

1. Key Constraints

Every table should have a primary key that is unique and NOT NULL.

2. Domain Constraints

Column values must match data type and meaning.
Example: Phone number must be 10 digits.

3. Referential Integrity Constraints

Foreign key must reference valid data from another table.

Database Constraints – Crisp Notes

What are Constraints?

Constraints are rules applied to columns or tables to **ensure accuracy, reliability, and validity of data** in a database.
If any operation violates a constraint, the database **rejects** the action.

Types of Constraints (Covered Here)

1. NOT NULL Constraint

- Ensures a column **cannot be left empty**.
- Prevents inserting or updating a record with a `NULL` value.
- Used for important fields like `customer_id`, `customer_name`, etc.

Example

sql Copy code

```sql
CREATE TABLE Customers (
  customer_id INT NOT NULL,
  customer_name VARCHAR(50) NOT NULL
);
```

2. DEFAULT Constraint

- Automatically inserts a **default value** when no value is provided.
- Reduces repetitive data entry.
- Helpful when most data follows the same value (like city = "Barcelona").

Example

sql Copy code

```sql
CREATE TABLE Player (
  player_name VARCHAR(50) NOT NULL,
  city VARCHAR(50) DEFAULT 'Barcelona'
);
```

Key Points to Remember

- Constraints improve **data quality** and prevent invalid data.
- Constraints can be defined at **column level** or **table level**.
- Common constraints: **NOT NULL**, **DEFAULT**, **PRIMARY KEY**, **FOREIGN KEY**, etc.
- If data violates a constraint → **Operation is aborted**.

---

4. Introduction to MySQL

Installing & Connecting MySQL

- Install MySQL Server
- Install MySQL Workbench
- Connect using localhost / root user

Using MySQL in VS Code

- Install "MySQL" extension
- Configure connection using host, user, password

---

5. SQL Datatypes

Common SQL Datatypes

- **INT** → whole numbers
- **FLOAT, DOUBLE** → decimals
- **BOOLEAN** → true/false
- **DATE, DATETIME, TIMESTAMP** → date & time
- **CHAR** → fixed-length string
- **VARCHAR** → variable length string

CHAR vs VARCHAR

- **CHAR(n)** = fixed length
- **VARCHAR(n)** = flexible length
- CHAR is faster but wastes space
- VARCHAR is optimal for most cases

Text Datatypes

- TINYTEXT (<255 chars)
- TEXT (<65k chars)
- MEDIUMTEXT (16.7M chars)

---

6. CRUD Operations

CREATE

```
CREATE TABLE students(student_id VARCHAR(50), name
VARCHAR(225));
```

```
To use that table, run this USE table_name;
```

SELECT

```
SELECT * FROM students;
```

UPDATE

```
UPDATE students SET age=21 WHERE id=1;
```

DELETE

```
DELETE FROM students WHERE id=1;
```

---

7. SQL Operators

Arithmetic Operators

+, −, ×, ÷, %

Comparison Operators

=, !=, <, >, <=, >=

WHERE Clause

Filters records based on conditions.The WHERE clause in SQL is there for the purpose of filtering records and fetching only the necessary records. This can be used in SQL SELECT, UPDATE and DELETE statements.

The filtering happens based on a condition. The condition can be written using any of the following comparison or logical operators.

Comparison operators

Operator	Description
=	Checks if the values of two operands are equal or not. If yes, then condition becomes true.
!=	Checks if the values of two operands are equal or not. If values are not equal, then condition becomes true.
<>	Checks if the values of two operands are equal or not. If values are not equal, then condition becomes true.
>	Checks if the value of the left operand is greater than the value of the right operand. If yes, then condition becomes true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then condition becomes true.
>=	Checks if the value of the left operand is greater than or equal to the value of right operand. If yes, then condition becomes true.
<=	Check if the value of the left operand is less than or equal to the value of the right operand. If yes then condition becomes true.
!<	Checks if the value of the left operand is not less than the value of the right operand. If yes, then condition becomes true.

```
1  SELECT *
2
3  FROM invoices
4
5  WHERE BillingCountry = 'USA' OR BillingCountry='France';
```

Run

Reset

Logical operators

Operator	Description
ALL	Used to compare a single value to all the values in another value set.
AND	Allows for the existence of multiple conditions in an SQL statement's WHERE clause.
ANY	Used to compare a value to any applicable value in the list as per the condition.
BETWEEN	Used to search for values that are within a set of values, given the minimum value and the maximum value.
EXISTS	Used to search for the presence of a row in a specified table that meets a certain criterion.
IN	Used to compare a value to a list of literal values that have been specified.
LIKE	Used to compare a value to similar values using wildcard operators.
NOT	Reverses the meaning of the logical operator with which it is used. For example: NOT EXISTS, NOT BETWEEN, NOT IN, etc. **This is a negate operator.**
OR	Used to combine multiple conditions in an SQL statement's WHERE clause.
IS NULL	Used to compare a value with a NULL value.
UNIQUE	Searches every row of a specified table for uniqueness (no duplicates).

ORDER BY

Sort results ASC or DESC.(by default ascending).

✅ Example 1: Sort by city first, then by name

```sql
SELECT *
FROM Customers
ORDER BY city ASC, customer_name ASC;
```

Meaning:

- First sort all rows by `city`.
- If two customers are from the same city, sort them by their `customer_name`.

⭐ When do we use multi-column ORDER BY?

Use it when:

- You want **secondary sorting criteria**.
- You want results to be **organized more clearly**.
- You want **grouped + sorted** data presentation.

SELECT DISTINCT - Removes duplicate values.

---

8. Designing Database Schema

What is a Schema?

A blueprint that defines structure: tables, columns, types, constraints, relationships.

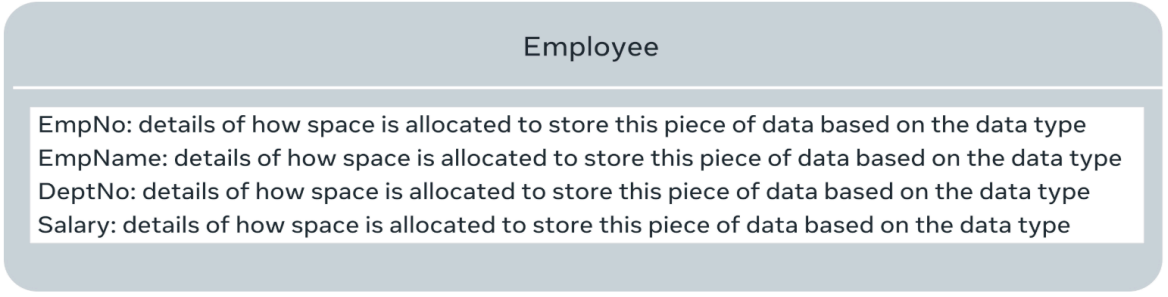
Key Concepts

- **Conceptual Schema:** The conceptual schema defines the structure of the entire database in terms of entities, attributes, and relationships, typically represented by an Entity Relationship Diagram (ER-D).
- **Database Schema:** A database schema is the structure of a database, defining how data is organized into tables with columns and rows, serving as a blueprint for data storage.
- **Three-Schema Architecture:** The three-schema architecture is a framework that outlines the three levels of database schema: conceptual, internal, and external, facilitating data abstraction and user interaction.
- **Internal Schema:** The internal schema describes the physical storage of the database, detailing how data is stored on disk in the form of tables, columns, and records.
- **External Schema:** The external schema describes how different users view the database, focusing on the specific data relevant to them while hiding nonrelevant details.

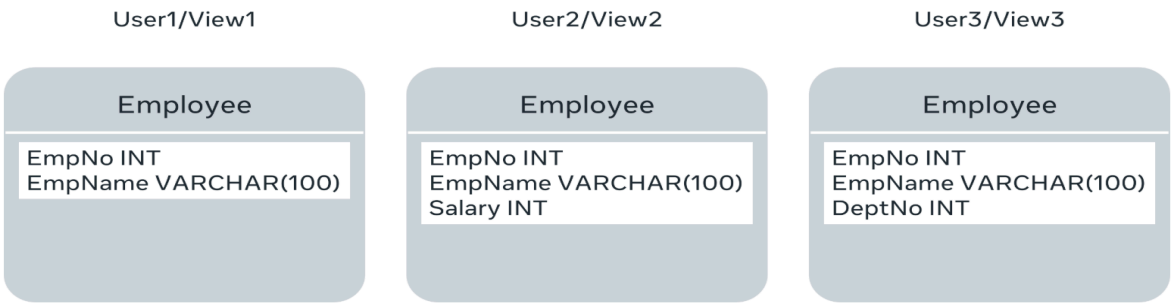
Types of Schemas

- **Conceptual or Logical schema -** The conceptual or logical schema describes the structure of the entire database for all the users. It describes the structure in terms of entities and features of the entities and the relationships between them. An Entity Relationship Diagram (ER-D) is usually drawn to represent the logical schema of a database. At this level, details about the physical storage and retrieval of data are hidden, and the database structure is described only at a concept level. The software developers work with the database at this level.

- **Internal or Physical schema -** The internal or physical schema describes the physical storage of the database. It represents the entire database but at a very low level. This means it describes how the data is really stored on disk in the form of tables, columns and records. It defines what data is stored in the database and how.



- **External or view schema** - The external or view schema describes the database like an external user would want to see it. This schema only describes the part of the database that the specific user is interested in. It hides the nonrelevant details of the database from a user.



Why Schema Design?

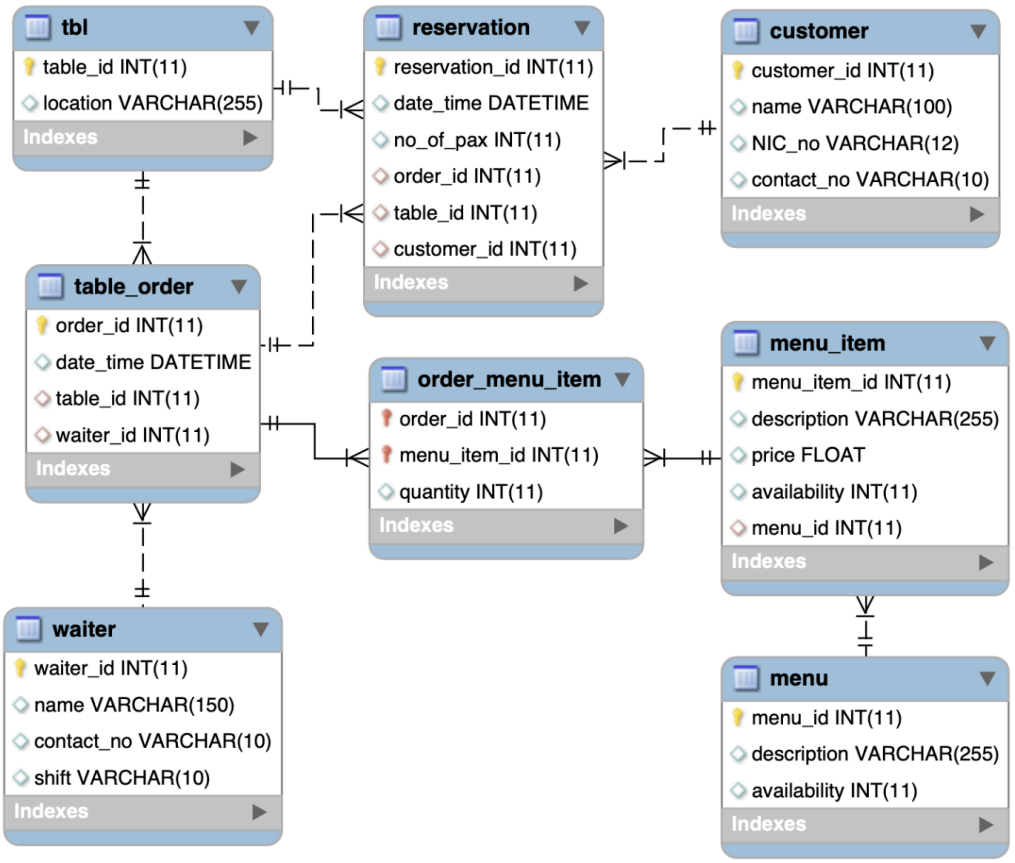
A database schema helps database engineers to organize data into well-defined tables with relevant attributes in them. It also shows the interrelationships between tables and depicts the data types that each column must have. A well-designed database schema makes life easier for database engineers as well as developers. It helps to:

- Maintain a clean set of data in the database related to an application.
- Avoid reverse-engineering of the underlying data model from time to time.
- Write efficient queries to retrieve data for reporting purposes, analytics and so on.

In other words, it prevents you from ending up with a database design that requires a database engineer to do a lot of reverse-engineering down the line, wasting time and effort that leads to increased costs for organizations.

Building a database schema for a restaurant booking scenario.

The logical database schema



The physical database schema

```sql
CREATE TABLE tbl(

    table_id INT,

    location VARCHAR(255),

    PRIMARY KEY (table_id)

);
```

```sql
CREATE TABLE waiter(

    waiter_id INT,

    name VARCHAR(150),

    contact_no VARCHAR(10),

    shift VARCHAR(10),

    PRIMARY KEY (waiter_id)

);
```

```sql
CREATE TABLE table_order(

    order_id INT,

    date_time DATETIME,

    table_id INT,

    waiter_id INT,

    PRIMARY KEY (order_id),

    FOREIGN KEY (table_id) REFERENCES tbl(table_id),

    FOREIGN KEY (waiter_id) REFERENCES waiter(waiter_id)

);
```

```sql
CREATE TABLE customer(

    customer_id INT,

    name VARCHAR(100),

    NIC_no VARCHAR(12),

    contact_no VARCHAR(10),

    PRIMARY KEY (customer_id)

);
```

```sql
CREATE TABLE reservation(

    reservation_id INT,

    date_time DATETIME,

    no_of_pax INT,

    order_id INT,

    table_id INT,

    customer_id INT,

    PRIMARY KEY (reservation_id),

    FOREIGN KEY (order_id) REFERENCES table_order(order_id),

    FOREIGN KEY (table_id) REFERENCES tbl(table_id),

    FOREIGN KEY (customer_id) REFERENCES customer(customer_id)

);
```

```sql
CREATE TABLE menu(

    menu_id INT,

    description VARCHAR(255),

    availability INT,

    PRIMARY KEY (menu_id)

);
```

```sql
CREATE TABLE menu_item(

    menu_item_id INT,

    description VARCHAR(255),

    price FLOAT,

    availability INT,

    menu_id INT,

    PRIMARY KEY (menu_item_id),

    FOREIGN KEY (menu_id) REFERENCES menu(menu_id)

);
```

```sql
CREATE TABLE order_menu_item(

    order_id INT,

    menu_item_id INT,

    quantity INT,

    PRIMARY KEY (order_id,menu_item_id),

    FOREIGN KEY (order_id) REFERENCES table_order(order_id),

    FOREIGN KEY (menu_item_id) REFERENCES menu_item(menu_item_id)

);
```

---

9. Relational Database Design

Key Concepts

- **Referential Integrity:** Referential integrity ensures that relationships between tables remain consistent, requiring that foreign keys must match primary keys in related tables.
- **Cardinality:** Cardinality refers to the number of records in a table, indicating how many instances of data are stored.
- **Primary Key:** A primary key is an attribute or set of attributes that uniquely identifies each record in a table, ensuring data integrity.
- **Record (Tuple):** A record, or tuple, is a row within a table that contains related data values for each attribute defined by the columns.
- **Relation:** A relation represents a file that stores data, also known as a table, consisting of rows (records) and columns (attributes).
- **One-to-Many Relationship:** A one-to-many relationship occurs when a record in one table can relate to multiple records in another table.

Table Relations

- One-to-One
- One-to-Many
- Many-to-Many

Finding Entities

Identify objects (student, course, order).

ER Diagrams

What is an ERD?

A visual diagram showing entities, attributes, and relationships.

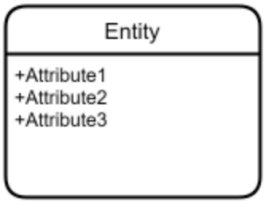
Key Concepts

- **Cardinality:** Cardinality refers to the number of instances of one entity that can or must be associated with each instance of another entity, crucial for defining relationships.
- **Attribute:** Attributes are the properties or details of an entity, defined with specific data types, that hold relevant information about the entity.
- **Entity:** An entity is represented in an ER-D as a box with compartments, where the top compartment contains the entity name and the bottom contains its attributes.
- **Relationship:** Relationships in an ER-D define how entities are related to one another, represented by lines that vary in style based on cardinality (1:1, 1:N, M:N).

10. Database Relations & Keys

Entity representation

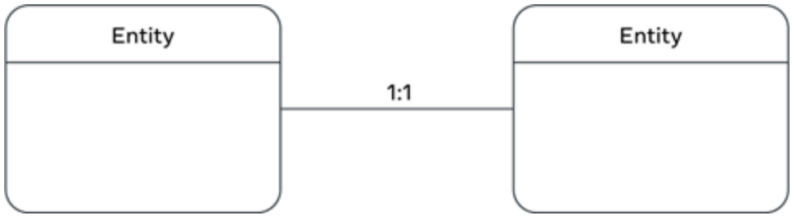
In the ER-D, a box with two compartments is used to represent the entity and its related attributes. The top compartment represents the entity name, and the bottom compartment includes the related attributes.



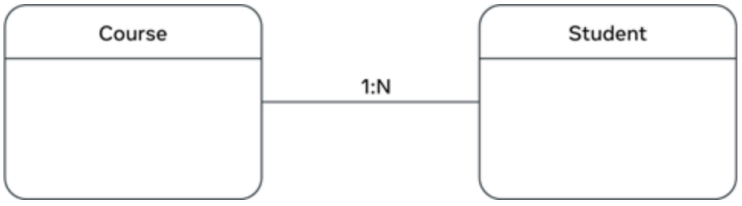
Relationship representation

The ER diagram uses different styles of lines to define the distinct types of relationships between entities. The line style depends on the cardinality of the relationship, which refers to the number of elements in a set of data as clarified in the following three cases.

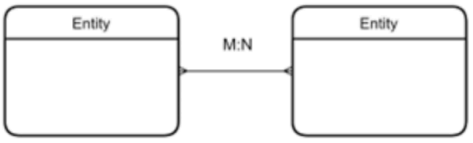
1:1 (one-to-one): The ER-D uses a straight-line representation for a one-to-one cardinality relationship. For example, each passenger on a train should have only one ticket.



1:N (one-to-many): The ER-D is a straight line with a crow's foot notation on one side only to represent a one-to-many cardinality relationship. For example, one parent can have many children.



M:N (many-to-many): The ER-D is a straight line with crow's foot notations on both sides of entities to represent a many-to-many cardinality relationship. For example, many players play many games.



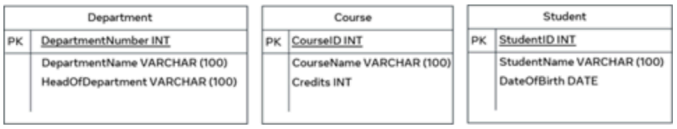
Attributes representation

Each entity has a set of attributes that hold relevant information about it. Each attribute must be defined with a data type.

In the college enrolment example, you can list the following attributes followed by relevant data types:

- The department attributes: department number, department name and head of department.
- The course attributes: course ID, course name, and course credits.
- The student attributes: student ID, name, and date of birth.

The three entities can be listed as three separate tables.



Why Use ERD?

- Planning
- Clarity
- Reduces redundancy

---

11. Database Normalization

What is Normalization?

Key Concepts

- **Normalization:** Normalization is the process of organizing data in a database to minimize redundancy and improve data integrity.
- **SQL Code:** SQL code is used to create and manipulate database tables, demonstrating the application of normalization rules.
- **Second Normal Form (2NF):** 2NF eliminates partial dependencies by ensuring that all non-key attributes are fully functionally dependent on the entire primary key.
- **Third Normal Form (3NF):** 3NF removes transitive dependencies, ensuring that non-key attributes are not dependent on other non-key attributes.
- **First Normal Form (1NF):** 1NF requires that all columns in a table contain atomic values, eliminating repeating groups of data.

Examples

Doctor ID	Doctor name	Region	Patient ID	Patient name	Surgery Number	Surgery council	Postcode	Slot ID
D1	Karl	West London	P1 P2 P3	Rami Kim Nora	3	Harrow	HA9SDE	A1 A2 A3
D1	Karl	East London	P4 P5	Kamel Sami	4	Hackney	E1 6AW	A1 A2
D2	Mark	East London	P5 P6	Sami Norma	4	Hackney	E1 6AW	A3 A4
D2	Mark	West London	P7 P1	Rose Rami	5	Harrow	HA862E	A4 A5

This unnormalized table can be written in SQL form as follows:

```sql
CREATE TABLE Surgery
    (DoctorID VARCHAR(10),
    DoctorName VARCHAR(50),
    Region VARCHAR(20),
    PatientID VARCHAR(10),
    PatientName VARCHAR(50),
    SurgeryNumber INT, Council VARCHAR(20),
    Postcode VARCHAR(10),
    SlotID VARCHAR(5),
    TotalCost Decimal);
```

First normal form

To simplify the data structure of the surgery table, let's apply the first normal form rules to enforce the data atomicity rule and eliminate unnecessary repeating data groups. The data atomicity rule means you can only have one single instance value of the column attribute in any table cell.

The atomicity problem only exists in the columns of data related to the patients. Therefore, it is important to create a new table for patient data to fix this. In other words, you can organize all data related to the patient entity in one separate table, where each column cell contains only one single instance of data, as depicted in the following example:

Patient ID	Patient name	Slot ID	Total Cost
P1	Rami	A1	1500
P2	Kim	A2	1200
P3	Nora	A3	1600
P4	Kamel	A1	2500
P5	Sami	A2	1000
P6	Norma	A5	2000
P7	Rose	A6	1000

This table includes one single instance of data in each cell, which makes it much simpler to read and understand. However, the patient table requires two columns, the patient ID and the Slot ID, to identify each record uniquely. This means that you need a composite primary key in this table. To create this table in SQL you can write the following code:

```
CREATE TABLE Patient
(PatientID VARCHAR(10) NOT NULL,
 PatientName VARCHAR(50),
 SlotID VARCHAR(10) NOT NULL,
 TotalCost Decimal,
 CONSTRAINT PK_Patient
 PRIMARY KEY (PatientID, SlotID));
```

Once you have removed the patient attributes from the main table, you just have the doctor ID, name, region, surgery number, council and postcode columns left in the table.

Doctor ID	Doctor name	Region	Surgery Number	Surgery council	Postcode
D1	Karl	West London	3	Harrow	HA9SDE
D1	Karl	East London	4	Hackney	E1 6AW
D2	Mark	West London	4	Hackney	E1 6AW
D2	Mark	East London	5	Harrow	HA862E

You may have noticed that the table also contains repeating groups of data in each column. You can fix this by separating the table into two tables of data: the doctor table and the surgery table, where each table deals with one specific entity.

Doctor table

Doctor ID	Doctor name
D1	Karl
D2	Mark

Surgery table

Surgery Number	Region	Surgery council	Postcode
3	West London	Harrow	HA9SDE
4	East London	Hackney	E1 6AW
5	West London	Harrow	HA862E

In the doctor table, you can identify the doctor ID as a single-column primary key. This table can be created in SQL by writing the following code:

In the doctor table, you can identify the doctor ID as a single-column primary key. This table can be created in SQL by writing the following code:

```
CREATE TABLE Doctor
  (DoctorID VARCHAR(10),
  DoctorName VARCHAR(50), PRIMARY KEY (DoctorID)
  );
```

Similarly, the surgery table can have the surgery number as a single-column primary key. The surgery table can be created in SQL by writing the following code:

```
CREATE TABLE Surgery
 (SurgeryNumber INT NOT NULL,
 Region VARCHAR(20), Council VARCHAR(20),
 Postcode VARCHAR(10), PRIMARY KEY (SurgeryNumber)
 );
```

By applying the atomicity rule and removing the repeating data groups, the database now meets the first normal form.

Second normal form

In the second normal form, you must avoid partial dependency relationships between data. Partial dependency refers to tables with a composite primary key. Namely, a key that consists of a combination of two or more columns, where a non-key attribute value depends only on one part of the composite key.

Since the patient table is the only one that includes a composite primary key, you only need to look at the following table.

Patient ID	**Patient name**	**Slot ID**	**Total Cost**
P1	Rami	A1	1500
P2	Kim	A2	1200
P3	Nora	A3	1600
P4	Kamel	A1	2500
P5	Sami	A2	1000
P5	Sami	A3	1000
P6	Sami	A4	1500
P7	Norma	A5	2000
P8	Rose	A6	1000
P1	Rami	A7	1500

In the patient table, you must check whether any non-key attributes depend on one part of the composite key. For example, the patient's name is a non-key attribute, and it can be determined by using the patient ID only.

Similarly, you can determine the total cost by using the Slot ID only. This is called partial dependency, which is not allowed in the second normal form. This is because all non-key attributes should be determined by using both parts of the composite key, not only one of them.

This can be fixed by splitting the patient table into two tables: patient table and appointment table. In the patient table you can keep the patient ID and the patient's name.

Patient table	
Patient ID	**Patient name**
P1	Rami
P2	Kim
P3	Nora
P4	Kamel
P5	Sami
P7	Norma
P8	Rose

The new patient table can be created in SQL using the following code:

The new patient table can be created in SQL using the following code:

```
CREATE TABLE Patient
 (PatientID VARCHAR(10) NOT NULL,
 PatientName, VARCHAR(50), PRIMARY KEY (PatientID)
 );
```

However, in the appointment table, you need to add a unique key to ensure you have a primary key that can identify each unique record in the table. Therefore, the appointment ID attribute can be added to the table with a unique value in each row.

Appointment table before adding a new unique key

Appointment table

Slot ID	Total Cost
A1	1500
A2	1200
A3	1600
A1	2500
A2	1000
A3	1000
A4	1500
A5	2000
A6	1000
A7	1500

Appointment table after adding a new unique key

Appointment table

Appointment ID	Slot ID	Total Cost
1	A1	1500
2	A2	1200
3	A3	1600
4	A1	2500
5	A2	1000
6	A3	1000
7	A4	1500
8	A5	2000
9	A6	1000
10	A7	1500

The new appointments table can be created in SQL using the following code:

```
CREATE TABLE Appointments
 (AppointmentID INT NOT NULL,
  SlotID, VARCHAR(10),
  TotalCost Decimal, PRIMARY KEY (AppointmentID)
 );
```

You have removed the partial dependency, and all tables conform to the first and second normal forms.

Third normal form

For a relation in a database to be in the third normal form, it must already be in the second normal form (2NF). In addition, it must have no transitive dependency. This means that any non-key attribute in the surgery table may not be functionally dependent on another non-key attribute in the same table. In the surgery table, the postcode and the council are non-key attributes, and the postcode depends on the council. Therefore, if you change the council value, you must also change the postcode. This is called transitive dependency, which is not allowed in the third normal form.

Surgery number	Region	Surgery council	Postcode
3	West London	Harrow	HA9SDE
4	East London	Hackney	E1 6AW
5	West London	Harrow	HA862E

In other words, changing the value of the council value in the above table has a direct impact on the postcode value, because each postcode in this example belongs to a specific council. This transitive dependency is not allowed in the third normal form. To fix it you can split this table into two tables: one for the region with the city and one for the surgery.

Location table

Surgery number	Postcode
3	HA9SDE
4	E1 6AW
5	HA862E

The new surgery location table can be created in SQL using the following code:

```
CREATE TABLE Location
 (SurgeryNumber INT NOT NULL,
  Postcode VARCHAR(10), PRIMARY KEY (SurgeryNumber)
 );
```

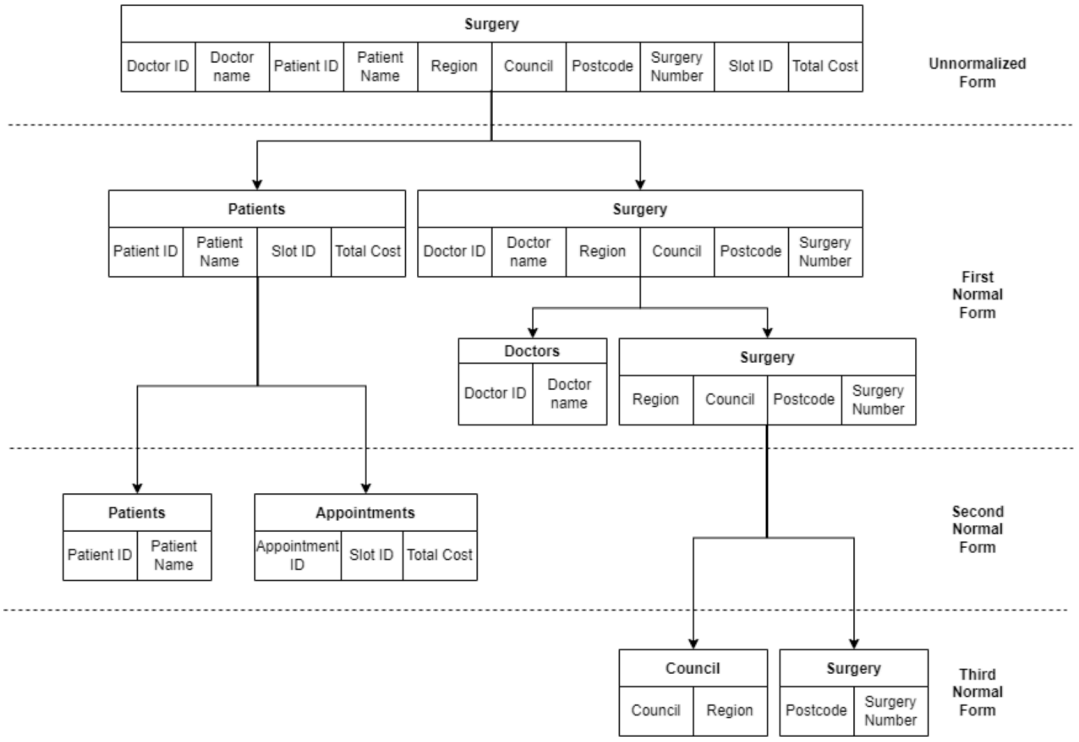
Council table

Surgery council	Region
Harrow	West London
Hackney	East London

The new surgery council table can be created in SQL using the following code:

```
CREATE TABLE Council
 (Council VARCHAR(20) NOT NULL,
  Region VARCHAR(20), PRIMARY KEY (Council)
 );
```

This ensures the database conforms to first, second, and third normal forms. The following diagram illustrates the stages through which the data moves from the unnormalized form to the first normal form, the second normal form, and finally to the third normal form.



However, it's important to link all tables together to ensure you have well-organized and related tables in the database. This can be done by defining foreign keys in the tables.