Introduction to Databases & SQL

When working withlargeamounts ofdata,databasesareessential.Databases are used to store, manage, and retrieve data efficiently.

Why Not Use Excel or Text Files to Store Data?

At a small scale, Excel sheets or text files look convenient. But as data grows, serious problems start to appear.

Limitations of Excel

- Hard to manage large datasets efficiently
- Very slow when data grows into lakhs or millions of rows
- No proper multi user access control
- High chances of accidental data modification
- Poor support for complex queries and automation

Limitations of Text Files

- No structure or enforced data types
- Searching and filtering is slow
- No relationships between data
- No security or access control
- Data consistency is impossible to guarantee

These tools are fine for temporary or personal use, not for real applications or analytics.

Why Do We Need Databases?

Adatabase is designedto store, manage, andretrieve large amounts of data efficiently and safely.

Databases solve problems that files and spreadsheets cannot.

What Databases Provide

- Fast data access even with millions of records
- Structured data with rules and constraints
- Data consistency and accuracy
- Secure access with permissions
- Multiple users working at the same time
- Easy data analysis using queries

For analytics, databases are essential because real world data is always large, messy, and constantly changing.

What Is a Database?

A**database** is an organized collection of data stored electronically so it can be easily accessed, managed, and updated.

Think of a database as: - A smart storage system - That understands your data - And allows you to ask questions about it efficiently
Example: Instead of scrolling through files, you can directly ask: - Give me total sales for last month - Find top 5 customers by revenue - Show users who signed up but never purchased

Databases make this possible in seconds.

Where SQL Fits In

SQL standsforStructured Query Language.

It is the language used to: - Store data - Retrieve data - Update data - Analyze data

SQL is the bridge between you and the database.

In this course, we will use **MySQL**, one of the most widely used SQL databases in the industry.

What You Should Remember From This Lecture

- Excel andtext files do not scale
- Real applications and analytics need databases
- A database is structured, fast, secure, and reliable
- SQL is the language used to work with databases
- MySQL will be our tool to learn SQL practically

Relational vs Non Relational Databases

Beforelearning MySQL,it isimportant to understand thetwomajortypesof databases.

Relational Databases

Relational databases store data in the form of **tables**.

Each table has: - Rows called records - Columns called fields - A fixed structure (schema)
Tables can be connected to each other using relationships.

Example

- A `users` table stores user details
- An `orders` table stores purchase details
- Both tables are linked using a common column like `user_id`

This structure makes data organized and reliable.

Key Characteristics

- Data is stored in tables
- Fixed schema
- Relationships between tables
- Strong data consistency
- SQL is used to query data

Popular Relational Databases

- MySQL

- PostgreSQL
- SQL Server
- Oracle

For **dataanalytics**, relational databases are the most commonly used.

Non Relational Databases

Non relational databases do not use tables.

They store data in flexible formats like: - Key value pairs - Documents (JSON) - Graphs - Columns
There is no fixed structure. Each record can look different.

Example

A user document may contain: - Name - Email - Preferences - Nested objects

Another user document may have completely different fields.

Key Characteristics

- Flexible schema
- No fixed table structure
- Scales very well horizontally
- Designed for large scale applications
- Querying is not standardized like SQL

Popular Non Relational Databases

- MongoDB
- Redis
- Cassandra
- Neo4j

Relational vs Non Relational (Quick Comparison)

Feature	Relational	Non Relational
Data format	Tables	JSON, key value, graph
Schema	Fixed	Flexible
Relationships	Supported	Limited or manual
Consistency	Strong	Often eventual
Query language	SQL	Database specific
Analytics friendly	Yes	Not ideal

Which One Should You Learn First?

For **dataanalytics**, reporting, and business intelligence: - Relational databases are the standard - SQL is mandatory - MySQL is a perfect starting point

Non relational databases are more useful for: - Real time apps - High scale systems - Unstructured or rapidly changing data

Installing MySQL Server and MySQL Workbench on Windows

We will be installing **MySQL Server (database engine)** first, and then **MySQL Workbench (GUI tool)** on a Windows system.

---

1. Install MySQL Server (Database Engine)

We will use the **official MySQL Installer**, which correctly installs and configures the MySQL Server.

Step 1: Download MySQL Server Installer

1. Open the official MySQL Installer page: https://dev.mysql.com/downloads/installer/
2. Download **mysql-installer-web-community** (recommended).

   The file name will look like:

   ```
   mysql-installer-web-community-8.0.xx.0.msi
   ```

3. If asked to log in, click **"No thanks, just start my download."**

Step 2: Run the Installer

1. Double-click the downloaded .msi file.
2. If Windows asks for permission, click **Yes**.

Step 3: Choose Setup Type

1. When the installeropens,select **Server only**.
2. Click **Next**.

> This installs **only MySQL Server**, without Workbench or extra tools.

Step 4: Install Required Components

1. Ifany required componentsaremissing,the installer will show them.
2. Click **Execute** to install them.
3. After completion, click **Next**.

Step 5: Install MySQL Server

1. Reviewthe selectedproducts.
2. Click **Execute**.
3. Wait until the installation completes successfully.
4. Click **Next**.

Step 6: Configure MySQL Server

6.1 Server Type

- Select **Development Computer**
- Click **Next**

6.2 Connectivity & Port

- Keep the default port **3306**
- Click **Next**

6.3 Authentication Method

- Select **Use Strong Password Encryption**

- Click **Next**

6.4 Set Root Password

- Set a password for the **root** user
- Confirm the password
- Save it securely
- Do not add extra users
- Click **Next**

6.5 Windows Service

- Service Name: **MySQL80**
- Enable **Start MySQL Server at System Startup**
- Click **Next**

6.6 Apply Configuration

- Click **Execute**
- Wait for all steps to complete
- Click **Finish**

Step 7: Finish MySQL Server Installation

1. Click**Next**
2. Click **Finish**

At this point, **MySQL Server is installed and running**.

---

2. Install MySQL Workbench (GUI Tool)

Now we install **MySQL Workbench**, which is used to visually manage databases and run SQL queries.

Step 1: Download MySQL Workbench

1. VisittheMySQLWorkbenchdownloadpage:https://dev.mysql.com/downloads/workbench/
2. Select **Windows (x86, 64-bit)**.
3. Download the installer.
4. Click **“No thanks, just start my download.”** if prompted.

Step 2: Install MySQL Workbench

1. Run thedownloaded installer.
2. Click **Next** and keep default options.
3. Click **Install**.
4. Once installation completes, click **Finish**.

Step 3: Open MySQL Workbench

1. Open **MySQL Workbench** fromtheStart Menu.
2. On the home screen, under **MySQLConnections** , click: **Local instance MySQL80**
3. Enter the **root password** you created earlier.
4. Click **OK**

The SQL Editor screen will open successfully.

You have now **successfully installed MySQL Server and MySQL Workbench on Windows**.

Writing Our First SQL Query

Inthislesson,wewillwriteourveryfirstSQLcommandsandunderstand what they do.

Creating a Database

```
CREATE DATABASE ecom;
```

This command creates a new database named `ecom`. A database is a structured place where we store our tables and data.

Using a Database

```
USE ecom;
```

This tells MySQL that all the queries we run now should apply to the `ecom` database.

If you do not use a database, MySQL will not know where to create tables or store data.

Saving and Opening SQL Scripts in MySQL Workbench

Saving a SQL Script

1. Write your queries in the SQL editor.
2. Click **File → Save Script**.

3. Save the file with a `.sql` extension Example: `first_queries.sql`

Opening a SQL Script

1. Click **File → Open SQL Script**.
2. Select your .sql file.
3. The queries will open in a new SQL editor tab.

MySQL Starter Script

Sofarwe learntthat Relationaldatabases use **SQL** to store data in tables, while non relational databases do **not use SQL** and store data in flexible formats like documents or key-value pairs.

CRUD Operations

CRUD stands for **Create, Read, Update, Delete**. These are the four basic operations we can perform on data in a database. - **Create** : Adding new data (using `INSERT`). - **Read**: Retrieving data (using `SELECT`). - **Update**: Modifying existing data (using `UPDATE`). - **Delete**: Removing data (using `DELETE`).

Resetting the Database

```
DROP DATABASE IF EXISTS ecom;
```

Deletes the database if it already exists so we can start fresh.

Creating and Using the Database

```
CREATE DATABASE ecom;
USE ecom;
```

Creates a new database named `ecom` and sets it as the active database.

Creating the `orders` Table

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY AUTO_INCREMENT,
    customer_name      VARCHAR(100),      city
    VARCHAR(50),     product     VARCHAR(100),
    category    VARCHAR(50),    quantity    INT,
    price_per_unit              DECIMAL(10,2),
    discount_percent  INT,  order_date  DATE,
    delivery_date      DATE,      payment_mode
    VARCHAR(30),   order_status   VARCHAR(30),
    rating INT




);
```

This table stores order related data like customer details, product info, dates, payments, and ratings.

Inserting Data into `orders`

```
INSERT INTO orders
(customer_name, city, product, category, quantity, price_per_unit,
discount_percent, order_date, delivery_date, payment_mode, order_status, rating)
VALUES
('Amit Sharma','Delhi','Laptop','Electronics',1,65000,10,'2025-01-05','2025
-01-08','Credit Card','Delivered',5),
('Neha Verma','Mumbai','Headphones','Electronics',2,2500,0,'2025-01-10','20
25-01-12', 'UPI', 'Delivered', 4),
('Rahul Khan','Delhi','Office Chair','Furniture',1,12000,15,'2025-01-12','2
025-01-20','Debit Card','Delivered',5),
('Priya Singh','Bangalore','Notebook','Stationery',10,80,0,'2025-01-15','20
25-01-16', 'Cash', 'Delivered', 3),
('Arjun Mehta','Ahmedabad','Smartphone','Electronics',1,30000,5,
```

```
'2025-01-18', NULL,'UPI','Cancelled', NULL),
('Sara Ali','Delhi','Table Lamp','Home Decor',2,1500,20,'2025-01-20','2025-01-23','Credit Card','Delivered',4),
('Rohit Gupta','Mumbai','Water Bottle','Kitchen',5,500,0,'2025-01-22','2025-01-24', 'Cash', 'Delivered', 2),
('Kavita Joshi','Pune','Backpack','Accessories',1,3500,10,'2025-01-25','2025-01-29','Debit Card','Delivered',5),
('Mohammed Faisal','Hyderabad','Keyboard','Electronics',1,1800,0,'2025-01-28', '2025-02-01', 'UPI', 'Delivered', 4),
('Ananya Roy','Kolkata','Study Table','Furniture',1,15000,25,'2025-02-01', NULL,'Credit Card','Pending', NULL),
('Vikram Patel','Surat','Mixer Grinder','Appliances',1,4200,5,'2025-02-03', '2025-02-06', 'UPI', 'Delivered', 4),
('Pooja Nair','Chennai','Yoga Mat','Fitness',2,1200,0,'2025-02-05','2025-02-07', 'Cash', 'Delivered', 5);
```

This adds sample data so we can practice real SQL queries.

Viewing the Data

```
SELECT * FROM orders;
```

DataTypes & Creating a Table in MySQL

Atableiswhere we storedatainsideadatabase.Eachtablehas **columns**,and every column has a **data type**.

Example Table: `customers`

```
CREATE TABLE customers (
    customer_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100), email VARCHAR(150), age
    INT, phone VARCHAR(15), is_active BOOLEAN,
    signup_date   DATE,   created_at   DATETIME,
    total_spent DECIMAL(10,2)




);
```

This table stores basic customer information.

Commonly Used MySQL Data Types

- `INT` Stores whole numbers Example: age, quantity, ids
- `VARCHAR(n)` Stores text with a fixed maximum length Example: name, email, city
- `DECIMAL(p,s)` Stores precise decimal numbers Example: price, salary, total_spent

- BOOLEAN Stores true or false values Example: is_active
- DATE Stores only date Example: signup_date
- DATETIME Stores date and time together Example: created_at
- ENUM('val1', 'val2', ...) Stores one value from a predefined list Example: status (‘active’, ‘inactive’, ‘pending’)

Removing the Table

After learning, we can delete the table.

```
DROP TABLE customers;
```

This removes the table completely from the database.

Renaming the Table

```
ALTER TABLE customers RENAME TO clients;
```

Inserting Rows in a Table

Once atableis created,we canadddatatoitusingthe `INSERT` statement. Each row represents one record in the table.

Example Table: `customers`

Assume this table already exists.

```
CREATE TABLE customers (
    customer_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100), email VARCHAR(150), age
    INT, phone VARCHAR(15), is_active BOOLEAN,
    signup_date   DATE,   created_at   DATETIME,
    total_spent DECIMAL(10,2)




);
```

Inserting a Single Row

```
INSERT INTO customers
(name, email, age, phone, is_active, signup_date, created_at, total_spent)
VALUES
('Amit Sharma','amit@gmail.com',28,'9876543210', TRUE,'2025-01-10',
'2025-01-10 10:30:00',12000.50);
```

This adds one customer record to the table.

Inserting Multiple Rows

```
INSERT INTO customers
(name, email, age, phone, is_active, signup_date, created_at, total_spent)
VALUES
('Neha Verma','neha@gmail.com',25,'9123456789', TRUE,'2025-01-12','2025-01-12
09:15:00', 5400.00),
('Rahul Khan','rahul@gmail.com',32,'9988776655', FALSE,'2025-01-15','2025-01-1
5 14:20:00',0.00);
```

This inserts multiple rows in a single query.

Viewing Inserted Data

```
SELECT * FROM customers;
```

This shows all rows stored in the table.

Selecting Data from the `orders` Table

To read data from a table, we use the `SELECT` statement.

Selecting All Columns

```
SELECT * FROM orders;
```

This returns all rows and all columns from the `orders` table.

Selecting Specific Columns

```
SELECT customer_name, product, city
FROM orders;
```

This returns only the selected columns.

Filtering Rows Using `WHERE`

```
SELECT *
FROM orders
WHERE city ='Delhi';
```

This returns orders placed from Delhi.

Using Conditions

```
SELECT customer_name, product, price_per_unit
FROM orders
WHERE price_per_unit >5000;
```

This returns expensive products.

Here is another example using not equal to:

```
SELECT customer_name, product, price_per_unit
FROM orders
WHERE price_per_unit !=5000;
```

In SQL, we use IS NULL instead of = NULL because NULL represents an unknown or missing value, not an actual value, and any comparison using operators like =, !=, <, or > with NULL always results in UNKNOWN rather than TRUE or FALSE. Since a WHERE clause only selects rows where the condition evaluates to TRUE, expressions like column = NULL or column != NULL never return any rows. IS NULL and IS NOT NULL are special SQL conditions designed specifically to check whether a column has no value, avoiding this ambiguity and correctly identifying missing data.

```
SELECT *
FROM orders
WHERE delivery_date IS NULL;
```

Using `AND` and `OR`

```
SELECT *
FROM orders
WHERE city ='Delhi' AND order_status ='Delivered';
```

This returns delivered orders from Delhi.

Sorting Data

```
SELECT customer_name, order_date, price_per_unit
FROM orders
ORDER BY order_date DESC;
```

This sorts orders by latest first.

Updating & Deleting Data in a Table

Updating Rows

Tomodify existingdatain a table, we use the `UPDATE` statement. Always use `WHERE` to avoid updating all rows by mistake.

Updating a Single Row

```
UPDATE orders
SET order_status ='Delivered'
WHERE order_id =10;
```

This updates the status of the order with `order_id = 10`.

Updating Multiple Columns

```
UPDATE orders
SET discount_percent =10,
    rating =4
WHERE customer_name ='Neha Verma';
```

This updates multiple columns for the matching row.

Updating Multiple Rows

```
UPDATE orders
SET order_status ='Cancelled'
WHERE order_status ='Pending';
```

This updates all pending orders to cancelled.

Updating Using a Condition

```
UPDATE orders
SET discount_percent =20
WHERE category ='Electronics' AND price_per_unit >30000;
```

This applies a discount to expensive electronics.

Always Check Before Updating

```
SELECT *
FROM orders
WHERE order_status ='Pending';
```

Run a `SELECT` first to confirm which rows will be updated.

Deleting Rows

Toremovedata from a table, we use the `DELETE` statement. Always use `WHERE` to avoid deleting all rows by mistake.

Deleting a Single Row

```
DELETE FROM orders
WHERE order_id =5;
```

This deletes the order with `order_id = 5`.

Deleting Multiple Rows

```
DELETE FROM orders
WHERE order_status ='Cancelled';
```

This deletes all cancelled orders.

Deleting Using a Condition

```
DELETE FROM orders
WHERE order_date <'2025-01-10';
```

This deletes old orders placed before a specific date.

Always Check Before Deleting

```
SELECT *
FROM orders
WHERE order_status ='Cancelled';
```

Run a `SELECT` first to confirm which rows will be deleted.

Deleting All Rows (Use Carefully)

```
DELETE FROM orders;
```

This removes all rows but keeps the table structure.

Altering and Dropping Tables

In SQL, we use ALTER to **change a table structure** and DROP to **remove a table completely**.

Altering a Table

`ALTER TABLE` modifies the table structure **without deleting existing data**.

Adding a New Column

```
ALTER TABLE orders
ADD COLUMN delivery_partner VARCHAR(50);
```

Adds a new column to the `orders` table. All existing rows remain unchanged.

Modifying a Column

```
ALTER TABLE orders
MODIFY price_per_unit DECIMAL(12,2);
```

Changes the data type or size of a column. Data stays as long as it fits the new definition.

Renaming a Column

```
ALTER TABLE orders
RENAME COLUMN city TO customer_city;
```

Renames a column while keeping all data.

Dropping a Column

```
ALTER TABLE orders
DROP COLUMN rating;
```

Deletes **only the column**, not the table.

Deleting Table Data vs Table Structure

Delete All Rows but Keep Structure

```
DELETE FROM orders;
```

- Removes **all data**
- Table structure stays
- Can be rolled back in transactions

Dropping a Table

Delete Everything Including Structure

```
DROP TABLE orders;
```

- Deletes**table data**
- Deletes **table structure**
- Table no longer exists
- Cannot be rolled back

Drop Table Only If It Exists

```
DROP TABLE IF EXISTS orders;
```

Prevents errors if the table does not exist.

Quick Summary

- `ALTER TABLE` Changes structure, data stays
- `DELETE FROMtable` Deletes data, structure stays
- `DROP TABLE` Deletes data and structure completely

Transactions in MySQL

AtransactionisagroupofSQLstatementsthatare executed **as one unit**. Either **all changes are saved**, or **none of them are**.

Turning Auto Commit OFF

By default, MySQL saves every change immediately (auto commit ON). Let's turn it OFF using:

```
SET autocommit =0;
```

This tells MySQL **not to save changes automatically**.

From now on, changes will be saved only when we explicitly commit them.

What Is Auto Commit

When `autocommit is` ON (default):

- Every `INSERT`, `UPDATE`, or `DELETE is` saved immediately
- You cannot undo changes

When `autocommit is` OFF:

- Changes are temporary
- You can choose to save or undo them

Example Transaction on `orders`

Step 1: Update Data

```
UPDATE orders
SET order_status ='Cancelled'
WHERE order_id =3;
```

The change is **not permanent yet**.

Step 2: Check the Change

```
SELECT order_id, order_status
FROM orders
WHERE order_id =3;
```

You will see the updated value.

Committing a Transaction

```
COMMIT;
```

- Saves all changes permanently
- Changes cannot be undone after this

Rolling Back a Transaction

```
ROLLBACK;
```

- Cancels all changes made after the last commit

- Restores the data to its previous state

Transaction Flow Summary

```
SET autocommit =0
UPDATE / INSERT / DELETE
COMMIT   → save changes
ROLLBACK → undo changes
```

Important Notes

- Transactions work only with **transactional engines** like InnoDB
- SELECT statements do not need commit or rollback
- Always use transactions when working with critical data

Constraints in MySQL

Constraints arerules appliedtotablecolumnsto **control what data can be stored**. They help maintain **data accuracy and integrity**.

Commonly Used Constraints

UNIQUE

Ensures that all values in a column are different. No duplicate values are allowed.

NOT NULL

Ensures that a column cannot store `NULL` values.

CHECK

Applies a condition that every value in the column must satisfy.

DEFAULT

Automatically assigns a value if no value is provided.

Creating Table with Constraints

Step 1: Create `employees` Table

```
CREATE TABLE employees (
    emp_id INT PRIMARY KEY AUTO_INCREMENT,
    email VARCHAR(150) UNIQUE,
```

```
    name VARCHAR(100) NOT NULL, age INT CHECK
    (age    >=18),    department    VARCHAR(50)
    DEFAULT  'General',  salary  DECIMAL(10,2)
    CHECK   (salary   >0),   joining_date   DATE
    DEFAULT (CURRENT_DATE)
);
```

This table demonstrates multiple constraints together.

UNIQUE Constraint Example

```
INSERT INTO employees (email, name, age, salary)
VALUES ('amit@company.com','Amit Sharma',25,45000);
```

Trying to insert the same email again will fail.

NOT NULL Constraint Example

```
INSERT INTO employees (email, age, salary)
VALUES ('neha@company.com',24,40000);
```

This will fail because `name` cannot be NULL.

CHECK Constraint Example

```
INSERT INTO employees (email, name, age, salary)
VALUES ('rahul@company.com','Rahul Khan',16,30000);
```

This will fail because age must be 18 or above.

DEFAULT Constraint Example

```
INSERT INTO employees (email, name, age, salary)
VALUES ('pooja@company.com','Pooja Nair',26,50000);
```

- `department` will be set to General
- `joining_date` will be set to today's date

Viewing the Data

```
SELECT * FROM employees;
```

Primary Key & AUTO_INCREMENT in MySQL

A **PRIMARY KEY** uniquely identifies each row in a table. It **cannot be NULL** and **cannot have duplicate values**.

`AUTO_INCREMENT` automatically generates the next unique number for a column.

Example Table: `employees`

```
CREATE TABLE employees (
    emp_id INT PRIMARY KEY AUTO_INCREMENT,
    email    VARCHAR(150)    UNIQUE,    name
    VARCHAR(100) NOT NULL
);
```

- `emp_id` uniquely identifies each employee
- MySQL automatically assigns values like 1, 2, 3, and so on

Inserting Data Without emp_id

```
INSERT INTO employees (email, name)
VALUES ('amit@company.com','Amit Sharma');
```

MySQL automatically assigns `emp_id = 1`.

Viewing the Data

```
SELECT * FROM employees;
```

Trying to Add Another Primary Key (This Will Fail)

```
ALTER TABLE employees
ADD PRIMARY KEY (email);
```

This query will **fail** because:

- A table can have **only one PRIMARY KEY**
- `emp_id` is already defined as the primary key

MySQL will throw an error saying the table already has a primary key.

Why Primary Key Is Important

- Ensures row uniqueness
- Makes searching faster
- Required for table relationships

ORDER BY and LIMIT in MySQL

`ORDER BY` is used to **sort data**, and `LIMIT` is used to **controlhowmany rows are returned**.

Sorting Data Using ORDER BY

Sort by Price (Low to High)

```
SELECT *
FROM orders
ORDER BY price_per_unit;
```

Sorts orders in ascending order by default.

Sort by Price (High to Low)

```
SELECT *
FROM orders
ORDER BY price_per_unit DESC;
```

Sorts orders from highest to lowest price.

Sort by Date

```
SELECT order_id, customer_name, order_date
FROM orders
ORDER BY order_date DESC;
```

Shows the latest orders first.

Sorting by Multiple Columns

```
SELECT *
FROM orders
ORDER BY city ASC, price_per_unit DESC;
```

First sorts by city, then by price within each city.

Limiting the Number of Rows

```
SELECT *
FROM orders
LIMIT 5;
```

Returns only the first 5 rows.

Using ORDER BY with LIMIT

```
SELECT customer_name, product, price_per_unit
FROM orders
```

```
ORDER BY price_per_unit DESC
LIMIT 3;
```

Returns the top 3 most expensive orders.

OFFSET with LIMIT

```
SELECT *
FROM orders
ORDER BY order_date
LIMIT 5 OFFSET 5;
```

Skips the first 5 rows and returns the next 5.

Key Points

- `ORDER BY` controls sorting
- `LIMIT` controls how many rows are shown
- Always use `ORDER BY` with `LIMIT` for predictable results

Functions in MySQL

Functions are built-inoperationsthat**performcalculations or transformations on data**. They are commonly used inside `SELECT` queries.

Aggregate Functions

These work on **multiple rows**and return a **single value**.

COUNT

```
SELECT COUNT(*)
FROM orders;
```

Returns the total number of orders.

SUM

```
SELECT SUM(quantity * price_per_unit) AS total_revenue
FROM orders;
```

Calculates total revenue.

AVG

```
SELECT AVG(price_per_unit)
FROM orders;
```

Returns the average product price.

MIN and MAX

```
SELECT MIN(price_per_unit), MAX(price_per_unit)
FROM orders;
```

Finds the cheapest and most expensive products.

Scalar Functions

These work on **each row individually**.

ROUND

```
SELECT customer_name, ROUND(price_per_unit,0)
FROM orders;
```

Rounds the price to the nearest whole number.

UPPER and LOWER

```
SELECT UPPER(customer_name), LOWER(city)
FROM orders;
```

Converts text to uppercase or lowercase.

LENGTH

```
SELECT customer_name, LENGTH(customer_name)
FROM orders;
```

Returns the number of characters in a string.

Date Functions

CURRENT_DATE

```
SELECT CURRENT_DATE;
```

Returns today’s date.

DATEDIFF

```
SELECT order_id, DATEDIFF(delivery_date, order_date) AS delivery_days
FROM orders;
```

Calculates delivery time in days.

Using Functions with WHERE

```
SELECT *
FROM orders
WHERE YEAR(order_date)=2025;
```

Filters orders placed in 2025.

Key Points

- Aggregate functions summarize data
- Scalar functions modify individual values
- Functions are often combined with `WHERE`, `ORDER BY`, and `GROUP BY`

Logical Operators in MySQL

Logical operators help us **filter data based on conditions**

Using `IN`

`IN` is used to match a value against a **list of values**.

```
SELECT *
FROM orders
WHERE city IN ('Delhi','Mumbai','Bangalore');
```

This returns orders from only the listed cities.

`NOT IN`

```
SELECT *
FROM orders
WHERE payment_mode NOT IN ('Cash','UPI');
```

Returns orders where payment was not done using Cash or UPI.

Using `BETWEEN` and `NOT BETWEEN`

`BETWEEN` checks for values within a range and includes both ends.

```sql
SELECT *
FROM orders
WHERE price_per_unit BETWEEN 1000 AND 10000;
```

Returns orders with price between 1000 and 10000.

`NOT BETWEEN`

```sql
SELECT *
FROM orders
WHERE price_per_unit NOT BETWEEN 1000 AND 10000;
```

Returns orders outside the given price range.

Using `LIKE` with Wildcards

`LIKE` is used for **pattern matching** in text.

`%` Wildcard

Matches **any number of characters**.

```sql
SELECT *
FROM orders
WHERE customer_name LIKE 'A%';
```

Returns customers whose name starts with A.

```sql
SELECT *
FROM orders
WHERE product LIKE '%Table%';
```

Returns products containing the word “Table”.

`_` Wildcard

Matches **exactlyone character**.

```sql
SELECT *
FROM orders
WHERE city LIKE 'D_lhi';
```

Matches `Delhi` but not `Dehli` or `Dilli` .

Combining Logical Conditions

```sql
SELECT * FROM orders WHERE category IN
('Electronics','Furniture') AND price_per_unit
NOT BETWEEN 5000 AND 20000;
```

Filters data using multiple logical rules.

Key Points

- `IN` replaces multiple `OR` conditions
- `NOT BETWEEN` filters values outside a range
- `%` matches many characters

- `_` matches exactly one character

Foreign Keys in MySQL

A**FOREIGNKEY** connects two tables andensures that the relationship between them is valid. It prevents orders from referencing sellers that do not exist.

Existing Table: `orders`

The orders table already exists and stores order details.

Now we want each order to be associated with a **seller**.

Step 1: Create `sellers` Table

```
CREATE TABLE sellers (
    seller_id INT PRIMARY KEY AUTO_INCREMENT,
    seller_name VARCHAR(100) UNIQUE NOT NULL,
    city VARCHAR(50)
);
```

This table stores unique seller information.

Step 2: Add Seller Column to `orders`

```
ALTER TABLE orders
ADD COLUMN seller_id INT;
```

This column will store the seller for each order.

Step 3: Create Foreign Key Relationship

```
ALTER TABLE orders
ADD CONSTRAINT fk_orders_seller
FOREIGN KEY (seller_id)
REFERENCES sellers(seller_id);
```

This ensures:

- Every `seller_id` in `orders` must exist in `sellers`
- Invalid seller references are not allowed

You can also create the foreign key while creating the `orders` table:

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY AUTO_INCREMENT,
    customer_name     VARCHAR(100),     city
    VARCHAR(50),    product    VARCHAR(100),
    category   VARCHAR(50),   quantity   INT,
    price_per_unit            DECIMAL(10,2),
    discount_percent  INT,  order_date  DATE,
    delivery_date     DATE,     payment_mode
    VARCHAR(30),  order_status  VARCHAR(30),
    rating  INT,  seller_id  INT,  CONSTRAINT
    fk_orders_seller FOREIGN KEY (seller_id)
    REFERENCES sellers(seller_id)
);
```

Example: Valid Data Insert

```
INSERT INTO sellers (seller_name, city)
VALUES ('TechWorld','Delhi');
```

```
INSERT INTO orders (seller_id, product, quantity, price_per_unit)
VALUES (1,'Laptop',1,65000);
```

This works because seller `1` exists.

Example: Invalid Insert (Will Fail)

```
INSERT INTO orders (seller_id, product, quantity, price_per_unit)
VALUES (999,'Laptop',1,65000);
```

This fails because seller `999` does not exist.

Failed Data Deletion Due to Foreign Key

```
DELETE FROM sellers
WHERE seller_id =1;
```

This fails if there are orders linked to seller `1`.

Why Foreign Keys Matter

- Enforces dataintegrity
- Prevents invalid relationships
- Makes data consistent

- Essential for relational databases

ON DELETE in MySQL

Whentablesare related using a**foreignkey**, `ON DELETE` defines **what happens to child rows** when a row in the parent table is deleted.

Here:

- `sellers` is the **parent table**
- `orders` is the **child table**
- `seller_id` connects them

Relationship Between Tables

- One seller can have many orders
- Each order belongs to one seller

Adding Foreign Key Without ON DELETE

```
ALTER TABLE orders
ADD CONSTRAINT fk_orders_seller
FOREIGN KEY (seller_id)
REFERENCES sellers(seller_id);
```

By default, MySQL uses `RESTRICT` .

ON DELETE CASCADE

Deletesallrelatedorderswhenaseller is deleted.

```
ALTER TABLE orders
DROP FOREIGN KEY fk_orders_seller;

ALTER TABLE orders
ADD CONSTRAINT fk_orders_seller
FOREIGN KEY (seller_id)
REFERENCES sellers(seller_id)
ON DELETE CASCADE;
```

What Happens

```
DELETE FROM sellers
WHERE seller_id =3;
```

- Seller with seller_id = 3 is deleted
- All orders linked to seller 3 are deleted automatically

ON DELETE SET NULL

Keepsorders but removes theseller reference.

> seller_id in orders must allow NULL for this to work.

```
ALTER TABLE orders
MODIFY seller_id INT NULL;
```

```
ALTER TABLE orders
DROP FOREIGN KEY fk_orders_seller;

ALTER TABLE orders
```

```
ADD CONSTRAINT fk_orders_seller
FOREIGN KEY (seller_id)
REFERENCES sellers(seller_id)
ON DELETE SET NULL;
```

What Happens

- Seller is deleted
- Orders remain
- seller_id in orders becomes `NULL`

ON DELETE RESTRICT (Default)

Prevents deletion if orders exist.

```
ALTER TABLE orders
DROP FOREIGN KEY fk_orders_seller;

ALTER TABLE orders
ADD CONSTRAINT fk_orders_seller
FOREIGN KEY (seller_id)
REFERENCES sellers(seller_id)
ON DELETE RESTRICT;
```

What Happens

```
DELETE FROM sellers
WHERE seller_id =5;
```

- MySQL throws an error
- Seller is not deleted

Quick Comparison

- `CASCADE` Deletes orders automatically
- `SET NULL` Keeps orders, removes seller link
- `RESTRICT` Blocks seller deletion

Key Rule

Choose `ONDELETE` based on **business logic**:

- Marketplace shutdown → `CASCADE`
- Seller leaves platform, orders kept→ `SET NULL`
- Prevent accidental deletion → `RESTRICT`

JOINs in SQL

JOINs are usedto **combinedatafrommultiple tables** based on a related column.

Here, both tables are connected using `seller_id` .

Tables Involved

- `orders` Contains order details and `seller_id`
- `sellers` Contains seller information

INNER JOIN

Returns only rows where matching data exists in **both tables**.

```
SELECT
    o.order_id,
    o.product,
    o.city AS customer_city,
    s.seller_name
FROM orders o
INNER JOIN sellers s
ON o.seller_id = s.seller_id;
```

Only orders that have a valid seller are shown.

LEFT JOIN

Returns **all orders**,even if the seller is missing.

```
SELECT
    o.order_id,
    o.product,
    s.seller_name
FROM orders o
LEFT JOIN sellers s
ON o.seller_id = s.seller_id;
```

Orders without a seller will show `NULL` for seller details.

RIGHT JOIN

Returns **allsellers**,even if they have no orders.

```
SELECT
s.seller_name,
o.order_id,
o.product
FROM orders o
RIGHT JOIN sellers s
ON o.seller_id = s.seller_id;
```

Sellers with no orders will show `NULL` for order columns.

JOIN with WHERE Condition

```
SELECT
    o.order_id,
    o.product,
```

```
    s.seller_name
FROM orders o
JOIN sellers s
ON o.seller_id = s.seller_id
WHERE o.order_status ='Delivered';
```

Filters only delivered orders.

Key Points to Remember

- `INNER JOIN` returns matching rows only
- `LEFT JOIN` keeps all left table rows
- `RIGHT JOIN` keeps all right table rows
- JOINs use **foreign key relationships**

Indexes in MySQL

Use indexes **onlywhenatable is read more than it is updated** . Indexes make `SELECT` queries faster but **slow down** **`INSERT`**, **`UPDATE`**, **and** **`DELETE`** operations.

What Is an Index

An index is a data structure that helps MySQL **find rows quickly** without scanning the entire table.

Think of it like an index page in a book.

Creating an Index

Index on a Frequently Searched Column

```
CREATE INDEX idx_orders_city
ON orders (city);
```

This speeds up queries that filter by city.

Using the Index

```
SELECT *
FROM orders
WHERE city ='Delhi';
```

MySQL can use the index to locate rows faster.

Composite Index

An index can be created on multiple columns.

```
CREATE INDEX idx_orders_city_status
ON orders (city, order_status);
```

Useful when queries filter by both city and order status.

When NOT to Use Indexes

- Columns with very few unique values
- Tables with frequent updates
- Small tables

Indexes are not free, they trade **writeperformance** for **read speed**.

Removing an Index

```
DROP INDEX idx_orders_city ON orders;
```

Deletes the index but keeps the table data.

Key Points

- Indexes speed up `SELECT`
- Indexes slow down `INSERT` ,`UPDATE`, `DELETE`

- Create indexes only on frequently searched columns

Views in SQL

A **view** is a **saved SQL query** that behaves like a virtual table. It does not store data, it only stores the query.

Views are used to **simplify queries, hide complexity, and control data access**.

Creating a View

```
CREATE VIEW delivered_orders AS
SELECT order_id,
       customer_name,
       city,
       product,
       price_per_unit,
       order_date
FROM orders
WHERE order_status ='Delivered';
```

This view shows only delivered orders.

Using a View

```
SELECT *
FROM delivered_orders;
```

You can query a view just like a table.

Why Use Views

- Reuse complex queries
- Improve readability
- Restrict access to sensitive columns
- Keep business logic in one place

Updating Data Through a View

```
UPDATE delivered_orders
SET price_per_unit = price_per_unit +500
WHERE order_id =1;
```

If the view is simple, updates affect the original table.

Dropping a View

```
DROP VIEW delivered_orders;
```

Deletes the view, not the table.

Key Difference

- View: Virtual,storesquery only
- Table: Physical, stores actual data

Subqueries in SQL

A**subquery** isa query written **insideanother query** . It is used when one query depends on the result of another.

Subquery in `WHERE`

```
SELECT *
FROM orders
WHERE price_per_unit >(
    SELECT AVG(price_per_unit)
    FROM orders
);
```

This returns orders that are priced above the average price.

Subquery with `IN`

```
SELECT *
FROM orders
WHERE city IN (
SELECT city
FROM orders
WHERE category ='Electronics'
);
```

This returns orders from cities where electronics were sold.

Subquery in `SELECT`

```
SELECT order_id,
       customer_name,
       price_per_unit,
       (SELECT AVG(price_per_unit) FROM orders) AS avg_price
FROM orders;
```

Adds the average price as a column in every row.

Subquery with `EXISTS`

```
SELECT *
FROM orders o
WHERE EXISTS (
    SELECT 1
    FROM orders
    WHERE city = o.city
      AND category ='Furniture'
);
```

Returns orders from cities where furniture orders exist.

Key Points

- Subqueries run first
- Can be used in `WHERE`, `SELECT`, and `FROM`
- Useful when queries depend on other query results

GROUP BY in SQL

`GROUP BY` is usedto **grouprowsthathave the same values** and perform calculations on each group.

It is commonly used with **aggregate functions**.

GROUP BY with COUNT

```
SELECT city, COUNT(*) AS total_orders
FROM orders
GROUP BY city;
```

Returns the number of orders from each city.

GROUP BY with SUM

```
SELECT category,
       SUM(quantity * price_per_unit) AS total_sales
FROM orders
GROUP BY category;
```

Calculates total sales for each category.

GROUP BY with AVG

```
SELECT city, AVG(price_per_unit) AS avg_price
FROM orders
GROUP BY city;
```

Returns the average price per city.

GROUP BY with Multiple Columns

```
SELECT city, order_status, COUNT(*) AS count
FROM orders
GROUP BY city, order_status;
```

Groups data by both city and order status.

GROUP BY Rules

- Every selected columnmust be either
  - in `GROUP BY`, or
  - inside an aggregate function
- `GROUP BY` reduces multiple rows into one row per group

GROUP BY with ORDER BY

```
SELECT  category,  COUNT(*)  AS  total_orders
FROM orders
```

```
GROUP BY category
ORDER BY total_orders DESC;
```

Sorts groups based on aggregated values.

HAVING Clause

`HAVING` is used to filter groups after aggregation.

```
SELECT city, COUNT(*) AS total_orders
FROM orders
GROUP BY city
HAVING COUNT(*)>5;
```

Returns cities with more than 5 orders.

UNION in SQL

`UNION` is usedto **combineresults of multiple SELECT queries**into a single result set.

Rules to remember:

- Same number of columns
- Same order of columns
- Compatible data types

Existing Table: `orders`

We already have customer orders in the `orders` table.

Now we will create another table that stores **orders placed by employees**.

Creating `employee_orders` Table

```
CREATE TABLE employee_orders (
    order_id INT,
    name VARCHAR(100),
    city VARCHAR(50),
    product VARCHAR(100),
    price_per_unit DECIMAL(10,2)
);
```

This table has a similar structure so it can be combined with `orders`.

Inserting Data into `employee_orders`

```
INSERT INTO employee_orders
VALUES
(201,'Ravi Kumar','Delhi','Laptop',62000),
(202,'Sneha Patel','Mumbai','Headphones',2300),
(203,'Ankit Verma','Pune','Office Chair',11000),
(204,'Ananya Roy','Kolkata','Study Table',15000);
```

Using UNION

```
SELECT order_id, customer_name AS name, city, product, price_per_unit
FROM orders

UNION

SELECT order_id, name, city, product, price_per_unit
FROM employee_orders;
```

This combines:

- Customer orders
- Employee orders into one result set.

Duplicate rows are removed automatically.

Using UNION ALL

```
SELECT order_id, customer_name AS name, city, product, price_per_unit
FROM orders

UNION ALL
```

```
SELECT order_id, name, city, product, price_per_unit
FROM employee_orders;
```

This keeps **all rows**, including duplicates.

Key Difference

- `UNION` Removes duplicates
- `UNION ALL` Keeps duplicates and is faster

When to Use UNION

- Combining similar data from multiple tables
- Creating combined reports
- Merging historical or temporary data

ROLLUP in SQL

`ROLLUP` is used with `GROUP BY` to generate **summary rows** like subtotals and a grand total.

It is mainly used in **reporting and analytics**.

Basic ROLLUP Example

```
SELECT city,
       category,
       SUM(quantity * price_per_unit) AS total_sales
FROM orders
GROUP BY city, category WITH ROLLUP;
```

This query produces:

- Sales per city and category
- Subtotal for each city
- One grand total row at the end

How to Read ROLLUP Results

- `(city, category)` Normal grouped data
- `(city, NULL)` Total sales for that city
- `(NULL, NULL)` Grand total of all sales

ROLLUP with Single Column

```
SELECT category,
       COUNT(*) AS total_orders
FROM orders
GROUP BY category WITH ROLLUP;
```

This gives:

- Order count per category
- Overall total orders

Identifying Total Rows

```
SELECT
    city,
    category,
    SUM(quantity * price_per_unit) AS total_sales
FROM orders
GROUP BY city, category WITH ROLLUP;
```

Rows containing `NULL` values represent **summary totals**, not missing data.

When to Use ROLLUP

- Businessreports
- Sales summaries
- Dashboard totals
- Financial aggregation

Key Points

- `ROLLUP` adds subtotals automatically
- Order of columns in `GROUP BY` matters
- Supported natively in MySQL

Stored Procedures in MySQL

A **stored procedure** is a **saved block of SQL statements** that runs as a single unit. It helps reuse logic, reduce repetition, and keep business logic inside the database.

Why We Change the Delimiter

By default, MySQL uses `;` to end a statement.

A stored procedure contains **multiple SQL statements**, each ending with `;`. So MySQL gets confused about **where the procedure ends**.

To fix this, we temporarily change the delimiter.

Changing the Delimiter

```
DELIMITER //
```

Now MySQL will treat `//` as the end of the procedure.

Creating a Stored Procedure

Example: Get Delivered Orders

```
CREATE PROCEDURE get_delivered_orders()
BEGIN
    SELECT *
    FROM orders
```

```
    WHERE order_status ='Delivered';
END //
```

The procedure ends with `//`, not `;`.

Resetting the Delimiter

```
DELIMITER ;
```

Always reset the delimiter after creating the procedure.

Calling a Stored Procedure

```
CALL get_delivered_orders();
```

This executes the stored procedure.

Stored Procedure with Parameter

```
DELIMITER //

CREATE PROCEDURE get_orders_by_city(IN city_name VARCHAR(50))
BEGIN
    SELECT *
    FROM orders
    WHERE city = city_name;
END //

DELIMITER ;
```

Calling the Procedure with Parameter

```
CALL get_orders_by_city('Delhi');
```

Why Use Stored Procedures

- Reuse SQL logic
- Improve maintainability
- Reduce network traffic
- Centralize business rules

Key Points

- Stored procedures are reusable SQL blocks
- Delimiter is changed to avoid confusion
- Always reset the delimiter after use

Triggers in MySQL

A **trigger** is a pieceof SQLthat **automatically runs when a specific event happens** on a table.

Triggers are commonly used for:

- Logging changes
- Enforcing rules
- Maintaining audit history

Practical Scenario

Whenever an order is **cancelled**, we want to **store a record** of that cancellation.

This should happen automatically, without writing extra queries.

Step 1: Create a Log Table

```
CREATE TABLE order_cancellations (
    log_id INT PRIMARY KEY AUTO_INCREMENT,
    order_id INT,
    cancelled_on DATETIME,
    reason VARCHAR(100)
);
```

This table stores cancellation history.

Step 2: Change the Delimiter

```
DELIMITER //
```

We change the delimiter because a trigger contains multiple SQL statements.

Step 3: Create the Trigger

```
CREATE TRIGGER trg_log_order_cancel
AFTER UPDATE ON orders
FOR EACH ROW
BEGIN
    IF NEW.order_status ='Cancelled'
       AND OLD.order_status <>'Cancelled' THEN

        INSERT INTO order_cancellations
        (order_id, cancelled_on, reason)
        VALUES
        (NEW.order_id, NOW(),'Order cancelled by user');

    END IF;
END //
```

Step 4: Reset the Delimiter

```
DELIMITER ;
```

How the Trigger Works

- Runs **after an update** on `orders`

- Checks if the order status changed to `Cancelled`
- Automatically inserts a log record

Testing the Trigger

```
UPDATE orders
SET order_status ='Cancelled'
WHERE order_id =2;
```

Now check the log table:

```
SELECT * FROM order_cancellations;
```

A cancellation record is automatically created.

Other Commonly Used Triggers

- `BEFORE INSERT`
- `AFTER INSERT`
- `BEFORE UPDATE`
- `BEFORE DELETE`
- `AFTER DELETE`

Why Triggers Are Powerful

- No manual logging needed
- Logic runs automatically
- Prevents missed business rules
- Keeps database consistent

Important Notes

- Triggers run automatically
- Cannot be called manually
- Use carefully, they add hidden logic

MySQL Workbench GUI Tutorial

MySQLWorkbench allows you to manage and modifydatabase data **without writing SQL**, using its graphical interface. This is especially useful for beginners, quick fixes, and real-world admin tasks.

Editing Data from SELECT Query Results

How it works

1. Right-click a table
2. Click **Select Rows – Limit 1000**
3. Edit any cell directly in the result grid

Apply to Save Changes

- After editing a value, click **Apply**
- MySQL Workbench shows the generated `UPDATE` query
- Click **Apply** again to commit changes

Revert to Undo Changes

- If you made a mistake **before applying**, click **Revert**
- This restores the original values from the database

Note: Edits work reliably only if the table has a **Primary Key**.

Form Editor (Row-by-Row Editing)

The **Form Editor** letsyoueditonerowatatimeinastructured layout.

How to open

- Click the **Form Editor icon** in the result grid toolbar

When to use

- Tables with many columns
- Long text fields
- Avoiding accidental edits

Safer than inline grid editing for beginners

Export Table Data as CSV

Steps

1. Right-click the table
2. Choose **TableDataExport Wizard**
3. Select **CSV**
4. Choose file location
5. Click **Next→ Finish**

Use cases:

- Data sharing
- Backups
- Excel analysis

Import Data from CSV (GUI)

Steps

1. Right-click the table
2. Select **TableDataImport Wizard**
3. Choose the CSV file
4. Map columns carefully
5. Review preview
6. Click **Import**

Common mistakes:

- Wrong date format
- Column mismatch
- Header row not handled correctly

Key Takeaway

> MySQL WorkbenchGUI is powerful for **learning, debugging,andadmin work**, but understanding what happens **behind the scenes** is still important.

Using AI to Generate SQL Queries

AItoolscan help you**writeSQLfaster**,**understandqueries**,and **explore data** more efficiently. They are assistants, not replacements for understanding SQL.

How AI Helps with SQL

AI can:

- Convert plain English to SQL
- Generate SELECT, JOIN, `GROUP BY`, and subqueries
- Explain complex queries
- Help debug errors

Example: English to SQL

Prompt to AI:

> Show all delivered orders from Delhi sorted by latest order date

Generated SQL:

```
SELECT *
FROM orders
WHERE city ='Delhi'
AND order_status ='Delivered'
ORDER BY order_date DESC;
```

Example: Analytics Query

Prompt to AI:

> Find total sales per category

```
SELECT category,
       SUM(quantity * price_per_unit) AS total_sales
FROM orders
GROUP BY category;
```

Example: Fixing a Query

Prompt to AI:

> This query gives an error, fix it

AI can explain:

- Missing `GROUP BY`
- Wrong column names
- Logical mistakes

Best Way to Use AI

- UseAI to**speed upwriting**
- Always **read and understand** the query
- Verify results with `SELECT`
- Never blindly trust generated SQL in production

What AI Cannot Replace

- Understanding tablestructure
- Knowing business logic
- Knowing when a query is correct
- Database design decisions

Key Takeaway

AIisa **SQLcopilot**,nota shortcut to skip learning. The better your SQL basics, the better AI helps you.

Capstone Project - Database Design for an E-commerce Store

HarryShop is a merchandisestoreforcoderssellinghoodies,mugs,stickers, notebooks, and accessories.

This project demonstrates:

- Database design
- Table relationships
- Foreign keys
- Realistic demo data
- Business-style SQL analysis

---

1. Create Database

```
CREATE DATABASE harryshop;
USE harryshop;
```

---

2. Database Schema

Customers Table

```
CREATE TABLE customers (
    customer_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100),
    email VARCHAR(150) UNIQUE,
    city VARCHAR(50),
```

```
    signup_date DATE
);
```

Products Table

```
CREATE TABLE products (
    product_id INT PRIMARY KEY AUTO_INCREMENT,
    product_name VARCHAR(100),
    category VARCHAR(50),
    price DECIMAL(10,2),
    stock INT
);
```

Orders Table

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY AUTO_INCREMENT, customer_id INT,
    order_date DATE, order_status VARCHAR(30), FOREIGN KEY
    (customer_id) REFERENCES customers(customer_id)


);
```

Order Items Table

```
CREATE TABLE order_items (
    order_item_id INT PRIMARY KEY AUTO_INCREMENT,
    order_id INT,
    product_id INT,
    quantity INT,
    FOREIGN KEY (order_id) REFERENCES orders(order_id),
```

```
    FOREIGN KEY (product_id) REFERENCES products(product_id)
);
```

Payments Table

```
CREATE TABLE payments (
    payment_id INT PRIMARY KEY AUTO_INCREMENT,
    order_id INT,
    payment_mode VARCHAR(30),
    amount DECIMAL(10,2),
    payment_date DATE,
    FOREIGN KEY (order_id) REFERENCES orders(order_id)
);
```

---

3. Insert Demo Data

Customers (20 rows)

```
INSERT INTO customers (name, email, city, signup_date) VALUES
('Amit  Sharma','amit@gmail.com','Delhi','2025-01-01'),  ('Neha
Verma','neha@gmail.com','Mumbai','2025-01-02'),          ('Rahul
Khan','rahul@gmail.com','Bangalore','2025-01-03'),       ('Pooja
Nair','pooja@gmail.com','Chennai','2025-01-04'),         ('Rohit
Gupta','rohit@gmail.com','Delhi','2025-01-05'),         ('Ananya
Roy','ananya@gmail.com','Kolkata','2025-01-06'),         ('Karan
Mehta','karan@gmail.com','Ahmedabad','2025-01-07'),     ('Simran
Kaur','simran@gmail.com','Chandigarh','2025-01-08'),     ('Mohit
Jain','mohit@gmail.com','Jaipur','2025-01-09'),          ('Sneha
Patel','sneha@gmail.com','Surat','2025-01-10'),         ('Vikram
Singh','vikram@gmail.com','Lucknow','2025-01-11'),        ('Alok
Mishra','alok@gmail.com','Patna','2025-01-12'),          ('Nidhi
Agarwal','nidhi@gmail.com','Noida','2025-01-13'),      ('Saurabh
Verma','saurabh@gmail.com','Ghaziabad','2025-01-14'),     ('Riya
Sen','riya@gmail.com','Pune','2025-01-15'),
```

```
('Aditya Malhotra','aditya@gmail.com','Delhi','2025-01-16'),
('Kritika Shah','kritika@gmail.com','Mumbai','2025-01-17'),
('Yash Tiwari','yash@gmail.com','Kanpur','2025-01-18'),
('Mehul Joshi','mehul@gmail.com','Vadodara','2025-01-19'),
('Isha Kapoor','isha@gmail.com','Gurgaon','2025-01-20');
```

Products (20 rows)

```
INSERT INTO products (product_name, category, price, stock) VALUES
('Python Hoodie','Clothing',1999,50),
('Java Hoodie','Clothing',1899,40),
('Debugging Mug','Accessories',599,100),
('Code Like a Pro Mug','Accessories',649,80),
('DSA Notebook','Stationery',499,150),
('SQL Cheat Sheet','Stationery',299,200),
('Sticker Pack','Stationery',249,300),
('Algorithm T-Shirt','Clothing',1499,60),
('GitHub Cap','Accessories',799,70),
('Keyboard Mat','Accessories',999,90),
('Linux Hoodie','Clothing',2099,35),
('AI Nerd T-Shirt','Clothing',1599,55),
('Whiteboard Notebook','Stationery',699,120),
('Bug Hunter Mug','Accessories',549,100),
('Terminal Stickers','Stationery',199,250),
('Coder Bottle','Accessories',899,110),
('Late Night Hoodie','Clothing',2199,30),
('Python Socks','Accessories',399,140),
('DSA Flash Cards','Stationery',349,180),
('Clean Code Notebook','Stationery',599,130);
```

Orders (20 rows)

```
INSERT INTO orders (customer_id, order_date, order_status) VALUES
(1,'2025-02-01','Delivered'), (2,'2025-02-02','Delivered'),
```

```
(3,'2025-02-03','Delivered'),
(4,'2025-02-04','Cancelled'),
(5,'2025-02-05','Delivered'),
(6,'2025-02-06','Pending'),
(7,'2025-02-07','Delivered'),
(8,'2025-02-08','Delivered'),
(9,'2025-02-09','Delivered'),
(10,'2025-02-10','Cancelled'),
(11,'2025-02-11','Delivered'),
(12,'2025-02-12','Delivered'),
(13,'2025-02-13','Pending'),
(14,'2025-02-14','Delivered'),
(15,'2025-02-15','Delivered'),
(16,'2025-02-16','Delivered'),
(17,'2025-02-17','Cancelled'),
(18,'2025-02-18','Delivered'),
(19,'2025-02-19','Delivered'),
(20,'2025-02-20','Delivered');
```

Order Items (20+ rows)

```
INSERT INTO order_items (order_id, product_id, quantity) VALUES
(1,1,1),(1,7,2), (2,3,1),(2,5,1), (3,8,1), (4,2,1), (5,6,3),
(6,4,1), (7,10,1), (8,12,2), (9,15,3), (10,9,1), (11,11,1),
(12,13,1), (13,14,2), (14,16,1), (15,17,1),
```

```
(16,18,2),
(17,19,1),
(18,20,1),
(19,1,1),
(20,5,2);
```

Payments (20 rows)

```
INSERT INTO payments (order_id, payment_mode, amount, payment_date) VALUES
(1,'UPI',2497,'2025-02-01'),
(2,'Credit Card',1098,'2025-02-02'),
(3,'UPI',1499,'2025-02-03'),
(5,'Debit Card',897,'2025-02-05'),
(7,'UPI',999,'2025-02-07'),
(8,'Credit Card',3198,'2025-02-08'),
(9,'UPI',747,'2025-02-09'),
(11,'UPI',2099,'2025-02-11'),
(12,'Debit Card',699,'2025-02-12'),
(14,'UPI',899,'2025-02-14'),
(15,'Credit Card',2199,'2025-02-15'),
(16,'UPI',798,'2025-02-16'),
(18,'Debit Card',599,'2025-02-18'),
(19,'UPI',1999,'2025-02-19'),
(20,'Credit Card',998,'2025-02-20');
```

---

4. Analysis Queries

Total Revenue

```
SELECT SUM(amount) AS total_revenue
FROM payments;
```

Revenue by Product

```
SELECT p.product_name,

       SUM(oi.quantity * p.price) AS revenue
FROM order_items oi
JOIN products p ON oi.product_id = p.product_id
JOIN orders o ON oi.order_id = o.order_id
WHERE o.order_status ='Delivered'
GROUP BY p.product_name
ORDER BY revenue DESC;
```

Top Customers by Spend

```
SELECT c.name,
       SUM(p.amount) AS total_spent
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
JOIN payments p ON o.order_id = p.order_id
GROUP BY c.name
ORDER BY total_spent DESC;
```

Best Selling Products

```
SELECT p.product_name,
       SUM(oi.quantity) AS total_sold
FROM order_items oi
JOIN products p ON oi.product_id = p.product_id
GROUP BY p.product_name
ORDER BY total_sold DESC;
```

Cancelled Orders Count

```
SELECT COUNT(*) AS cancelled_orders
FROM orders
WHERE order_status ='Cancelled';
```