RDBMS

Database

What it is: a structured collection of related data (e.g., customers, orders, products).

Analogy: a library (books = records, shelves/categories = tables).

Key ideas:

- **Schema** = the structure (tables, columns, relationships).
- **Instance** = the actual data stored at some point in time.
- **Logical vs physical:** logical is how data is modeled (tables, columns); physical is how it's stored on disk.

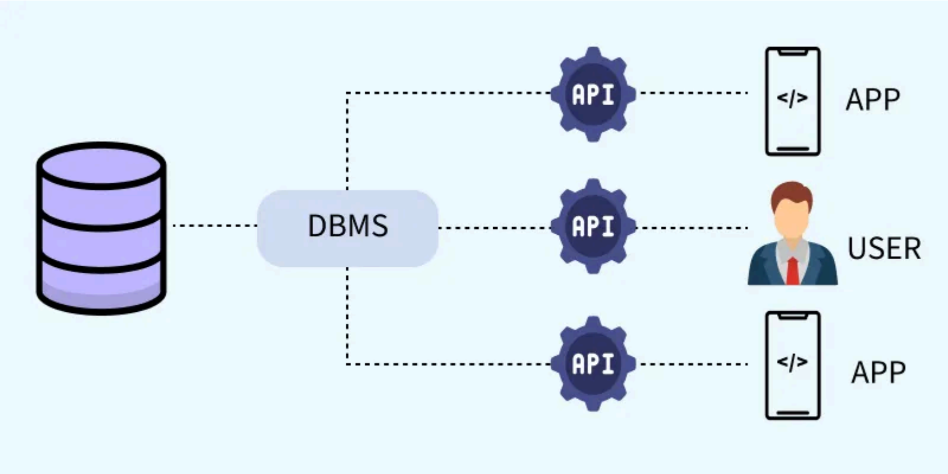
DBMS (Database Management System)

What it is: the software that stores, retrieves and manages the database. Examples: MySQL, PostgreSQL, Oracle, SQL Server, MongoDB (NoSQL).

Core responsibilities:

- Data storage & retrieval
- Query processing and optimization
- Transaction management (ACID)
- Concurrency control (multiple users)
- Backup & recovery
- Access control/security
- Enforcing integrity constraints (primary keys, foreign keys, uniqueness)

Why not just files? DBMS provides indexing, transactions, integrity rules, multi-user access, and query languages — things plain files don't.



Data Models

What it is: a formal way to structure data and relationships. It defines how data is organized and how relationships are represented.

Major types:

1. **Relational model (tables, rows, columns)** — rows = records, columns = attributes. (Most common for business apps.)
2. **Document model** — JSON-like documents (e.g., MongoDB).
3. **Key–Value** — very simple store (map from key → value), great for caching.
4. **Column-family (wide-column)** — stores columns grouped by row key (e.g., Cassandra).
5. **Graph** — nodes & edges for highly connected data (e.g., Neo4j).
6. **Hierarchical / Network** — older models where records have tree/graph structures.

Levels of modeling:

- **Conceptual model** (ER diagrams — entities, relationships)
- **Logical model** (tables/columns for relational DBs)
- **Physical model** (indexes, partitions, file layout)

RDBMS (Relational Database Management System)

What it is: a DBMS that implements the **relational model** (data stored as relations/tables). Examples: PostgreSQL, MySQL, Oracle, SQL Server.

Core relational concepts:

- **Table (relation):** set of rows (tuples) with columns (attributes).
- **Primary key:** unique identifier for a row.
- **Foreign key:** a column that references primary key in another table (enforces referential integrity).
- **Normalization:** organizing tables to reduce redundancy and anomalies (1NF, 2NF, 3NF, BCNF).
- **Transactions & ACID:** Atomicity, Consistency, Isolation, Durability — guarantees for reliable updates.

When RDBMS shines: structured data, strong consistency, complex queries & joins, integrity constraints.

SQL (Structured Query Language)

What it is: the standard declarative language used to define and manipulate relational data. SQL is *declarative* — you state *what* you want, not how to get it.

Main categories:

- **DDL (Data Definition Language):** `CREATE`, `ALTER`, `DROP` — defines schema.
- **DML (Data Manipulation Language):** `SELECT`, `INSERT`, `UPDATE`, `DELETE` — read/write data.
- **DCL (Data Control Language):** `GRANT`, `REVOKE` — permissions.
- **TCL (Transaction Control):** `BEGIN` / `START TRANSACTION`, `COMMIT`, `ROLLBACK`.

🗄️ DDL (Data Definition Language)

Purpose: Defines or changes the *structure* of the database objects (schemas, tables, indexes, etc.).

CREATE – make a new object

Create a Database

```
CREATE DATABASE school_db;
```

Create a Table

```
CREATE TABLE students (
  student_id INT PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
```

```
  age INT,
  email VARCHAR(100) UNIQUE
);
```

Create an Index (helps speed up search on columns)

```
CREATE INDEX idx_student_name ON students(name);
```

ALTER (Modify Existing Objects)

1. **Add a Column**

```
ALTER TABLE courses ADD COLUMN instructor VARCHAR(100);
```

👉 Adds a new column called `instructor` to the `courses` table.

---

2. **Modify/Change a Column**

- Change datatype or size:

```
ALTER TABLE courses MODIFY credits DECIMAL(3,1);
```

👉 Changes `credits` from `INT` to allow values like `3.5`.

- Rename a column (syntax depends on RDBMS):

```
ALTER TABLE courses RENAME COLUMN instructor TO teacher;
```

---

3. **Drop (Remove) a Column**

```
ALTER TABLE courses DROP COLUMN teacher;
```

👉 Removes the `teacher` column permanently.

---

4. **Add Constraints**

```
ALTER TABLE courses ADD CONSTRAINT chk_credits CHECK (credits > 0);
```

👉 Ensures `credits` must always be greater than 0.

---

5. **Drop Constraints**

```
ALTER TABLE courses DROP CONSTRAINT chk_credits;
```

---

6. **Rename the Table**

```
ALTER TABLE courses RENAME TO subjects;
```

---

7. **Add Primary or Foreign Key**

- Add Primary Key (if not already defined):

```
ALTER TABLE courses ADD PRIMARY KEY (course_id);
```

- Add Foreign Key (link to another table):

```
ALTER TABLE courses
ADD CONSTRAINT fk_instructor
FOREIGN KEY (instructor_id)
REFERENCES instructors(instructor_id);
```

📌 **Key Idea:** `ALTER` = *evolve the design* without dropping and recreating the table.

DROP – delete the object completely

Drop a Table

```
DROP TABLE students;
```

Drop a Database

```
DROP DATABASE school_db;
```

Drop an Index

```
DROP INDEX idx_student_name;
```

TRUNCATE – remove all rows, keep structure

```
TRUNCATE TABLE students;
```

(Table is empty, but still exists to insert new data later.)

📝 DML (Data Manipulation Language)

Purpose: Work with the *data inside tables*, not the structure.

1. **SELECT** – Query data

```
-- Get all student records
SELECT * FROM students;

-- Get only student names and ages
SELECT name, age FROM students;

-- Find students aged above 20
SELECT * FROM students WHERE age > 20;

-- Sort students by age
SELECT * FROM students ORDER BY age DESC;
```

2. **INSERT** – Add new records

```
-- Insert one record
INSERT INTO students (student_id, name, age, email)
VALUES (1, 'Alice Johnson', 21, 'alice@email.com');

-- Insert multiple records
INSERT INTO students (student_id, name, age, email)
VALUES
```

```
(2, 'Bob Smith', 19, 'bob@email.com'),
(3, 'Charlie Brown', 22, 'charlie@email.com');
```

3. **UPDATE** – Change existing records

```
-- Update age of a student
UPDATE students
SET age = 20
WHERE student_id = 2;

-- Update multiple fields
UPDATE students
SET name = 'Charlie B', email = 'charlieb@email.com'
WHERE student_id = 3;
```

⚠️ Without `WHERE`, it will update **all rows** in the table.

4. **DELETE** – Remove records

```
-- Delete one student
DELETE FROM students WHERE student_id = 1;

-- Delete all rows (table structure remains)
DELETE FROM students;
```

📌 **Simple Rule**:

- DML = *work with rows* (add, read, change, remove).
- DDL = *work with structure* (create, alter, drop).

🔑 DCL (Data Control Language)

1. **GRANT** – give permissions

Used to allow a user certain privileges on database objects.

```
-- Give user1 permission to SELECT data from students table
GRANT SELECT ON students TO user1;

-- Give user2 permission to INSERT and UPDATE records in students
GRANT INSERT, UPDATE ON students TO user2;

-- Give all privileges (not recommended for production!)
GRANT ALL PRIVILEGES ON students TO admin_user;
```

2. **REVOKE** – remove permissions

Used to take away previously granted privileges.

```
-- Remove SELECT privilege from user1
REVOKE SELECT ON students FROM user1;

-- Remove all privileges from user2
REVOKE ALL PRIVILEGES ON students FROM user2;
```

🛠️ DCL Example Workflow in MySQL

1. Login as `root` (superuser)

```
mysql -u root -p
```

---

2. Create a new user (`jane_doe`)

```
CREATE USER 'jane_doe'@'localhost' IDENTIFIED BY 'StrongP@ssw0rd';
```

👉 Creates a new MySQL user `jane_doe` who can log in **only from localhost**.

---

3. Grant privileges

```
GRANT SELECT, INSERT ON test.* TO 'jane_doe'@'localhost';
FLUSH PRIVILEGES;
```

👉 This allows `jane_doe` to **only read (`SELECT`) and add (`INSERT`) data** in all tables of the `test` database.

👉 `FLUSH PRIVILEGES;` makes the changes take effect immediately.

---

4. Test as `jane_doe` user

Login as `jane_doe` :

```
mysql -u jane_doe -p
```

Now try different actions:

```
-- This should work (has SELECT privilege)
SELECT * FROM students;

-- This should work (has INSERT privilege)
INSERT INTO students VALUES (6, 'New Student', 20, 'new@email.com');

-- This should FAIL (no UPDATE privilege)
UPDATE students SET age = 25 WHERE student_id = 6;
```

👉 You should see an **"Access denied; you need (at least one of) the UPDATE privilege(s)"** error when trying `UPDATE` . ✅

---

5. Check users

```
SELECT User, Host FROM mysql.user;
```

👉 Shows all MySQL users and where they can connect from.

---

6. Revoke privileges

```
REVOKE SELECT, INSERT ON test.* FROM 'jane_doe'@'localhost';
FLUSH PRIVILEGES;
```

👉 Now `jane_doe` has **no access** to the `test` database.

If `jane_doe` tries to `SELECT` or `INSERT` again, they'll get an error.

🔄 TCL (Transaction Control Language)

1. **COMMIT** – Save changes permanently

```
START TRANSACTION;

INSERT INTO students (student_id, name, age, email)
VALUES (7, 'Frank Adams', 20, 'frank@email.com');

UPDATE students SET age = 21 WHERE student_id = 7;

COMMIT;  -- ✅ Changes are now permanent
```

👉 Both the `INSERT` and `UPDATE` become permanent once `COMMIT` is executed.

2. **ROLLBACK** – Undo changes since last COMMIT

```
START TRANSACTION;

INSERT INTO students (student_id, name, age, email)
VALUES (8, 'Grace Lee', 22, 'grace@email.com');

-- Oops! Wrong data inserted
ROLLBACK;  -- ❌ Cancels the insert
```

👉 The row for Grace will **not** be saved, because we rolled back.

3. **SAVEPOINT** – Create a rollback checkpoint

```
START TRANSACTION;

INSERT INTO students VALUES (9, 'Harry Potter', 20, 'harry@email.com');
SAVEPOINT sp1;  -- 📌 checkpoint created

INSERT INTO students VALUES (10, 'Ivy White', 21, 'ivy@email.com');
SAVEPOINT sp2;

-- Rollback only to sp1 (Ivy’s row will be undone, Harry’s stays)
ROLLBACK TO sp1;

COMMIT;  -- ✅ Harry’s row is permanent, Ivy’s is gone
```

🔷 Step 1: Table creation and initial data

```
CREATE TABLE Accounts (
  AccountID INT PRIMARY KEY,
  AccountHolder VARCHAR(100),
  Balance DECIMAL(10,2)
);

INSERT INTO Accounts (AccountID, AccountHolder, Balance)
VALUES (1, 'Alice', 5000.00);
```

👉 At this point:

```
+-----------+---------------+---------+
| AccountID | AccountHolder | Balance |
+-----------+---------------+---------+
| 1         | Alice         | 5000.00 |
+-----------+---------------+---------+
```

🔷 Step 2: Start a transaction and deduct balance

```
START TRANSACTION;

UPDATE Accounts SET Balance = Balance - 1000 WHERE AccountID = 1;

SELECT * FROM Accounts;
```

👉 After this update **inside the transaction (before COMMIT)**:

```
| 1 | Alice | 4000.00 |
```

🔷 Step 3: COMMIT

```
COMMIT;
```

👉 Now the balance reduction is **permanent**. Even if you disconnect and reconnect, Alice's balance will remain `4000.00`.

🔷 Step 4: UPDATE + ROLLBACK

```
UPDATE Accounts SET Balance = Balance - 1000 WHERE AccountID = 1;

SELECT * FROM Accounts;
```

👉 After this update (but before rollback):

```
| 1 | Alice | 3000.00 |
```

Now rollback:

```
ROLLBACK;

SELECT * FROM Accounts;
```

👉 After rollback, the update is undone, and balance goes back to:

```
| 1 | Alice | 4000.00 |
```

🔷 Example: Using `SAVEPOINT` with the `Accounts` Table

Step 1: Starting point

Suppose your table has this data after the first COMMIT:

```
+-----------+---------------+---------+
| AccountID | AccountHolder | Balance |
+-----------+---------------+---------+
```

```
| 1        | Alice         | 4000.00 |
+-----------+---------------+---------+
```

Step 2: Begin transaction

```
START TRANSACTION;
```

Step 3: First deduction (1000)

```
UPDATE Accounts SET Balance = Balance - 1000 WHERE AccountID = 1;

-- Balance now = 3000.00 (inside transaction, not committed yet)
SAVEPOINT sp1;   -- 📌 Savepoint after deducting 1000
```

Step 4: Second deduction (500)

```
UPDATE Accounts SET Balance = Balance - 500 WHERE AccountID = 1;

-- Balance now = 2500.00
SAVEPOINT sp2;   -- 📌 Savepoint after deducting additional 500
```

Step 5: Rollback to a savepoint

```
ROLLBACK TO sp1;
```

👉 Effect: The update after `sp1` (deduction of 500) is undone, but the first deduction (1000) **remains**.

```
Balance = 3000.00
```

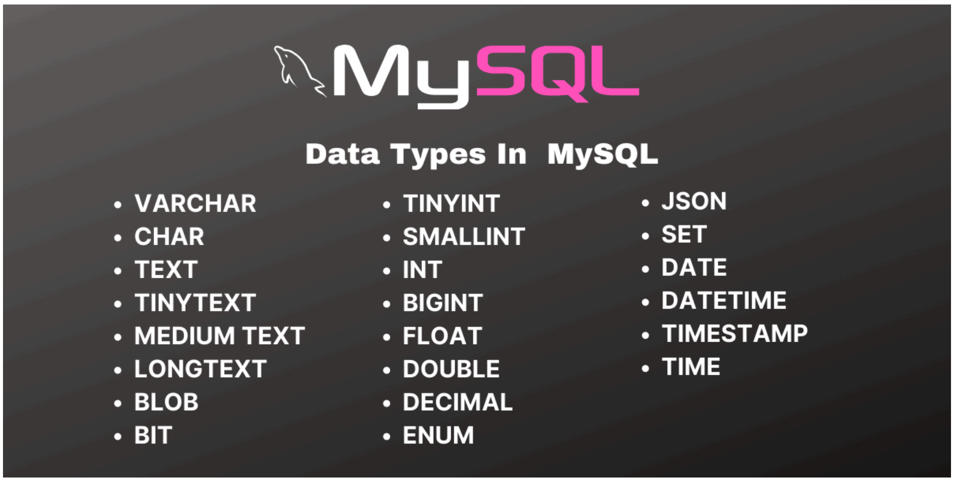
Step 6: Commit

```
COMMIT;
```

👉 Final permanent balance:

```
| 1 | Alice | 3000.00 |
```

📌 SQL Data Types


MySQL
Data Types In MySQL
• VARCHAR
• CHAR
• TEXT
• TINYTEXT
• MEDIUM TEXT
• LONGTEXT
• BLOB
• BIT
• TINYINT
• SMALLINT
• INT
• BIGINT
• FLOAT
• DOUBLE
• DECIMAL
• ENUM
• JSON
• SET
• DATE
• DATETIME
• TIMESTAMP
• TIME

MySQL Data Types

Numeric Types

Integer Types (Exact Value)

Standard SQL integer types includes INT, and SMALLINT. Additional integer types include TINYINT, MEDIUMINT, and BIGINT.

We can optionally specify the maximum display width within the parenthesis. The display width has no impact on the minimum and maximum values. It's deprecated for integer types since MySQL 8.0.17 and it will be removed in future releases. Ex: INT(10)

Type	Size	Min Signed	Max Signed	Min Unsigned	Max Unsigned
TINYINT	1 Byte	-128	127	0	255
SMALLINT	2 Bytes	-32768	32767	0	65535
MEDIUMINT	3 Bytes	-8388608	8388607	0	16777215
INTEGER or INT	4 Bytes	-2147483648	2147483647	0	4294967295
BIGINT	8 Bytes	-2^{63}	$2^{63}-1$	0	$2^{64}-1$

Fixed-Point Types (Exact Value)

Standard SQL types includes DECIMAL, and NUMERIC.
These data types store the exact numeric value preserving the exact precision.

NUMERIC is implemented as DECIMAL, hence both follow the same rules.

DECIMAL – Maximum number of digits is 65 including precision(M), and scale(D) and the maximum number of supported decimals or scale(D) is 30.

DECIMAL(M, D)
DECIMAL(M) – DECIMAL(M) or DECIMAL(M, 0)
DECIMAL – The precision(M) will be 10 by default

Floating-Point Types (Approximate Value)

Supports SQL standard FLOAT or FLOAT(p) where p is precision.
Supports nonstandard FLOAT(M,D). It's deprecated since MySQL 8.0.17 and it will be removed in future releases.

FLOAT and **DOUBLE** data types represent approximate numeric data values. MySQL supports 4 bytes for single-precision and 8 bytes for double-precision values.

FLOAT(p) – MySQL uses the optional precision value to only determine the actual storage size. The precision value from 0 to 23 results in 4-byte single-precision column, and value from 24 to 53 results in 8-byte double-precision column.

Bit-Value Type

The **BIT** data type is being used to store binary values. For example - b'111' and b'10000000' represent 7 and 128, respectively.

BIT(M) – Stores M-bit values where M can range from 1 to 64.

Date and Time Types

DATE, DATETIME, and TIMESTAMP Types

MySQL converts DATESTAMP value from current timezone to UTC before storing and convert back to current timezone from UTC while retrieving. The current timezone for each connection will be system timezone unless specified while creating the connection. The same timezone must be used while storing and retrieving to avoid changes.

DATE – YYYY-MM-DD – Used for date values without time part.
Min Value 1000-01-01
Max Value 9999-12-31

DATETIME – YYYY-MM-DD hh:mm:ss – Used for date values with time part. The limits with and without fractional seconds part are as listed below.

Min Value 1000-01-01 00:00:00
1000-01-01 00:00:00.000000
Max Value 9999-12-31 23:59:59
9999-12-31 23:59:59.999999

TIMESTAMP – YYYY-MM-DD hh:mm:ss – Used for date values with time part. The limits with and without fractional seconds part are as listed below.

Min Value 1970-01-01 00:00:01 UTC
1970-01-01 00:00:01.000000 UTC
Max Value 2038-01-19 03:14:07 UTC
2038-01-19 03:14:07.999999 UTC

Time Types

TIME – hh:mm:ss OR hhh:mm:ss – Used for time values. The large hour values can be used for elapsed time or larger time difference. The limits with and without fractional seconds part are as listed below.

Min Value -838:59:59
-838:59:59.000000
Max Value 838:59:59
838:59:59.999999

Notes: Invalid TIME values will be converted to 00:00:00

Year Types

YEAR – 1-byte type used to represent the year values.

4 Digit Min Value 1901
4 Digit Max Value 2155

1 or 2 Digit Min Value 1
1 or 2 Digit Max Value 99

Notes: MySQL converts values in the ranges from 1 to 69 and 70 to 99 to YEAR values in the ranges from 2001 to 2069 and 1970 to 1999.

String Types

CHAR and VARCHAR Types

CHAR and VARCHAR data types can be used to store string values.

CHAR – Length can be any value from 0 to 255 characters. The length of CHAR column will be fixed to the length specified while declaring the column.

VARCHAR – Length can be any value from 0 to 65,535 characters subject to a maximum row size of 65,535 characters shared among all the columns. Values are stored as a 1-byte or 2-byte length prefix plus data.

Comparison – Storing the string i.e. 'ab' in CHAR(4) will consume 4 bytes whereas VARCHAR(4) will consume only 3 bytes. The column length of CHAR remains same irrespective of actual value.

BINARY and VARBINARY Types

BINARY and VARBINARY data types can be used to store binary string values.

BINARY – Length can be any value from 0 to 255 bytes. The length of the BINARY column will be fixed to the length specified while declaring the column.

VARBINARY – Length can be any value from 0 to 65,535 bytes.

BLOB and TEXT Types

TINYBLOB, BLOB, MEDIUMBLOB, and LONGBLOB are the BLOB data types available in MySQL and used to store the variable amount of data. The BLOB values are treated as binary or byte strings.

TINYTEXT, TEXT, MEDIUMTEXT, and LONGTEXT are the TEXT data types and used to store character strings.

The maximum length of these data types are as listed below:

TINYBLOB, TINYTEXT	255 bytes
BLOB, TEXT	65,535 bytes ~64kb
MEDIUMBLOB, MEDIUMTEXT	16,777,215 bytes ~16Mb
LONGBLOB, LONGTEXT	4,294,967,295 bytes ~4GB

ENUM Type

ENUM represents string object having a value from the possible values specified in the column specification while creating the table or table column.

The ENUM data type can be used to store data in a compact form where the actual value will be stored as encoded in number form. The corresponding string value will be returned while retrieving the value.

SET Type

SET can be used to store comma separated values from a definite set specified while creating the table or table column.

For example - A column specified as SET('Apple', 'Ball', 'Cat') can have below listed values.

''
'Apple'
'Ball,Apple'

Spatial Data Types

Single geometry values

GEOMETRY – Stores geometry values of any type.

POINT – Stores point values of a single point.

LINESTRING – Collection of points to form line having multiple points.

POLYGON – Collection of points to form a polygonal surface.

Collections of values

MULTIPOINT – Stores collection of multiple points.

MULTILINESTRING – Stores collection of multiple lines.

MULTIPOLYGON – Stores collection of multiple polygons.

GEOMETRYCOLLECTION – Stores collection of geometry values of any type.

JSON Data Type

The native JSON data type supported by MySQL can be used to directly store the JSON object. MySQL ensures that a valid JSON document gets stored in the JSON column.

TECHNOLUSH
www.technolush.com

1. **Numeric Data Types**

Used to store numbers.

- **INT / INTEGER** → Whole numbers.

```
age INT;
-- Example: 25, -100, 0
```

- **SMALLINT / BIGINT** → Smaller or larger ranges of integers.
- **DECIMAL(p, s) / NUMERIC(p, s)** → Fixed-point numbers.
  - `p` = total digits, `s` = digits after decimal.

```
salary DECIMAL(10,2);
-- Example: 12345.67
```

- **FLOAT / DOUBLE** → Approximate floating-point numbers (useful for scientific values).

```
temperature FLOAT;
-- Example: 36.6
```

---

2. **Character (String) Data Types**

Used to store text.

- **CHAR(n)** → Fixed-length string (padded with spaces).

```
gender CHAR(1);
-- Example: 'M', 'F'
```

- **VARCHAR(n)** → Variable-length string (up to n characters).

```
name VARCHAR(100);
-- Example: 'Alice Johnson'
```

- **TEXT** → Large variable-length text.

```
description TEXT;
-- Example: long essay or article
```

---

3. **Date and Time Data Types**

Used to store temporal values.

- **DATE** → Stores date (YYYY-MM-DD).

```
dob DATE;
-- Example: '2002-05-15'
```

- **TIME** → Stores time (HH:MM:SS).

```
login_time TIME;
-- Example: '14:30:59'
```

- **DATETIME** → Stores both date and time.

```
created_at DATETIME;
-- Example: '2025-09-12 14:30:59'
```

- **TIMESTAMP** → Like DATETIME, but also tracks timezone/automatic updates.

---

4. Boolean Type

- **BOOLEAN / BOOL** → Stores `TRUE` or `FALSE` (internally as `1` or `0` in MySQL).

```
is_active BOOLEAN;
-- Example: TRUE / FALSE
```

✅ Example: Using Different Data Types in a Table

```
CREATE TABLE Students (
  StudentID INT PRIMARY KEY,
  Name VARCHAR(50),
  Gender CHAR(1),
  DOB DATE,
  GPA DECIMAL(3,2),
  Email VARCHAR(100),
  IsActive BOOLEAN
);
```

📌 This allows:

- `StudentID` → 101
- `Name` → 'John Doe'
- `Gender` → 'M'
- `DOB` → '2002-03-15'
- `GPA` → 3.75
- `Email` → 'john.doe@email.com'
- `IsActive` → TRUE

📌 SQL Operators

1. Arithmetic Operators

Used for mathematical calculations.

Operator	Description	Example
`+`	Addition	`SELECT 10 + 5;` → 15
`-`	Subtraction	`SELECT 10 - 5;` → 5
`*`	Multiplication	`SELECT 10 * 5;` → 50
`/`	Division	`SELECT 10 / 2;` → 5
`%`	Modulus (remainder)	`SELECT 10 % 3;` → 1

👉 Example with table:

```
SELECT name, age + 1 AS age_next_year
FROM students;
```

This shows each student's age next year.

2. **Comparison Operators**

Used to compare two values (result is TRUE/FALSE).

Operator	Description	Example
`=`	Equal to	`age = 20`
`!=` or `<>`	Not equal to	`age <> 20`
`>`	Greater than	`age > 18`
`<`	Less than	`age < 25`
`>=`	Greater than or equal to	`age >= 18`
`<=`	Less than or equal to	`age <= 30`

👉 Example:

```
SELECT * FROM students WHERE age >= 20;
```

3. **Logical Operators**

Used to combine multiple conditions.

Operator	Description	Example
`AND`	True if both conditions are true	`age > 18 AND age < 25`
`OR`	True if at least one condition is true	`age = 18 OR age = 20`
`NOT`	Negates a condition	`NOT age = 18`

👉 Example:

```
SELECT * FROM students
WHERE age > 18 AND name LIKE 'A%';
```

→ Students older than 18 whose name starts with "A".

4. **Special Operators**

Other important operators:

- **BETWEEN** → Check if value is within a range

  ```
  SELECT * FROM students WHERE age BETWEEN 18 AND 22;
  ```

- **IN** → Match any value in a list

  ```
  SELECT * FROM students WHERE age IN (18, 20, 22);
  ```

- **LIKE** → Pattern matching with wildcards
  - `%` → any number of characters
  - `_` → exactly one character

  ```
  SELECT * FROM students WHERE name LIKE 'J%';   -- names starting with J
  SELECT * FROM students WHERE name LIKE '_a%'; -- names with 'a' as 2nd letter
  ```

- **IS NULL / IS NOT NULL** → Check for null values

  ```
  SELECT * FROM students WHERE email IS NULL;
  ```

5. **Concatenation Operator** (string joining)

- In MySQL → use `CONCAT()` function

```
SELECT CONCAT(name, ' - ', email) AS student_info FROM students;
```

Employees Table

```
CREATE TABLE employees (
employee_id   INT PRIMARY KEY,
first_name    VARCHAR(50),
last_name     VARCHAR(50),
email         VARCHAR(100) UNIQUE,
phone_number  VARCHAR(20),
hire_date     DATE,
job_id        VARCHAR(10),
salary        DECIMAL(10, 2),
manager_id    INT,
department    VARCHAR(50)
);

INSERT INTO employees VALUES
(101, 'Amit', 'Sharma', 'amit.sharma@example.com', '9876543210', '2022-01-15', 'DEV01', 65000.00, NULL, 'Engine
ering'),
(102, 'Priya', 'Rao', 'priya.rao@example.com', '9876543211', '2022-03-10', 'DEV02', 62000.00, 101, 'Engineering'),
(103, 'John', 'Doe', 'john.doe@example.com', '9876543212', '2021-11-05', 'HR01', 48000.00, NULL, 'Human Resourc
es'),
(104, 'Meena', 'Kumari', 'meena.kumari@example.com', '9876543213', '2023-02-20', 'MK01', 52000.00, 103, 'Mark
eting'),
(105, 'Raj', 'Singh', 'raj.singh@example.com', '9876543214', '2020-07-18', 'FIN01', 70000.00, NULL, 'Finance'),
(106, 'Sara', 'Ali', 'sara.ali@example.com', '9876543215', '2021-09-25', 'DEV03', 60000.00, 101, 'Engineering'),
(107, 'Kiran', 'Patel', 'kiran.patel@example.com', '9876543216', '2022-12-01', 'MK02', 51000.00, 104, 'Marketing'),
(108, 'David', 'Lee', 'david.lee@example.com', '9876543217', '2023-04-12', 'HR02', 47000.00, 103, 'Human Resourc
es'),
(109, 'Anita', 'Desai', 'anita.desai@example.com', '9876543218', '2021-06-30', 'FIN02', 68000.00, 105, 'Finance'),
(110, 'Vikram', 'Joshi', 'vikram.joshi@example.com', '9876543219', '2022-08-08', 'DEV04', 63000.00
```

📌 SQL Constraints (Summary with Examples)

Constraints are rules applied to table columns to maintain **data integrity**.

1. **PRIMARY KEY**

- Ensures each row is **unique** and **not null**.
- A table can have only **one primary key** (can be composite with multiple columns).

✅ Example (Valid insert):

```
CREATE TABLE rides (
   ride_id INT PRIMARY KEY,
   driver_id INT NOT NULL,
   rider_id INT NOT NULL,
   fare DECIMAL(10,2) NOT NULL
);

INSERT INTO rides VALUES (1, 101, 201, 500.00); -- Success
```

❌ Failure (Duplicate key):

```
INSERT INTO rides VALUES (1, 102, 202, 800.00);
-- Error: Duplicate entry for PRIMARY KEY ride_id=1
```

2. UNIQUE

- Ensures all values in a column are different.
- Allows multiple `NULL` s.

✅ Example:

```
CREATE TABLE users (
    user_id INT PRIMARY KEY,
    email VARCHAR(100) UNIQUE
);

INSERT INTO users VALUES (1, 'user1@example.com'); -- Success
INSERT INTO users VALUES (2, NULL);              -- Success
INSERT INTO users VALUES (3, 'user1@example.com'); -- ❌ Fails (duplicate email)
```

3. NOT NULL

- Ensures a column **cannot store NULL** values.

✅ Example:

```
CREATE TABLE drivers (
    driver_id INT PRIMARY KEY,
    driver_name VARCHAR(100) NOT NULL
);

INSERT INTO drivers VALUES (101, 'John Doe'); -- Success
INSERT INTO drivers VALUES (102, NULL);      -- ❌ Fails
```

4. CHECK

- Restricts values based on a condition.

✅ Example:

```
CREATE TABLE rides (
    ride_id INT PRIMARY KEY,
    fare DECIMAL(10,2) CHECK (fare > 0)
);

INSERT INTO rides VALUES (201, 500.00); -- Success
INSERT INTO rides VALUES (202, -100.00); -- ❌ Fails (fare must be > 0)
```

5. FOREIGN KEY

- Creates a relationship between two tables.
- Ensures that values in one table must exist in another.

✅ Example (One-to-Many):

```
CREATE TABLE drivers (
    driver_id INT PRIMARY KEY,
    driver_name VARCHAR(100)
);

CREATE TABLE rides (
    ride_id INT PRIMARY KEY,
    driver_id INT,
    fare DECIMAL(10,2),
    FOREIGN KEY (driver_id) REFERENCES drivers(driver_id)
);

INSERT INTO drivers VALUES (101, 'John');
INSERT INTO rides VALUES (301, 101, 600.00); -- Success
```

❌ Failure (Invalid reference):

```
INSERT INTO rides VALUES (302, 999, 700.00);
-- Error: driver_id=999 not found in drivers
```

🔷 **ON DELETE CASCADE** → If parent row is deleted, child rows are also deleted.

```
CREATE TABLE rides (
    ride_id INT PRIMARY KEY,
    driver_id INT,
    fare DECIMAL(10,2),
    FOREIGN KEY (driver_id) REFERENCES drivers(driver_id) ON DELETE CASCADE
);

DELETE FROM drivers WHERE driver_id=101;
-- Automatically deletes all rides of driver 101
```

---

6. DEFAULT

- Provides a default value if none is supplied.

✅ Example:

```
CREATE TABLE customers (
    customer_id INT PRIMARY KEY,
    customer_name VARCHAR(100),
    country VARCHAR(50) DEFAULT 'India'
);

INSERT INTO customers (customer_id, customer_name) VALUES (1, 'Alice');
-- country = 'India'

INSERT INTO customers VALUES (2, 'Bob', 'USA');
-- country = 'USA'
```

Result:

```
+-------------+---------------+---------+
| customer_id | customer_name | country |
+-------------+---------------+---------+
```

```
| 1        | Alice       | India  |
| 2        | Bob         | USA    |
```

📌 Minimality in Candidate Keys

🔷 What is "Minimal"?

- A **candidate key** must be:
  1. **Unique** → It should uniquely identify each row.
  2. **Minimal** → No extra/unnecessary column should be present.

👉 If you can remove a column and the set still uniquely identifies rows, then it's **not minimal**.

🔷 Example 1: Simple Case

Table: **Students**

RollNo	Email	Name
1	alice@mail.com	Alice
2	bob@mail.com	Bob
3	charlie@mail.com	Charlie

- Candidate Keys:
  - `(RollNo)` → unique, minimal ✅
  - `(Email)` → unique, minimal ✅
- `(RollNo, Email)` → unique, but **not minimal** ❌ (since RollNo or Email alone is enough).

🔷 Example 2: Composite Key Case

Table: **Enrollments**

StudentID	CourseID	Semester
101	CSE101	1
101	CSE102	1
102	CSE101	1

- `(StudentID, CourseID)` → unique, minimal ✅ (can't drop either attribute).
- `(StudentID, CourseID, Semester)` → unique but **not minimal** ❌ (Semester is extra).

🔷 Formal One-Liner

A **candidate key** is a **minimal superkey** → meaning it is just sufficient to uniquely identify rows, and no subset of it can do the same.

📌 Natural Key vs Surrogate Key

Aspect	Natural Key	Surrogate Key
Meaning	Real-world, meaningful data	Artificial, system-generated identifier
Purpose	Uniquely identifies a record based on existing data	Uniquely identifies a record without business meaning
Examples	SSN, email address, product serial number	Auto-increment ID, UUID
Stability	May change if real-world data changes	Stable, does not change once assigned
Usage	Can directly relate to business processes	Often used internally by the database to simplify joins and indexing

---

🔷 Example Table: Customers

```
CREATE TABLE customer (
    customer_key INT AUTO_INCREMENT PRIMARY KEY,   -- Surrogate Key
    aadhar_number INT NOT NULL,                    -- Natural Key
    customer_name VARCHAR(100),
    city VARCHAR(100),
    effective_date DATE,                         -- Start date of record version
    end_date DATE,                               -- End date of record version
    is_current BOOLEAN                            -- Flag for current active record
);
```

Explanation:

1. **customer_key** → Surrogate Key
    - Auto-generated integer
    - No real-world meaning
    - Used internally by database for joins and indexing
2. **aadhar_number** → Natural Key
    - Unique identifier from the real world
    - Has business meaning

🔷 Why Use Both?

- **Natural Key**: Keeps business integrity and ensures uniqueness based on real data.
- **Surrogate Key**: Simplifies database operations (e.g., joins, versioning, historical tracking).

✅ This approach is common in **data warehouses**, **slowly changing dimensions (SCDs)**, and tables with **versioned records**.

📌 Keys in SQL: Explanation & Example

1️⃣ Primary Key

- **Definition:** A chosen candidate key that uniquely identifies each row and **cannot be NULL**.
- **Example in your table:**

    ```
    student_id INT PRIMARY KEY
    ```

    - Minimal and uniquely identifies each row.

2️⃣ Candidate Key

- **Definition:** A **minimal set of column(s)** that can uniquely identify a row.
- **Example:**
    - `email` → unique, minimal
    - `phone` → unique, minimal
- **Note:** Only **minimal keys** are candidate keys.

3️⃣ Super Key

- **Definition:** Any set of columns that can uniquely identify a row.
- **Example:**

- `(student_id, phone)` → still uniquely identifies rows.
- **Note:** Not necessarily minimal.
  - `(student_id)` is minimal → candidate key
  - `(student_id, phone)` → super key but **not a candidate key**

4️⃣ Composite Key

- **Definition:** A key that consists of **two or more columns**.
- **Example:**

  ```
  PRIMARY KEY (student_id, course_id)
  ```

  - Combines multiple columns to form a key.
- **Note:**
  - If a single column is enough, adding more columns makes it a **composite super key**, not a composite candidate key.

🔷 Example Table: Enrollment

```
CREATE TABLE Enrollment (
    student_id INT NOT NULL,
    course_id INT NOT NULL,
    enrollment_date DATE,
    course_name VARCHAR(100),
    CONSTRAINT pk_enrollment PRIMARY KEY (student_id, course_id)  -- Composite Primary Key
);
```

🔷 Variations:

1. **Primary Key on single column**

```
CONSTRAINT pk_enrollment PRIMARY KEY (student_id)
```

1. **Primary Key on two columns (composite)**

```
CONSTRAINT pk_enrollment PRIMARY KEY (student_id, course_id)
```

1. **Primary Key on three columns**

```
CONSTRAINT pk_enrollment PRIMARY KEY (student_id, course_id, course_name)
```

- Adding extra columns beyond what's needed makes it **non-minimal**.
- Only the **minimal combination** can be a candidate key.

🔷 Summary Table

Key Type	Definition	Example in Table	Minimal?
Primary Key	Chosen candidate key, unique & NOT NULL	`student_id` or `(student_id, course_id)`	Yes
Candidate Key	Minimal unique key	`email`, `phone`	Yes
Super Key	Any unique combination	`(student_id, phone)`	No
Composite Key	Key with ≥2 columns	`(student_id, course_id)`	Can be candidate or super key depending on minimality

🔑 What is `CASE WHEN`?

The `CASE` expression is SQL's **if-else logic**.

It lets you check conditions and return values accordingly.

Syntax

```
CASE
  WHEN condition1 THEN result1
  WHEN condition2 THEN result2
  ...
  ELSE default_result
END
```

🧩 Types of `CASE`

1. **Simple CASE**

Compares an expression against values.

```
SELECT customer_name,
       amount,
       CASE amount
           WHEN 5000 THEN 'Exactly 5000'
           WHEN 3000 THEN 'Exactly 3000'
           ELSE 'Other Amount'
       END AS amount_status
FROM CustomerData;
```

---

2. **Searched CASE** (most common)

Checks conditions one by one (like `if-else-if`).

```
SELECT customer_name,
       amount,
       CASE
          WHEN amount > 4000 THEN 'High Spender'
          WHEN amount BETWEEN 2000 AND 4000 THEN 'Medium Spender'
          WHEN amount <= 2000 THEN 'Low Spender'
          ELSE 'No Data'
       END AS spending_category
FROM CustomerData;
```

⚡ Real Examples with Your `CustomerData`

Example A – Handle Missing Contact

```
SELECT customer_name,
       CASE
           WHEN email IS NULL AND phone_number IS NULL THEN '9999'
           WHEN email IS NULL THEN phone_number
           ELSE email
       END AS contact
FROM CustomerData;
```

Example B – Combine Contact + Spending

```
SELECT customer_name,
    CASE
      WHEN amount > 4000 THEN 'High Spender'
      WHEN amount BETWEEN 2000 AND 4000 THEN 'Medium Spender'
      WHEN amount <= 2000 THEN 'Low Spender'
      ELSE 'No Data'
    END AS spending_category,
    CASE
      WHEN email IS NULL AND phone_number IS NULL THEN '9999'
      WHEN email IS NULL THEN phone_number
      ELSE email
    END AS contact
FROM CustomerData;
```

👉 So, `CASE WHEN` = **conditional logic inside SQL** (works in `SELECT`, `WHERE`, `ORDER BY`, `GROUP BY`).

🔷 1. Aggregation Functions

SQL provides aggregate functions that work on a set of rows and return a single value.

- `COUNT(*)` → counts rows
- `AVG(amount)` → average
- `MIN(amount)` → minimum
- `MAX(amount)` → maximum
- `SUM(amount)` → total

Examples

```
CREATE TABLE CustomerTransactions (
  transaction_id INT PRIMARY KEY AUTO_INCREMENT,
  customer_id INT NOT NULL,
  customer_name VARCHAR(100) NOT NULL,
  transaction_date DATE NOT NULL,
  transaction_status VARCHAR(20) NOT NULL, -- e.g., 'Success', 'Failed', 'Pending'
  amount DECIMAL(10, 2) NOT NULL,        -- transaction amount
  login_device VARCHAR(30) NOT NULL       -- e.g., 'Mobile', 'Desktop', 'Tablet'
);
```

```
INSERT INTO CustomerTransactions (customer_id, customer_name, transaction_date, transaction_status, amount, l
ogin_device)
VALUES
(101, 'Arun Kumar',     '2025-09-01', 'Success', 1500.00, 'Mobile'),
(102, 'Priya Sharma',   '2025-09-02', 'Failed',  2000.00, 'Desktop'),
(103, 'Rahul Das',      '2025-09-03', 'Success',  750.50, 'Tablet'),
(104, 'Meena Iyer',     '2025-09-04', 'Pending', 1200.00, 'Mobile'),
(105, 'Vikram Joshi',   '2025-09-05', 'Success',  980.75, 'Desktop'),
(106, 'Sneha Reddy',    '2025-09-06', 'Success',  500.00, 'Mobile'),
(107, 'Karthik Nair',   '2025-09-07', 'Failed',   300.00, 'Tablet'),
(108, 'Divya Menon',    '2025-09-08', 'Success', 2200.00, 'Desktop'),
(109, 'Anil Kapoor',    '2025-09-09', 'Pending', 1000.00, 'Mobile'),
(110, 'Lakshmi Rao',    '2025-09-10', 'Success', 1350.25, 'Tablet');
```

```
-- Count only successful transactions
SELECT COUNT(*) AS success_count
```

```
FROM CustomerTransactions
WHERE transaction_status NOT IN ('Failed','Pending');

-- Average transaction amount
SELECT AVG(amount) AS avg_transaction
FROM CustomerTransactions;

-- Minimum transaction amount
SELECT MIN(amount) AS min_transaction
FROM CustomerTransactions;

-- Maximum transaction amount
SELECT MAX(amount) AS max_transaction
FROM CustomerTransactions;

-- Total revenue
SELECT SUM(amount) AS total_revenue
FROM CustomerTransactions;
```

🔷 2. GROUP BY

`GROUP BY` groups rows that have the same values in specified column(s), and then aggregate functions apply to each group.

Example

```
SELECT
    login_device,
    SUM(amount) AS total_revenue
FROM CustomerTransactions
GROUP BY login_device;
```

✅ Output (illustration):

login_device	total_revenue
Mobile	77000
Desktop	85000
Tablet	81500

🔷 3. HAVING

- `WHERE` filters **before grouping**.
- `HAVING` filters **after grouping (on aggregated values)**.

Example

```
SELECT
    login_device,
    SUM(amount) AS total_revenue
FROM CustomerTransactions
GROUP BY login_device
HAVING SUM(amount) > 80000;
```

✅ Output: only devices where `total_revenue > 80000`.

For example:

login_device	total_revenue
Desktop	85000
Tablet	81500

🔷 4. Aliases

- Short names for columns or tables.
- Make query results more readable.

Examples:

```
SELECT login_device AS Device,
       SUM(amount) AS Total_Revenue
FROM CustomerTransactions
GROUP BY login_device
HAVING SUM(amount) > 80000;
```

Null Handling

🔷 1. `IS NULL`

Checks if a column has a `NULL` value.

```
SELECT COUNT(*) AS null_phone_count
FROM CustomerData
WHERE phone_number IS NULL;
```

✅ Counts rows where `phone_number` is `NULL`.

Result: `2` (Arjun, Meena).

🔷 2. `IS NOT NULL`

Checks if a column has a **non-null** value.

```
SELECT COUNT(*) AS not_null_phone_count
FROM CustomerData
WHERE phone_number IS NOT NULL;
```

✅ Counts rows where `phone_number` is **not NULL**.

Result: `3` (Ravi, Priya, Karthik).

🔷 3. Select Rows with At Least One Null

Check across multiple columns.

```
SELECT id, customer_name, email, phone_number, address
FROM CustomerData
WHERE email IS NULL OR phone_number IS NULL OR address IS NULL;
```

✅ Returns rows where any of these are missing.

Example output: Priya (no email, no amount), Arjun (no phone), Meena (no email + no phone), Karthik (no address).

🔷 4. Invalid NULL check

```
SELECT *
FROM CustomerData
WHERE email = NULL;   -- ❌ Wrong
```

⚠️ This always returns **0 rows** because `NULL` means "unknown".

👉 Correct way:

```
WHERE email IS NULL;
```

🔷 5. `COALESCE()`

Replaces `NULL` with the **first non-null value** from a list.

```
SELECT customer_name,
       COALESCE(amount, 0) AS adjusted_amount
FROM CustomerData;
```

✅ If `amount` is `NULL`, replace with `0`.

E.g., Priya → 0.

🔷 6. `IFNULL()` (MySQL only)

Similar to `COALESCE()`, but only takes **two arguments**.

```
SELECT customer_name,
       IFNULL(amount, 0) AS adjusted_amount
FROM CustomerData;
```

✅ Works like `COALESCE(amount, 0)` in MySQL.

String Handling

```
SELECT
id,
customer_name,
-- Extract the length of the customer name
LENGTH(customer_name) AS name_length,
-- Convert customer name to uppercase and lowercase
UPPER(customer_name) AS uppercase_name,
LOWER(customer_name) AS lowercase_name,

-- Concatenate city and phone number with formatting
CONCAT(city, ' - ', COALESCE(phone_number, '00000')) AS contact_info,

-- Extract a substring from the customer name
SUBSTRING(customer_name, 1, 5) AS name_prefix,

-- Trim whitespace from a sample city string
TRIM('   ExampleCity   ') AS trimmed_city,

-- Pad customer name on the left and right
LPAD(customer_name, 15, '*') AS left_padded_name,
RPAD(customer_name, 15, '-') AS right_padded_name,
```

```
-- Replace spaces in customer name with underscores
REPLACE(customer_name, ' ', '_') AS updated_name,

-- Find the position of the letter 'a' in customer name
INSTR(customer_name, 'a') AS position_of_a,

-- Extract the first 5 and last 5 characters from the customer name
LEFT(customer_name, 5) AS first_5_chars,
RIGHT(customer_name, 5) AS last_5_chars,

-- Reverse the customer name
REVERSE(customer_name) AS reversed_name,

-- Format a sample number
FORMAT(9876543210, 2) AS formatted_number
FROM
CustomerDetails;
```

🔷 Sample Table (for reference)

Suppose we have this table:

```
CREATE TABLE CustomerDetails (
    id INT PRIMARY KEY,
    customer_name VARCHAR(100),
    city VARCHAR(100),
    phone_number VARCHAR(15)
);

INSERT INTO CustomerDetails VALUES
(1, 'Ravi Kumar', 'Chennai', '98765'),
(2, 'Priya', 'Bangalore', NULL),
(3, 'Arjun', 'Hyderabad', '12345'),
(4, 'Meena', 'Mumbai', '67890'),
(5, 'Karthik', 'Pune', '54321');
```

🔷 String Handling Functions Demo

```
SELECT
    id,
    customer_name,

    -- 1️⃣ Find length
    LENGTH(customer_name) AS name_length,

    -- 2️⃣ Convert case
    UPPER(customer_name) AS uppercase_name,
    LOWER(customer_name) AS lowercase_name,

    -- 3️⃣ Concatenate with fallback for NULL
    CONCAT(city, ' - ', COALESCE(phone_number, '00000')) AS contact_info,

    -- 4️⃣ Substring (first 5 chars)
    SUBSTRING(customer_name, 1, 5) AS name_prefix,
```

```
    -- 5️⃣ Trim whitespace
    TRIM('   ExampleCity   ') AS trimmed_city,

    -- 6️⃣ Padding
    LPAD(customer_name, 15, '*') AS left_padded_name,
    RPAD(customer_name, 15, '-') AS right_padded_name,

    -- 7️⃣ Replace spaces with underscores
    REPLACE(customer_name, ' ', '_') AS updated_name,

    -- 8️⃣ Position of letter 'a'
    INSTR(customer_name, 'a') AS position_of_a,

    -- 9️⃣ First and last 5 chars
    LEFT(customer_name, 5) AS first_5_chars,
    RIGHT(customer_name, 5) AS last_5_chars,

    -- 🔟 Reverse name
    REVERSE(customer_name) AS reversed_name,

    -- 1️⃣1️⃣ Format a number
    FORMAT(9876543210, 2) AS formatted_number

FROM
    CustomerDetails;
```

🔷 Example Output (illustration for Ravi Kumar)

customer_name	name_length	uppercase_name	lowercase_name	contact_info	name_prefix	trimmed_city	left_p
Ravi Kumar	10	RAVI KUMAR	ravi kumar	Chennai - 98765	Ravi	ExampleCity	*****

🔑 Key Takeaways

- **Length/Case** → `LENGTH()`, `UPPER()`, `LOWER()`.
- **Concatenation** → `CONCAT()` + `COALESCE()` for null safety.
- **Substring/Trim** → `SUBSTRING()`, `TRIM()`.
- **Padding** → `LPAD()`, `RPAD()`.
- **Search/Replace** → `INSTR()`, `REPLACE()`.
- **Extract** → `LEFT()`, `RIGHT()`.
- **Reverse/Format** → `REVERSE()`, `FORMAT()`.

🔷 Subqueries in SQL

A **subquery** = a query inside another query.

It can appear in the `SELECT`, `WHERE`, `FROM`, `HAVING`, or even inside `CASE`.

1️⃣ Subquery in `SELECT` → **Derived Value for Each Row**

You have two tables: `Customers` and `Orders`. You want to display each customer's name along with the **highest order**

amount found in the entire `Orders` table. This is done using a **scalar subquery** that returns a single value—`MAX(order_amount)`—and includes it in every row of the result.

```
SELECT
    customer_name,
    (SELECT MAX(order_amount) FROM Orders) AS max_order_amount
FROM Customers;
```

👉 For every customer, we attach the same **max order amount** from the `Orders` table.

2️⃣ Subquery in `WHERE`

Example 1: `IN`

```
SELECT customer_name
FROM Customers
WHERE customer_id IN (
    SELECT DISTINCT customer_id
    FROM Orders
    WHERE order_amount > 5000
);
```

✅ Returns customers who placed at least one order > 5000.

Example 2: `EXISTS`

```
CREATE TABLE Customers (
customer_id INT PRIMARY KEY,
customer_name VARCHAR(100) NOT NULL,
email VARCHAR(100),
city VARCHAR(50)
);

INSERT INTO Customers (customer_id, customer_name, email, city)
VALUES
(1, 'Arun Kumar',     'arun.kumar@example.com',     'Chennai'),
(2, 'Priya Sharma',   'priya.sharma@example.com',   'Delhi'),
(3, 'Meena Iyer',     'meena.iyer@example.com',     'Mumbai'),
(4, 'Rahul Das',      'rahul.das@example.com',      'Kolkata'),
(5, 'Sneha Reddy',    'sneha.reddy@example.com',    'Hyderabad');
```

```
CREATE TABLE Orders (
    order_id INT PRIMARY KEY AUTO_INCREMENT,
    customer_id INT NOT NULL,
    order_date DATE NOT NULL,
    order_amount DECIMAL(10, 2) NOT NULL,
    FOREIGN KEY (customer_id) REFERENCES Customers(customer_id)
);

INSERT INTO Orders (customer_id, order_date, order_amount)
VALUES
(1, '2025-09-10', 4500.00),  -- Recent
(2, '2025-08-20', 6200.00),  -- Recent
(3, '2025-07-25', 3000.00),  -- Older
(4, '2025-09-01', 5100.00),  -- Recent
(5, '2025-08-10', 2500.00);  -- Older
```

```
SELECT customer_name
FROM Customers
WHERE EXISTS (
    SELECT 1
    FROM Orders
    WHERE Orders.customer_id = Customers.customer_id
    AND order_date >= CURDATE() - INTERVAL 30 DAY
);
```

✅ Returns customers who placed an order in the last 30 days.

- `EXISTS` is faster than `IN` when checking row existence.

3️⃣ Subquery Instead of Join

```
SELECT
    customer_name,
    city,
    (SELECT SUM(order_amount)
     FROM Orders o
     WHERE o.customer_id = c.customer_id) AS total_orders
FROM Customers c;
```

✅ For each customer, get total orders (without explicitly joining).

4️⃣ Subquery in `FROM` → Derived Table

```
SELECT
    id,
    customer_name,
    name_length,
    uppercase_name,
    lowercase_name,
    contact_info,
    name_prefix,
    trimmed_city,
    left_padded_name,
    right_padded_name,
    updated_name,
    position_of_a,
    first_5_chars,
    last_5_chars,
    reversed_name,
    formatted_number
FROM (
    SELECT
        id,
        customer_name,
        LENGTH(customer_name) AS name_length,
        UPPER(customer_name) AS uppercase_name,
        LOWER(customer_name) AS lowercase_name,
        CONCAT(city, ' - ', COALESCE(phone_number, '00000')) AS contact_info,
        SUBSTRING(customer_name, 1, 5) AS name_prefix,
        TRIM('   ExampleCity   ') AS trimmed_city,
        LPAD(customer_name, 15, '*') AS left_padded_name,
        RPAD(customer_name, 15, '-') AS right_padded_name,
```

```
        REPLACE(customer_name, ' ', '_') AS updated_name,
        INSTR(customer_name, 'a') AS position_of_a,
        LEFT(customer_name, 5) AS first_5_chars,
        RIGHT(customer_name, 5) AS last_5_chars,
        REVERSE(customer_name) AS reversed_name,
        FORMAT(9876543210, 2) AS formatted_number
    FROM CustomerDetails
) AS string_handling_results;
```

✅ Treats the **inner query** as a temporary table.

5️⃣ Subquery with `CASE` → Conditional Filtering

```
SELECT
    customer_name,
    CASE
        WHEN (SELECT SUM(order_amount)
              FROM Orders
              WHERE Orders.customer_id = Customers.customer_id) >
             (SELECT AVG(order_amount) FROM Orders)
        THEN 'Above Average'
        ELSE 'Below Average'
    END AS order_category
FROM Customers;
```

✅ Compares **each customer's total order value** vs. **overall average**.

6️⃣ Subquery for Ranking (Second Highest, Nth Value, etc.)

```
SELECT
    customer_name,
    (SELECT MAX(order_amount)
     FROM Orders
     WHERE order_amount < (SELECT MAX(order_amount) FROM Orders))
     AS second_highest_order
FROM Customers;
```

✅ Finds the **second highest order amount**.

(You can extend this idea for third-highest, etc.)

🔑 Summary of Subquery Uses

- **SELECT** → add derived values.
- **WHERE / HAVING** → filter using conditions.
- **EXISTS vs IN** → check for matching rows.
- **FROM** → create derived tables.
- **CASE** → classify rows with subquery comparisons.
- **Ranking** → find Nth max/min values.

🔷 What is a View?

- A **view** is like a *virtual table*.
- It does **not** store data physically (unlike a real table).

- Instead, whenever you query the view, it runs the underlying SQL query and fetches the results dynamically.
- Views are useful for **simplifying queries**, **restricting access** to certain columns/rows, and **reusing complex queries**.

🔷 Your Example Explained

1. **Base Table**

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName  VARCHAR(50),
    LastName   VARCHAR(50),
    Salary DECIMAL(10,2),
    Department VARCHAR(50)
);
```

This is your real table holding employee data.

2. **Sample Data**

```
INSERT INTO Employees (EmployeeID, FirstName, LastName, Salary, Department)
VALUES
    (1, 'Alice', 'Johnson', 55000.00, 'Sales'),
    (2, 'Bob',   'Smith',   60000.00, 'IT'),
    (3, 'Carol', 'Davis',   52000.00, 'Sales'),
    (4, 'Dave',  'Wilson',  58000.00, 'HR');
```

Now the table has employees across multiple departments.

3. **View Creation**

```
CREATE VIEW SalesEmployees AS
SELECT EmployeeID, FirstName, LastName, Salary
FROM Employees
WHERE Department = 'Sales';
```

🔶 This creates a **virtual table** named `SalesEmployees`.

🔶 It only exposes employees from the `Sales` department.

1. **Querying the View**

```
SELECT * FROM SalesEmployees;
```

👉 Output will be:

```
EmployeeID | FirstName | LastName | Salary
-----------+-----------+----------+--------
1          | Alice     | Johnson  | 55000.00
3          | Carol     | Davis    | 52000.00
```

🔷 Key Points About Views

- **Read-only vs Updatable**:

Some views can be updated (if based on a single table without aggregates/joins), but others are read-only.

- **Security**:

  You can restrict sensitive columns by exposing only what's needed.

- **Reusability**:

  Instead of writing `WHERE Department = 'Sales'` everywhere, just query `SalesEmployees`.

1️⃣ INNER JOIN

Returns rows that have matches in **both** tables.

```
CREATE TABLE Restaurants (
id INT  PRIMARY KEY,
name VARCHAR(100) NOT NULL,
location VARCHAR(100) NOT NULL
);
```

```
INSERT INTO Restaurants (id, name, location) VALUES
(1, 'ABC Bistro', 'New York'),
(2, 'The Foodie', 'Los Angeles'),
(3, 'Tasty Treat', 'Chicago');
```

```
CREATE TABLE Orders (
order_id INT PRIMARY KEY,
restaurant_id INT NOT NULL,
order_date DATE NOT NULL
);
```

```
INSERT INTO Orders (order_id,restaurant_id, order_date) VALUES
(1, 1, '2023-01-01'),
(2, 1, '2023-01-02'),
(3, 2, '2023-01-05'),
(4, 4, '2023-01-07');
```

```
SELECT
   r.name AS restaurant_name,
   o.order_date
FROM Restaurants r
JOIN Orders o
   ON r.id = o.restaurant_id;
```

✅ Output (from your data):

```
restaurant_name | order_date
----------------+------------
ABC Bistro      | 2023-01-01
ABC Bistro      | 2023-01-02
The Foodie      | 2023-01-05
```

Notice: `order_id = 4` refers to `restaurant_id = 4`, which doesn't exist in `Restaurants`, so it's excluded.

2️⃣ LEFT JOIN

Returns **all restaurants** even if they have no orders.

```
SELECT
    r.name AS restaurant_name,
    o.order_date
FROM Restaurants r
LEFT JOIN Orders o
    ON r.id = o.restaurant_id;
```

✅ Output:

```
restaurant_name | order_date
----------------+------------
ABC Bistro      | 2023-01-01
ABC Bistro      | 2023-01-02
The Foodie      | 2023-01-05
Tasty Treat     | NULL
```

👉 `Tasty Treat` has no orders, so `order_date` is `NULL`.

3️⃣ RIGHT JOIN

Returns **all orders** even if no restaurant exists for them.

```
SELECT
    r.name AS restaurant_name,
    o.order_date
FROM Restaurants r
RIGHT JOIN Orders o
    ON r.id = o.restaurant_id;
```

✅ Output:

```
restaurant_name | order_date
----------------+------------
ABC Bistro      | 2023-01-01
ABC Bistro      | 2023-01-02
The Foodie      | 2023-01-05
NULL            | 2023-01-07
```

👉 The last row has `NULL` for `restaurant_name` because `restaurant_id = 4` doesn't exist in `Restaurants`.

⚡ **Extra Note:**

- `INNER JOIN` = Only matches
- `LEFT JOIN` = All left + matches
- `RIGHT JOIN` = All right + matches

Employees

```
CREATE TABLE employees (
employee_id INT AUTO_INCREMENT PRIMARY KEY,
first_name  VARCHAR(50),
last_name   VARCHAR(50),
position    VARCHAR(50),
salary      DECIMAL(10,2)
);
INSERT INTO employees (first_name, last_name, position, salary)
```

```
VALUES
('Alice', 'Smith', 'Developer', 70000.00),
('Bob', 'Johnson', 'Developer', 75000.00),
('Charlie', 'Lee', 'Manager', 90000.00);
```

Contractors

```
CREATE TABLE contractors (
contractor_id INT AUTO_INCREMENT PRIMARY KEY,
first_name    VARCHAR(50),
last_name     VARCHAR(50),
position      VARCHAR(50),
hourly_rate   DECIMAL(10,2)
);
INSERT INTO contractors (first_name, last_name, position, hourly_rate)
VALUES
('Dave', 'Williams', 'Developer', 40.00),
('Eve', 'Brown', 'Tester', 35.00),
('Bob', 'Johnson', 'Developer', 45.00);
```

```
contractor_id | first_name | last_name | position   | hourly_rate
--------------+------------+-----------+------------+------------
1             | Dave       | Williams  | Developer  | 40
2             | Eve        | Brown     | Tester     | 35
3             | Bob        | Johnson   | Developer  | 45
```

1️⃣ Using `UNION`

```
SELECT employee_id, first_name
FROM employees
UNION
SELECT contractor_id, first_name
FROM contractors;
```

➡️ `UNION` **removes duplicates** (per row).

Result:

```
employee_id | first_name
------------+-----------
1           | Alice
2           | Bob
3           | Charlie
1           | Dave
2           | Eve
```

🔷 Notice: `"Bob"` appears **only once**, even though he exists in both tables.

2️⃣ Using `UNION ALL`

```
SELECT employee_id, first_name
FROM employees
UNION ALL
SELECT contractor_id, first_name
FROM contractors;
```

➡️ `UNION ALL` **keeps duplicates**.

Result:

```
employee_id | first_name
------------+-----------
1           | Alice
2           | Bob
3           | Charlie
1           | Dave
2           | Eve
3           | Bob
```

🔷 `"Bob"` appears **twice** (from employees and contractors).

⚡ **Key Difference**

- `UNION` → combines and removes duplicates
- `UNION ALL` → combines and keeps everything (faster, because no duplicate check)

Windows Function

🔷 1. Aggregates with `OVER`

```
SELECT TransactionID, Store, SalesAmount,
       SUM(SalesAmount) OVER (PARTITION BY Store ORDER BY TransactionID DESC) AS TotalSales
FROM Sales;
```

- Without `GROUP BY`, you can still calculate totals.
- `PARTITION BY Store` → reset sum per store.
- Ordered → running total (like cumulative sum).

🔷 2. Row Numbering Functions

`ROW_NUMBER()`

```
ROW_NUMBER() OVER (PARTITION BY Store ORDER BY SalesAmount DESC)
```

- Assigns unique sequential numbers.
- No gaps, but same values still get different row numbers.

`RANK()` vs `DENSE_RANK()`

```
RANK() OVER (ORDER BY ExamScore DESC)
DENSE_RANK() OVER (ORDER BY ExamScore DESC)
```

- **RANK**: If ties → skips numbers. (95, 95 → 1, 1, next = 3)
- **DENSE_RANK**: If ties → no skipping. (95, 95 → 1, 1, next = 2)

🔷 3. Distribution

`NTILE(n)`

```
NTILE(4) OVER (ORDER BY SalesAmount DESC)
```

- Splits rows into `n` buckets.

- Useful for performance quartiles, salary bands, etc.

`PERCENT_RANK()`

```
PERCENT_RANK() OVER (ORDER BY SalesAmount DESC)
```

- Relative standing of a row between 0 and 1.
- First row = 0, last row = 1.

🔷 4. Navigation (LAG/LEAD)

```
LAG(Salary) OVER (PARTITION BY EmployeeID ORDER BY Year) AS PreviousYearSalary
LEAD(SaleAmount) OVER (ORDER BY SaleDate) AS NextSaleAmount
```

- `LAG` → look at previous row value.
- `LEAD` → look at next row value.
- Perfect for **year-over-year difference** or **trend analysis**.

🔷 5. First & Last Values

```
FIRST_VALUE(Salary) OVER (PARTITION BY EmployeeID ORDER BY Year
    ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING) AS firstsalary
LAST_VALUE(Salary) OVER (PARTITION BY EmployeeID ORDER BY Year
    ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING) AS lastsalary
```

⚠️ **Important**: Without frame (`ROWS BETWEEN ...`), `LAST_VALUE` may behave unexpectedly, because default window = `UNBOUNDED PRECEDING TO CURRENT ROW`.

🔷 6. NTH Value

```
NTH_VALUE(Salary, 2) OVER (PARTITION BY EmployeeID ORDER BY Year) AS SecondSalary
```

- Fetches the 2nd value from the ordered partition.
- Useful when you want **nth purchase, nth salary hike, etc.**

✅ So your examples cover:

- **Aggregates** (`SUM`)
- **Numbering** (`ROW_NUMBER` , `RANK` , `DENSE_RANK`)
- **Distribution** (`NTILE` , `PERCENT_RANK`)
- **Navigation** (`LAG` , `LEAD`)
- **Extremes** (`FIRST_VALUE` , `LAST_VALUE` , `NTH_VALUE`)

Index and Explain

🔷 Step 1: Table + Index

```
CREATE TABLE customers (
   customer_id INT AUTO_INCREMENT PRIMARY KEY,
   first_name  VARCHAR(50) NOT NULL,
   last_name   VARCHAR(50) NOT NULL,
   email       VARCHAR(100) NOT NULL,
```

```
    city       VARCHAR(100) NOT NULL
);

-- Create index on email
CREATE INDEX idx_email ON customers (email);
```

Here:

- `PRIMARY KEY (customer_id)` → automatically creates a **clustered index** (unique & ordered).
- `idx_email` → is a **secondary index** on `email`, helpful for fast lookups.

🔷 Step 2: Query without index (if index not created)

```
SELECT *
FROM customers
WHERE email = 'john@example.com';
```

If no index on `email`:

- MySQL will perform a **full table scan** (`ALL` in EXPLAIN).
- It checks every row until it finds matches.
- Bad for large tables.

🔷 Step 3: Query with index (`idx_email`)

Now, since `idx_email` exists:

- MySQL can do an **Index Lookup** → directly jump to rows with that `email`.
- Much faster.

🔷 Step 4: EXPLAIN

```
EXPLAIN
SELECT *
FROM customers
WHERE email = 'john@example.com';
```

👉 Possible output (simplified):

id	select_type	table	type	possible_keys	key	key_len	ref
1	SIMPLE	customers	ref	idx_email	idx_email	303	const

- **type = ref** → lookup using index.
- **key = idx_email** → MySQL is using the index.
- **rows = 1** → expected only 1 row checked.
- **Extra = Using index** → all needed data fetched from index (covering index case).

🔷 Step 5: EXPLAIN ANALYZE (MySQL 8+)

```
EXPLAIN ANALYZE
SELECT *
FROM customers
WHERE email = 'john@example.com';
```

👉 Possible output:

```
→ Index lookup on customers using idx_email (email='john@example.com')  (cost=0.35 rows=1) (actual time=0.05..0.06 rows=1 loops=1)
```

Breakdown:

- **Index lookup** → confirms index usage.
- **cost=0.35** → optimizer's estimated cost.
- **rows=1** → expected rows.
- **actual time=0.05..0.06** → real execution time.
- **rows=1** → actual rows fetched.

✅ **Key Difference**

- `EXPLAIN` → shows **optimizer's plan** (what MySQL *thinks* it will do).
- `EXPLAIN ANALYZE` → shows **actual runtime statistics** (what MySQL *actually did*).

Partition

🔷 1. Range Partitioning

```
CREATE TABLE orders (
    order_id   INT AUTO_INCREMENT,
    order_date DATE NOT NULL,
    customer_name VARCHAR(50),
    amount     DECIMAL(10,2),
    PRIMARY KEY(order_id, order_date)
)
PARTITION BY RANGE (YEAR(order_date)) (
    PARTITION p_before_2020 VALUES LESS THAN (2020),
    PARTITION p_2020      VALUES LESS THAN (2021),
    PARTITION p_2021      VALUES LESS THAN (2022),
    PARTITION p_2022      VALUES LESS THAN (2023),
    PARTITION p_future    VALUES LESS THAN MAXVALUE
);
```

- Data is **distributed by year** of `order_date`.
- `EXPLAIN FORMAT=JSON` with:

```
SELECT * FROM orders WHERE order_date = '2021-07-20';
```

➡️ MySQL **prunes partitions**: only checks `p_2021`.

- Huge performance win on large datasets.

🔷 2. List Partitioning

```
CREATE TABLE employees (
    employee_id INT AUTO_INCREMENT,
    first_name  VARCHAR(50),
    last_name   VARCHAR(50),
    department  VARCHAR(50),
    PRIMARY KEY (employee_id, department)
)
```

```
PARTITION BY LIST COLUMNS (department) (
    PARTITION p_sales       VALUES IN ('Sales'),
    PARTITION p_hr          VALUES IN ('HR'),
    PARTITION p_engineering VALUES IN ('Engineering', 'DevOps'),
    PARTITION p_other       VALUES IN ('Finance', 'Marketing', 'Operations')
);
```

- Rows are grouped by **discrete values** (`department`).
- Query:

```
SELECT * FROM employees WHERE department = 'Sales';
```

➡️ Only partition `p_sales` is scanned.

🔷 3. Hash Partitioning

```
CREATE TABLE sensor_data (
    sensor_id     INT NOT NULL,
    reading_time  DATETIME NOT NULL,
    reading_value DECIMAL(10,2),
    PRIMARY KEY (sensor_id, reading_time)
)
PARTITION BY HASH(sensor_id)
PARTITIONS 2;
```

- Hashing spreads rows **evenly** across partitions (good for large streaming data).
- Query:

```
EXPLAIN FORMAT=JSON
SELECT * FROM sensor_data WHERE sensor_id = 102;
```

➡️ Hash of `102` decides the partition, MySQL **jumps directly** to the right one.

- Great for **random distribution**.

🔷 4. Subpartitioning (Composite Partitioning)

```
CREATE TABLE orders (
    order_id      INT AUTO_INCREMENT PRIMARY KEY,
    order_date    DATE NOT NULL,
    customer_name VARCHAR(50),
    amount        DECIMAL(10,2)
)
PARTITION BY RANGE (YEAR(order_date))
SUBPARTITION BY HASH (MONTH(order_date))
PARTITIONS 3
SUBPARTITIONS 2
(
  PARTITION p_before_2020 VALUES LESS THAN (2020),
  PARTITION p_2020        VALUES LESS THAN (2021),
  PARTITION p_future      VALUES LESS THAN MAXVALUE
```

```
);
```

- First split by **year**, then inside each partition split by **month hash**.
- Useful when you need **two-level partitioning** (e.g., by year + monthly balance).

🔷 5. Partition with Index

```
CREATE TABLE orders (
    order_id      INT AUTO_INCREMENT PRIMARY KEY,
    order_date    DATE NOT NULL,
    customer_name VARCHAR(50),
    amount        DECIMAL(10,2)
)
PARTITION BY RANGE (YEAR(order_date)) (
    PARTITION p_before_2020 VALUES LESS THAN (2020),
    PARTITION p_2020       VALUES LESS THAN (2021),
    PARTITION p_2021       VALUES LESS THAN (2022),
    PARTITION p_2022       VALUES LESS THAN (2023),
    PARTITION p_future     VALUES LESS THAN MAXVALUE
);

CREATE INDEX idx_order_date ON orders (order_date);
```

- When you query:

```
SELECT *
FROM orders
WHERE order_date BETWEEN '2020-01-01' AND '2020-12-31';
```

➡️ Optimizer **prunes partitions (** `p_2020` **)** AND uses the **index** inside that partition.

- **Double optimization**: partition elimination + index lookup.

🔑 Key Takeaways

1. **Range** → best for time-series data (e.g., logs by year).
2. **List** → best for categories (e.g., department).
3. **Hash** → best for evenly spreading data (e.g., IoT sensors).
4. **Subpartition** → combine strategies (e.g., year + month).
5. **Partition + Index** → ideal combo for **fast pruning + fast lookup**.

Date and Time

🔷 Key Concepts First

- `DATETIME`
    - Stores the exact date/time value as entered (no time zone conversion).
    - Range: `'1000-01-01 00:00:00'` → `'9999-12-31 23:59:59'`.
    - Takes **8 bytes** of storage.
    - Think of it as a **fixed point** in time, not tied to a time zone.

- `TIMESTAMP`
  - Internally stored as **UTC**, converted automatically to/from the **current session's time zone** when inserted/retrieved.
  - Range: `'1970-01-01 00:00:01 UTC'` → `'2038-01-19 03:14:07 UTC'`.
  - Takes **4 bytes** of storage.
  - Often used for **created/modified times** with automatic updates.

🔷 Step-by-Step in Your Script

1. Create Database + Table

```
CREATE DATABASE IF NOT EXISTS datetime_vs_timestamp;
USE datetime_vs_timestamp;

CREATE TABLE demo_events (
    event_id INT AUTO_INCREMENT PRIMARY KEY,
    event_name VARCHAR(100),
    event_date DATETIME,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

✅ Table has both a `DATETIME` (`event_date`) and a `TIMESTAMP` (`created_at`).

---

2. Insert Sample Data

```
INSERT INTO demo_events (event_name, event_date)
VALUES
    ('New Year Celebration', '2025-01-01 00:00:00'),
    ('Summer Fest', '2025-06-15 12:30:00');
```

- `event_date` → stored **as-is** (`DATETIME`).
- `created_at` → auto-filled **in UTC**, displayed in session time zone.

---

3. Select in Default Time Zone

```
SELECT event_id, event_name, event_date, created_at FROM demo_events;
```

Suppose your server is in `+00:00 (UTC)`:

- `event_date` → `2025-01-01 00:00:00` (unchanged)
- `created_at` → `2025-01-01 00:00:00` (UTC stored = UTC displayed)

---

4. Change Session Time Zone

```
SET time_zone = 'America/Los_Angeles';
```

```
SELECT event_id, event_name, event_date, created_at FROM demo_events;
```

Now results differ:

- **`event_date (DATETIME)`** → stays the same (`2025-01-01 00:00:00`) ❌ no conversion
- **`created_at (TIMESTAMP)`** → shifts by `08:00` (Pacific Time) → `2024-12-31 16:00:00`

👉 Proof that `TIMESTAMP` is timezone-aware while `DATETIME` is not.

---

5. Auto-Update Demo

```
ALTER TABLE demo_events
  MODIFY COLUMN created_at TIMESTAMP
    DEFAULT CURRENT_TIMESTAMP
    ON UPDATE CURRENT_TIMESTAMP;

UPDATE demo_events
SET event_name = 'Summer Fest - Updated'
WHERE event_id = 2;
```

✅ Now, whenever you update a row:

- `created_at` automatically updates to the **current timestamp**.
- Perfect for "last modified" columns.

🔑 Takeaways

1. Use `DATETIME` for **business events** where the time is absolute (e.g., wedding date, historical record).
2. Use `TIMESTAMP` for **logging system events** (created_at/updated_at) since it tracks session time zone.
3. Be careful if your app is used in multiple time zones → `TIMESTAMP` may show different values depending on the client's session.

REGEXP

🔷 Key Points about MySQL `REGEXP`

- `REGEXP` matches a string against a regular expression pattern.
- By default, **case-insensitive** (use `REGEXP BINARY` for case-sensitive).
- Uses **POSIX Extended Regular Expressions** (not full PCRE).
- Alternatives: `RLIKE` is the same as `REGEXP`.

🔷 Quick Recap of Your `regex_samples`

✅ You already showed:

- Starts/ends with letters
- Digits at start
- Repeated characters
- String length check
- Specific alternations (`apple|banana`)

👉 This is **great for teaching** core regex basics.

🔷 Extended `demo_data` Use Cases

1️⃣ Strict Date Validation (YYYY-MM-DD ✅ / rejects bad dates like `2025-13-31`)

```
SELECT id, full_name, date_col
FROM demo_data
WHERE date_col REGEXP '^[0-9]{4}-[0-9]{2}-[0-9]{2}$';
```

2️⃣ Detect Bad Emails (e.g., multiple `@` or double dots)

```
SELECT id, email
FROM demo_data
WHERE email NOT REGEXP '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,}$';
```

✅ Flags invalid emails like:

- `alice@@example.net`
- `invalid@@example..com`

3️⃣ Phone Number Validation

👉 Accepts formats like `123-456-7890`, `(987) 654-3210`, `+1-555-123-4567`, `1234567890`.

```
SELECT id, phone
FROM demo_data
WHERE phone REGEXP '^(\\+?[0-9]{1,3}[- ]?)?(\\([0-9]{3}\\)|[0-9]{3})[- ]?[0-9]{3}[- ]?[0-9]{4}$';
```

4️⃣ Detect Status Normalization Issues (case mismatch like `pending` vs `PENDING`)

```
SELECT id, status
FROM demo_data
WHERE status NOT REGEXP '^(pending|inactive|active|complete)$';
```

✅ Flags row with `PENDING` (uppercase).

5️⃣ Validate SKU Codes

👉 Example: `SKU-123`, `SKU999`, but reject `ABCDE` or `XYZ000`.

```
SELECT id, sku
FROM demo_data
WHERE sku REGEXP '^SKU[-]?[0-9]+$';
```

6️⃣ Usernames (only letters, numbers, underscore)

```
SELECT id, username
FROM demo_data
WHERE username NOT REGEXP '^[A-Za-z0-9_]+$';
```

✅ Flags `johnsmith` (OK), `mary_white` (OK), `bob123` (OK),

but rejects things like `invalid!`.

7️⃣ Find Notes Containing `CA` or `NY` (case-insensitive)

```
SELECT id, notes
FROM demo_data
WHERE notes REGEXP '(CA|NY)';
```

Java DataBase Connectivity(JDBC)

🧩 1. Introduction to JDBC

- **JDBC (Java Database Connectivity)** is a standard API in Java that allows applications to **interact with relational databases**.
- It enables **executing SQL queries**, updating records, and retrieving results in a consistent way, regardless of the database vendor (MySQL, Oracle, PostgreSQL, etc.).
- Think of JDBC as the **bridge between Java code and the database**.

🏗️ 2. JDBC Architecture

1. **Application Layer** → Your Java code (the program written by you).
2. **JDBC API** → Provides interfaces like `Connection`, `Statement`, `PreparedStatement`, `ResultSet`.
3. **DriverManager** → Selects and loads the appropriate database driver at runtime.
4. **JDBC Driver** → Database-specific implementation that converts JDBC calls into native DB calls.
5. **Database** → The actual relational database (e.g., MySQL, Oracle, PostgreSQL).

📌 Flow:

`Java Code` → `JDBC API` → `DriverManager` → `Database Driver` → `Database`

🔌 3. Steps to Connect Java with Database

```
// 1. Load the Driver Class
Class.forName("com.mysql.cj.jdbc.Driver");

// 2. Establish Connection
Connection con = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/testdb", "root", "password");

// 3. Create Statement
Statement stmt = con.createStatement();

// 4. Execute Query
ResultSet rs = stmt.executeQuery("SELECT * FROM students");

// 5. Process Results
while(rs.next()) {
    System.out.println(rs.getInt("id") + " " + rs.getString("name"));
}

// 6. Close Resources
rs.close();
stmt.close();
con.close();
```

🧪 4. JDBC Interfaces

Interface	Purpose
Connection	Connects to the database, manages sessions
Statement	Executes static SQL queries (not recommended for dynamic input → risk of SQL Injection)
PreparedStatement	Executes parameterized queries (safer and more efficient)
CallableStatement	Calls stored procedures from the database
ResultSet	Stores and navigates through results of a query