WEEK-1 NOTES

MYSQL

MySQL is an open-source relational database management system (RDBMS) that uses Structured Query Language (SQL) for database operations. It's one of the most popular database systems worldwide, known for its reliability, performance, and ease of use.

Key Features of MySQL

- **Open Source**: Free to use and modify under the GPL license
- **Relational Database**: Stores data in tables with relationships between them
- **Cross-Platform**: Runs on various operating systems (Windows, Linux, macOS)
- **Client-Server Architecture**: Supports multiple clients connecting to the database server
- **ACID Compliance**: Ensures reliability for transactions
- **Scalability**: Handles databases from small applications to enterprise-level systems
- **Replication**: Supports master-slave replication for data redundancy
- **Security**: Robust access control and encryption features

▼ MySQL Workbench

MySQL Workbench is the official integrated development environment (IDE) for MySQL. It provides a visual interface for database design, development, and administration.

MySQL Workbench Features

- **SQL Development**: Editor with syntax highlighting and query execution

- **Data Modeling**: Visual tools to create and modify database schemas
- **Database Administration**: User management, performance monitoring, and backup
- **Database Migration**: Tools to migrate from other database systems
- **Visual Explain Plan**: Graphical representation of query execution plans
- **Server Configuration**: Visual tools for configuring MySQL server settings
- **Data Import/Export**: Import and export functionality for various formats

Forward and Reverse Engineering:

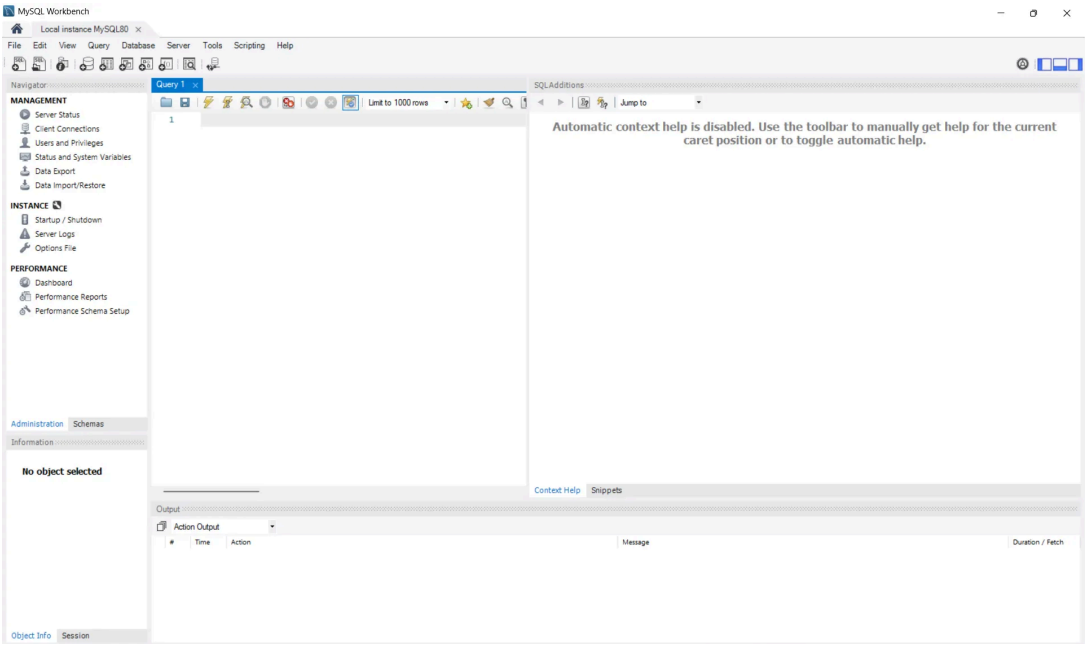
- **Forward Engineering**: This is the magic button. Once your diagram is complete, go to `Database > Forward Engineer`. Workbench will automatically generate the `CREATE TABLE` SQL script for your entire schema. You can then run this script on your server to build the database.
- **Reverse Engineering**: You can connect to an existing database (`Database > Reverse Engineer`) and Workbench will automatically generate a visual ER diagram from it. This is incredibly useful for understanding the structure of a database you've never seen before.

Getting Started with MySQL Workbench

1. **Connection Setup**: Create connections to local or remote MySQL servers
2. **Schema Creation**: Design new databases with visual modeling tools
3. **Query Execution**: Write and run SQL statements with the SQL editor
4. **Result Management**: View, filter, and export query results

5. **Server Monitoring**: Monitor server health and performance metrics

MySQL Workbench simplifies database management tasks through its intuitive interface, making it accessible for both beginners and experienced database administrators.



▼ DDL

DDL stands for **Data Definition Language**.

It is used to **define or modify the structure of database objects** such as:

- Databases
- Tables
- Indexes
- Constraints
- Views

Major Commands are CREATE, DROP, ALTER, RENAME and TRUNCATE

1. CREATE DATABASE

```
CREATE DATABASE database_name;
```

Edge Cases:

- If the database already exists →

```
CREATE DATABASE CompanyDB;
```

Results in error:

ERROR 1007 (HY000): Can't create database 'CompanyDB'; database exists

Fix:

```
CREATE DATABASE IF NOT EXISTS CompanyDB;
```

2. DROP DATABASE

```
DROP DATABASE database_name;
```

Edge Cases:

- If the database doesn’t exist →

```
DROP DATABASE NonExistentDB;
```

Results in error:

ERROR 1008 (HY000): Can't drop database 'NonExistentDB'; database doesn't exist

Fix:

```
DROP DATABASE IF EXISTS CompanyDB;
```

3. CREATE TABLE

```
CREATE TABLE table_name (
    column1 datatype constraints,
    column2 datatype constraints,
    ...
);
```

Example:

```
CREATE TABLE tb01(
    f01 INT AUTO_INCREMENT PRIMARY KEY,
    f02 VARCHAR(100) NOT NULL,
    f03 VARCHAR(100) UNIQUE,
    f04 DATE,
    f05 DECIMAL(10,2)
);
```

▼ Datatypes

Numeric Types

- `INT UNSIGNED`: The standard choice for **primary keys** (`AUTO_INCREMENT`) and counts. `UNSIGNED` ensures it's always a positive number, doubling the capacity.
- `TINYINT(1)`: The standard way to represent a **boolean** value (`0` for false, `1` for true).
- `DECIMAL(p, s)`: **Use this for money!** It provides exact precision for financial data. Example: `DECIMAL(10, 2)` for prices up to `99,999,999.99`.
- `DOUBLE`: For scientific or measurement data where absolute precision isn't critical (e.g., GPS coordinates).

String Types

- `VARCHAR(size)`: Your **go-to for most text**. It's a variable-length string perfect for names, titles, and emails.

- `CHAR(size)`: Only use for **fixed-length** text, like two-letter country codes (`'US'`) or MD5 hashes.
- `TEXT`: For **long-form text** like blog posts, user comments, or detailed descriptions.
- `ENUM('val1', 'val2', ...)`: For columns where the value must be **one of a predefined set** of options, like `status ENUM('pending', 'active', 'failed')`. Very efficient.

Date & Time Types

- `DATE`: For a date only. Format: `'YYYY-MM-DD'`. Use for birth dates, join dates, etc.
- `DATETIME`: Stores both date and time. It is **NOT timezone-aware**. Best for simple event logging.
- `TIMESTAMP`: Stores date and time. It **IS timezone-aware** (converts to/from UTC). The best choice for `created_at` and `updated_at` fields, especially in applications with global users.

Other Key Types

- `JSON`: For storing semi-structured data (like settings or API responses) in a single field. MySQL has special functions to query data inside a JSON object.

▼ Constraint

`PRIMARY KEY` Constraint

This is the most important constraint. It uniquely identifies each row in a table. A table can have only **one** `PRIMARY KEY`.

- **What it does:** It's a combination of the `NOT NULL` and `UNIQUE` constraints. This means the value in a primary key column must be unique for every row and cannot be empty.
- **Best Practice:** It's almost always used on an `INT UNSIGNED` column with `AUTO_INCREMENT` to automatically generate a

unique ID for each new record.

SQL

```
CREATE TABLE tb01 (
   f01 INT UNSIGNED AUTO_INCREMENT PRIMARY KEY, -- Employe
e ID
   f02 VARCHAR(100) NOT NULL
);
```

`FOREIGN KEY` Constraint

This constraint creates a link between two tables and is the foundation of a **relational** database.

- **What it does:** It ensures that a value entered into a column (the foreign key) in one table must already exist in the `PRIMARY KEY` of another table. This prevents "orphan" records. For example, you can't assign an employee to a department that doesn't exist.
- **Key Feature:** You can define actions for what happens if the referenced key is deleted or updated using `ON DELETE` and `ON UPDATE` clauses (`CASCADE`, `SET NULL`, `RESTRICT`).

```
CREATE TABLE tb01 (
   f01 INT UNSIGNED AUTO_INCREMENT PRIMARY KEY, -- Employe
e ID
   f02 VARCHAR(100) NOT NULL,
   f03 INT UNSIGNED, -- Department ID

   CONSTRAINT fk_department
   FOREIGN KEY (f03) REFERENCES tb02(f01)
);
```

`NOT NULL` Constraint

This is a straightforward but essential rule that ensures a column must always have a value.

- **What it does:** It prevents you from inserting or updating a row if you leave this column empty (`NULL`).
- **Use Case:** Apply this to any field that is essential for a record to make sense, such as a person's name, an email address, or a transaction amount.

```
CREATE TABLE tb01 (
   f01 INT PRIMARY KEY,
   f02 VARCHAR(100) NOT NULL -- Employee name cannot be empt
y
);
```

`UNIQUE` Constraint

This constraint ensures that every value in a column (or a combination of columns) is unique across all rows.

- **What it does:** It's similar to a `PRIMARY KEY`, but with two key differences: a table can have **multiple** `UNIQUE` constraints, and a `UNIQUE` column **can** accept `NULL` values.
- **Use Case:** Perfect for columns that must be unique but aren't the primary identifier, such as an email address, a user's phone number, or a product's serial number.

```
CREATE TABLE tb01 (
   f01 INT PRIMARY KEY,
   f02 VARCHAR(100) NOT NULL,
   f03 VARCHAR(255) UNIQUE
);
```

`CHECK` Constraint

This is a powerful constraint that allows you to enforce custom business rules directly in the database.

- **What it does:** It verifies that a value meets a specific logical condition before it's allowed into the column.
- **Use Case:** Enforcing a minimum salary (`salary > 0`), ensuring an age is over 18, or making sure a discount percentage is between 0 and 100.

```
CREATE TABLE tb01 (
    f01 INT PRIMARY KEY,
    f02 DECIMAL(10, 2) NOT NULL CHECK (f02 > 0), -- Salary must b
e positive
    f03 INT CHECK (f03 >= 18) -- Age must be 18 or greater
);
```

`DEFAULT` Constraint

This constraint provides a default value for a column when an `INSERT` statement does not specify a value.

- **What it does:** It automatically fills a column with a predefined value if one isn't provided.
- **Use Case:** Setting a default status for new records (e.g., `'pending'`), assigning a `join_date` to the current date, or setting a user's role to `'guest'`.

```
CREATE TABLE tb01 (
    f01 INT PRIMARY KEY,
    f02 VARCHAR(50) DEFAULT 'active', -- Status will be 'active' if not
specified
    f03 TIMESTAMP DEFAULT CURRENT_TIMESTAMP -- Record creati
on time
);
```

4. DROP TABLE

```
DROP TABLE table_name;
```

5. ALTER TABLE

Used to modify an existing table.

Add Column:

```
ALTER TABLE tb01 ADD COLUMN f06 VARCHAR(50);
```

Modify Column:

```
ALTER TABLE tb01 MODIFY COLUMN f03 DECIMAL(12,2);
```

Rename Column :

```
ALTER TABLE tb01 RENAME COLUMN f05 TO f07;
```

Drop Column:

```
ALTER TABLE tb01 DROP COLUMN f07;
```

Add Primary Key:

```
ALTER TABLE tb01 ADD PRIMARY KEY (f01);
```

Add Foreign Key:

```
ALTER TABLE tb01 ADD CONSTRAINT fk_dept FOREIGN KEY (f02) REFE
RENCES tb02(f08);
```

6. TRUNCATE TABLE

```
TRUNCATE TABLE table_name;
```

Edge Cases:

- Unlike `DELETE FROM Employees`, **TRUNCATE is faster but cannot be rolled back** if not in transaction.
- Using TRUNCATE when there are FOREIGN KEY constraints → ERROR
- Must drop foreign key constraint first or use `DELETE`.

7. RENAME TABLE

```
RENAME TABLE old_name TO new_name;
```

▼ DML

DML stands for **Data Manipulation Language**.

It is used to **manipulate data inside tables**:

- Insert new records
- Update existing data
- Delete data

1. INSERT

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

Example:

```
INSERT INTO tb01(f01, f02, f03, f04)
VALUES ('Nisharg Soni', 'ncsoni04@gmail.com', '2025-09-09', 55000.00);
```

2. UPDATE

```
UPDATE table_name SET column1 = value1, column2 = value2 WHERE c
ondition;
```

Example:

```
UPDATE tb01 SET f04 = 60000 WHERE name = 'Nisharg Soni';
```

Update all rows (Dangerous!):

```
UPDATE tb01 SET f05 = f05 * 1.1;
```

Always double-check the WHERE clause.

- Using LIMIT (MySQL-specific feature):

  ```
  UPDATE tb01 SET f04 = 50000 WHERE f04 < 40000 LIMIT 5;
  ```

  → Updates only first 5 matched rows.

3. DELETE

Used to remove records from a table.

```
DELETE FROM table_name WHERE condition;
```

▼ DQL

DQL stands for **Data Query Language**.

It’s primary purpose is to **retrieve data from database tables** using the ***SELECT*** statement and apply filtering, sorting, grouping, and conditions.

1. SELECT

```
SELECT
	column1,
	column2
FROM
	table_name;
```

Example:

```
SELECT
	f01,
	f02,
	f03
FROM
	Employees;
```

Select All Columns (Not In Production):

```
SELECT
	*
FROM
	Employees;
```

The Logical Processing Order:

1. `FROM` / `JOIN`: Gets all the data from the specified tables and joins them.
2. `WHERE`: Filters these rows based on your conditions.
3. `GROUP BY`: Aggregates the filtered rows into groups.
4. `HAVING`: Filters the *groups* created by `GROUP BY`.
5. `SELECT`: Selects the final columns to be displayed.
6. `DISTINCT`: Removes duplicate rows from the result.
7. `ORDER BY`: Sorts the final result set.
8. `LIMIT`: Restricts the number of rows returned.

2. WHERE Clause

Used to filter records.

```
SELECT
	column1
FROM
	table_name
WHERE
	condition;
```

Examples:

```
SELECT
	*
FROM
	tb01
WHERE
	f04 > 50000;


SELECT
	*
FROM
	tb01
WHERE
	f03 BETWEEN '2025-09-09' AND '2026-09-09';


SELECT
	*
FROM
	tb02
WHERE
	f02 LIKE 'N%';


SELECT
	*
```

```
FROM
        tb02
WHERE
        f04 IS NULL;



SELECT
        *
FROM
        tb02
WHERE
        f04 > 50000 AND f03 < '2023-01-01';
```

3. ORDER BY

Sorts the result set by one or more columns.

```
SELECT
        column1
FROM
        table_name
ORDER BY
        column1 ASC|DESC;
```

Examples:

```
SELECT
        f01,
        f02
FROM
        tb01
ORDER BY
        f02 DESC;


SELECT
        f01,
```

```
        f03
FROM
        tb02
ORDER BY
        f03 ASC;


SELECT
        f01,
        f02,
        f03
FROM
        tb01
ORDER BY
        f02 DESC, f03 ASC;
```

4. GROUP BY

Groups rows sharing a common attribute and allows aggregate functions.

```
SELECT
        column1,
        AGG_FUNC(column2)
FROM
        table_name
GROUP BY
        column1;
```

Examples:

```
SELECT
        f05,
        COUNT(*) AS emp_count
FROM
        tb01
GROUP BY
        f05;
```

```
SELECT
	f05,
	AVG(f02) AS avg_salary
FROM
	tb01
GROUP BY
	f05;
```

5. HAVING

Works like WHERE but applies conditions on aggregated data.

```
SELECT
	column1,
	AGG_FUNC(column2)
FROM
	table_name
GROUP BY
	column1
HAVING
	condition;
```

Examples:

```
SELECT
	f05,
	AVG(f02) AS avg_salary
FROM
	tb01
GROUP BY
	f05
HAVING
	AVG(f02) > 50000;


SELECT
```

```
        f05,
        COUNT(*) AS emp_count
FROM
        tb01
GROUP BY
        f05
HAVING
        emp_count >= 3;
```

▼ DCL

Core Concept: Security Guard of the Database

DCL is all about **security and access control**. It determines who can log in and what they are allowed to do. The main goal is to enforce the **Principle of Least Privilege**: users should only have the exact permissions they need to do their job, and nothing more.

The Three Key Commands

1. `CREATE USER`: Create a New Account

This is the first step. You create an account before you can assign permissions to it.

- Syntax:

  ```
  CREATE USER 'username'@'hostname' IDENTIFIED BY
  'password';
  ```

- **Key Parts**:
  - `'username'`: The name of the user.
  - `'hostname'`: Where the user can connect from.
    - `'localhost'`: Only from the same machine.
    - `'%'`: From any machine (a wildcard).
    - `'192.168.1.100'`: From a specific IP address.

- **Example**:

```
CREATE USER 'web_app'@'%' IDENTIFIED BY 'P@ssw0rd123!';
```

This creates a user named `web_app` that can connect from anywhere.

2. `GRANT`: Give Permissions

This command gives a user specific abilities.

- Syntax:

  ```
  GRANT [privilege(s)] ON [database.object] TO 'user'@'host';
  ```

- **Privileges (What they can do)**:
  - **Common DML**: `SELECT`, `INSERT`, `UPDATE`, `DELETE`
  - **Common DDL**: `CREATE`, `ALTER`, `DROP`
  - **Powerful**: `ALL PRIVILEGES` (grants all permissions)
- **Scope (Where they can do it)**:
  - `.*`: Global level (all databases). **Use with extreme caution.**
  - `database_name.*`: Database level (all tables in one database). **Most common.**
  - `database_name.table_name`: Table level (one specific table).
- **Example**:

```
GRANT SELECT, INSERT, UPDATE ON company_hr.* TO 'web_ap
p'@'%';
```

3. `REVOKE`: Take Permissions Away

This is the opposite of `GRANT`. It removes permissions from a user.

- Syntax:

```
REVOKE [privilege(s)] ON [database.object] FROM
'user'@'host';
```

- **Example**:SQL

```
REVOKE DELETE ON company_hr.* FROM 'web_app'@'%';
```

Special Keywords and Auditing

- `WITH GRANT OPTION`: This is a powerful addition to a `GRANT` statement. It allows the user you are granting permissions to, to then grant *their own permissions* to other users. This is how we can delegate administrative power.

```
GRANT ALL PRIVILEGES ON company_hr.* TO 'db_admin'@'localhos
t' WITH GRANT OPTION;
```

- `SHOW GRANTS`: This command is your best friend for security audits. It tells you exactly what permissions a user has.

```
SHOW GRANTS FOR 'web_app'@'%';
```

- `DROP USER`: This command completely deletes a user account.

```
DROP USER 'web_app'@'%';
```

▼ TCL

What is TCL?

Transaction Control Language (TCL) is a set of SQL commands used to manage transactions in a database. A **transaction** is a sequence of operations performed as a single logical unit of work. TCL ensures that your data remains consistent and integral by following an "all or nothing" rule.

Think of a bank transfer: debiting money from one account and crediting another must either both succeed or both fail. A transaction wraps these two operations into one atomic unit.

The ACID Properties

All transactions must adhere to four core principles known as ACID to guarantee reliability:

- **Atomicity**: The entire transaction is treated as a single, indivisible unit. It either completes fully ("all") or is completely undone ("nothing").
- **Consistency**: A transaction moves the database from one valid state to another. It never leaves the database in a half-finished, corrupt state.
- **Isolation**: Concurrent transactions do not interfere with each other. It's as if each transaction is running in its own isolated environment.
- **Durability**: Once a transaction is successfully saved (**committed**), the changes are permanent and will survive any system failure, like a power outage.

The TCL Commands

Prerequisite: TCL commands only work with transaction-safe storage engines like **InnoDB** (the default in modern MySQL). They do **not** work with older engines like **MyISAM**.

`START TRANSACTION`

This command explicitly begins a transaction block. Any commands that follow are part of this transaction until it is ended.

`COMMIT`

This command saves all the operations within the transaction, making the changes permanent in the database. A `COMMIT` marks the successful end of a transaction.

ROLLBACK

This command discards all the operations within the transaction, reverting the database to the state it was in before the transaction began. A `ROLLBACK` marks the unsuccessful end of a transaction.

SAVEPOINT

This command sets a named marker within a transaction. It allows you to perform a partial rollback to that specific point without undoing the entire transaction. You can then use `ROLLBACK TO SAVEPOINT savepoint_name;`.

Example:

Imagine processing a product order. You need to decrease the product's stock and record the sale.

```
-- Let's say we have 10 units of 'Product A' in stock.
-- A customer orders 2 units.
-- 1. Begin the transaction.
START TRANSACTION;
-- 2. Decrease the stock quantity.
UPDATE products
SET quantity_in_stock = quantity_in_stock - 2
WHERE product_id = 101;
-- At this point, the change is only visible within our current session.
-- If another user queries the stock, they will still see 10.
-- 3. Record the sale in the sales table.
INSERT INTO sales (product_id, quantity_sold)
VALUES (101, 2);
-- Now, we check if everything was successful.
-- If both statements executed without error, we make the changes per
manent.
COMMIT;
-- After COMMIT, all other users will see the stock as 8.
-- What if the INSERT statement had failed for some reason?
-- We would discard the changes to ensure data consistency.
-- ROLLBACK;
-- After ROLLBACK, the stock would be reverted to 10.
```

Autocommit Mode

By default, MySQL runs in **autocommit mode** (`autocommit = 1`). This means every single SQL statement (`INSERT`, `UPDATE`, `DELETE`) is treated as its own transaction and is **committed immediately** upon execution.

- **Why does it exist?** It's convenient for simple, single-line commands where you don't need to group operations.
- **How does `START TRANSACTION` work with it?** The `START TRANSACTION` command **temporarily pauses autocommit mode** for the duration of the transaction block. Once you `COMMIT` or `ROLLBACK`, the session automatically returns to autocommit mode. This is why you don't need to manually disable it for multi-statement transactions.

▼ JOINs

In relational databases, **JOINs** combine rows from two or more tables based on related columns (usually using foreign keys).

Used when you want to query data from multiple tables at once.

Table 1 – tb01 (Employees)

f01	f02	f03
1	Nisharg Soni	101
2	Dakshil Gorasiya	102
3	Hemang Patel	103
4	Manish Jadav	NULL

Table 2 – tb02 (Departments)

f03	f04
101	HR
102	Engineering
104	Sales

1. INNER JOIN

Returns rows where there is a **match in both tables**.

```
SELECT
	e.f02,
	d.f04
FROM
	tb01 e
	INNER JOIN
	tb02 d
	ON
	e.f03 = d.f03;
```

Result:

f02	f04
Nisharg Soni	HR
Dakshil Gorasiya	Engineering

2. LEFT JOIN (or LEFT OUTER JOIN)

Returns **all rows from the left table (Employees)**, and matching rows from the right table (Departments).

If no match → NULL in columns from right table.

```
SELECT
	e.f02,
	d.f04
FROM
	tb01 e
	LEFT JOIN
	tb02 d
	ON
	e.f03 = d.f03;
```

Result:

f02	f04
Nisharg Soni	HR
Dakshil Gorasiya	Engineering
Hemang Patel	NULL
Manish Jadav	NULL

3. RIGHT JOIN (or RIGHT OUTER JOIN)

Returns **all rows from the right table (Departments)**, and matching rows from the left table (Employees).

If no match → NULL in columns from left table.

```
SELECT
	e.f02,
	d.f04
FROM
	tb01 e
	RIGHT JOIN
	tb02 d
	ON
	e.f03 = d.f03 ;
```

Result:

f02	f04
Nisharg Soni	HR
Dakshil Gorasiya	Engineering
NULL	Sales

4. FULL OUTER JOIN

MySQL doesn’t support FULL OUTER JOIN natively.

But you can simulate it using `UNION`.

```
SELECT
	e.f02,
	d.f04
```

```
FROM
	tb01 e
	LEFT JOIN
	tb02 d
	ON
	e.f03 = d.f03

UNION

SELECT
	e.f02,
	d.f04
FROM
	tb01 e
	RIGHT JOIN
	tb02 d
	ON
	e.f03 = d.f03;
```

Result:

f02	f04
Nisharg Soni	HR
Dakshil Gorasiya	Engineering
Hemang Patel	NULL
Manish Jadav	NULL
NULL	Sales

5. CROSS JOIN

Returns **Cartesian product of both tables** (all combinations).

Syntax:

```
SELECT
	e.f02,
	d.f04
FROM
```

```
        tb01 e
        CROSS JOIN
        tb02 d;
```

Result:

f01	f04
Nisharg Soni	HR
Nisharg Soni	Engineering
Nisharg Soni	Sales
Dakshil Gorasiya	HR
Dakshil Gorasiya	Engineering
Dakshil Gorasiya	Sales
Hemang Patel	HR
Hemang Patel	Engineering
Hemang Patel	Sales
Manish Jadav	HR
Manish Jadav	Engineering
Manish Jadav	Sales

6. SELF JOIN

Joining a table to itself (useful for hierarchical data).

Example:

Employees table has a f06 (manager_id) column referring to f05 (emp_id) (self-referencing).

```
SELECT
        e1.f01 AS Employee,
        e2.f01 AS Manager
FROM
        tb01 e1
        LEFT JOIN
        tb01 e2
```

```
        ON
        e1.f06 = e2.f05;
```

▼ AGGREGATE FUNCTONS

Aggregate functions are used to **compute a single value from multiple rows of data**.

They are commonly used in combination with ***GROUP BY*** but can also be used alone.

1. SUM() – Total Sum of Values

```
SELECT
        SUM(column_name)
FROM
        table_name
WHERE
        condition;
```

Example:

Get total salary of all employees:

```
SELECT
        SUM(f04) AS total_salary
FROM
        tb01;
```

If matching rows Returns `NULL`.

Use `COALESCE()` to avoid NULL:

```
SELECT
        COALESCE(SUM(f04), 0) AS total_salary
FROM
        tb01
```

```
WHERE
        f04 > 1e6;
```

2. COUNT() – Count Number of Rows

```
SELECT
        COUNT(column_name)
FROM
        table_name
WHERE
        condition;
```

Examples:

- Count total employees:

```
SELECT
        COUNT(*) AS total_employees
FROM
        tb01;
```

- Count employees in a specific department:

```
SELECT
        COUNT(f01) AS engineering_count
FROM
        tb01
WHERE
        f03 = 101;
```

Edge Cases:

- `COUNT(column_name)` ignores NULL values in the column.

  Example:

```
SELECT
	COUNT(f03)
FROM
	tb01;
```

→ Does NOT count rows where `f03 IS NULL`.

- Use `COUNT(*)` to count all rows, regardless of NULLs.

3. AVG() – Average Value

```
SELECT
	AVG(column_name)
FROM
	table_name
WHERE
	condition;
```

Example:

Average salary of all employees:

```
SELECT
	AVG(f02) AS avg_salary
FROM
	tb01;
```

Edge Cases:

- AVG ignores NULL values by default.
- If no rows → Returns NULL.

Use `COALESCE()` for default fallback:

```
SELECT
	COALESCE(AVG(f02), 0) AS avg_salary
FROM
```

```
        tb01
WHERE
        f03 = 999;
```

4. MIN() – Minimum Value

```
SELECT
        MIN(column_name)
FROM
        table_name
WHERE
        condition;
```

Example:

Find the smallest salary:

```
SELECT
        MIN(f02) AS min_salary
FROM
        tb01;
```

5. MAX() – Maximum Value

```
SELECT
        MAX(column_name)
FROM
        table_name
WHERE
        condition;
```

Example:

Find the highest salary:

```
SELECT
        MAX(f02) AS max_salary
FROM
        tb01;
```

▼ FUNCTIONS

String Functions

- `CONCAT()`: Joins strings together.
  - *Example*: `CONCAT(first_name, ' ', last_name)`
- `UPPER()` / `LOWER()`: Changes the case of a string.
  - *Example*: `UPPER('hello')` returns `'HELLO'`.
- `SUBSTRING()`: Extracts a piece of a string.
  - *Example*: `SUBSTRING('MySQL', 1, 3)` returns `'MyS'`.
- `CHAR_LENGTH()`: Gets the number of characters in a string.
  - *Example*: `CHAR_LENGTH('hello')` returns `5`.
- `REPLACE()`: Finds and replaces a part of a string.
  - *Example*: `REPLACE('Hello World', 'World', 'MySQL')` returns `'Hello MySQL'`.

Numeric Functions

- `ROUND()`: Rounds a number to a specified decimal place.
  - *Example*: `ROUND(12.345, 2)` returns `12.35`.
- `CEIL()`: Rounds a number **up** to the nearest integer.
  - *Example*: `CEIL(7.1)` returns `8`.
- `FLOOR()`: Rounds a number **down** to the nearest integer.
  - *Example*: `FLOOR(7.9)` returns `7`.
- `RAND()`: Generates a random number between 0 and 1.
  - *Example*: `ORDER BY RAND() LIMIT 1` can fetch a random row.

Date & Time Functions

- `NOW()`: Gets the current date and time.
- `CURDATE()`: Gets the current date only.
- `DATEDIFF()`: Calculates the number of days between two dates.
  - *Example*: `DATEDIFF('2025-10-01', '2025-09-25')` returns `6`.
- `DATE_ADD()`: Adds a time interval to a date.
  - *Example*: `DATE_ADD(CURDATE(), INTERVAL 1 MONTH)`.
- `DATE_FORMAT()`: Formats a date into a specific string.
  - *Example*: `DATE_FORMAT(CURDATE(), '%d-%b-%Y')` returns `'11-Sep-2025'`.

Control Flow & Conditionals

- `IF()`: A simple `if-then-else` statement.
  - *Example*: `IF(salary > 70000, 'High', 'Standard')`.
- `COALESCE()`: Returns the first non-`NULL` value from a list.
  - *Example*: `COALESCE(NULL, department_name, 'Unassigned')`.
- `CASE`: Handles multiple `if-then-else` conditions.
  - *Example*: `CASE WHEN salary > 80000 THEN 'Tier 1' ELSE 'Tier 2' END`.

System & Information Functions

- `LAST_INSERT_ID()`: **Extremely useful**. Gets the `AUTO_INCREMENT` ID of the last row you just inserted.
- `DATABASE()`: Returns the name of the current database.
- `USER()`: Returns the name of the current database user.