Week 1

What is Database?

- An organized collection of data
- A method to manipulate and access the data
- Database can be the folder wherein the tables are the files in it

Overview of Workbench

MySQL Workbench is a GUI tool for working with MySQL databases. Its main purpose is to create database design, development, administration, and maintenance.

- **Main Features:**
    - Visual database design and modeling (ER diagrams).
    - SQL editor to write, run, and debug queries.
    - Database administration (user management, backup, recovery).
    - Performance monitoring and server configuration.

- **Advantages**:
    - Easy to use, beginner-friendly interface.
    - Combines multiple tools (design, query, admin) in one place.
    - Saves time with visual design and query building.

- **Uses**: Creating databases, writing SQL queries, managing users and permissions, and analyzing database performance.

DDL Commands

Data Definition Commands are used to define or change the structure of database objects

Key DDL Commands:

- **CREATE**: Used to create new database **objects** like tables, databases, views, indexes.

```
-- Create a new database
CREATE DATABASE myDB;

-- create a table
CREATE TABLE employees (
    employee_id INT,
  first_name VARCHAR(50),
  last_name VARCHAR(50),
  hourly_pay DECIMAL(5,2),
  hire_date DATE
);
```

- **ALTER:** modifies the structure of the existing table.

```
-- this sets the database to read onlu and hence we cannot modify or edit
ALTER DATABASE myDB READ ONLY = 1;

-- This adds a column to the already existing table
ALTER TABLE employees ADD email VARCHAR(50);

-- This will remove the column from the already existing table
ALTER TABLE employees DROP COLUMN email;

-- this will move the email column after last_name
ALTER TABLE employees
MODIFY email VARCHAR(100)
AFTER last_name;

-- this will move the email column at first
```

```
MODIFY email VARCHAR(100)
FIRST;
```

- **DROP:** This command deletes the database completely (including all its tables and data). Be careful – once dropped, it **cannot be recovered**.

```
-- drops database
DROP DATABASE myDB;

-- drop tables
DROP TABLE employees;
```

- **TRUNCATE:** Remove all rows from a table but keep its structure. Faster than DELETE because it **doesn't log individual row deletions**. Table structure, columns, and indexes remain intact.

DIFFERENCE BETWEEN DROP AND TRUNCATE :

Feature	DROP TABLE	TRUNCATE TABLE
Action	Deletes table **structure + data**	Deletes **only data**, keeps structure
Rollback	Cannot be rolled back (permanent)	Usually cannot be rolled back (auto-committed)
Speed	Slower for large tables	Faster, no row-by-row logging
Use	Remove table completely	Clear all data but reuse table

- RENAME: change the **name of a database object** like a table.

```
RENAME TABLE employees TO workers;
```

DML Commands

Data Manipulation Commands are used to **manipulate data** inside database tables.

- **INSERT:** adds new rows to a table

```
INSERT INTO employees VALUES
(1, "Eugene", "Krabs" , 25.50, "2023-01-02"),
(2, "Squidward", "Tentacles", 15.00, "2023-01-03"),
(3, "Spongebob", "Squarepants", 12.50, "2023-01-04"),
(4, "Patrik", "Star", 12.50, "2023-01-05"),
(5, "Sandy", "Cheeks", 17.25, "2023-01-06");

INSERT INTO employees(employee_id, first_name, last_name) VALUES
(6, "Sheldon", "Plankton" );
```

- **UPDATE:** modify existing data

```
UPDATE employees SET last_name='Jr' WHERE id=6;
```

always use the WHERE clause or else it will update all the rows with last_name "Jr".

- **DELETE:** removes rows from a table

```
-- delete the row with employee_id 1
DELETE FROM employees
WHERE employee_id = 1;
```

If you forget the WHERE clause,

```
DELETE FROM employees;
```

Deletes all data but keeps the table structure.

DQL Commands

Data Query Commands are used to **query or retrieve data** from database tables. DQL **does not modify data**, only reads it. They can be combined with clauses like WHERE, ORDER BY, GROUP BY, JOIN.

- **SELECT:** Fetches data from one or more tables.

```
SELECT * FROM employees;

SELECT first_name, last_name FROM employees;

SELECT * FROM employees
WHERE employee_id = 1;

SELECT * FROM employees
WHERE first_name = "Spongebob";

SELECT * FROM employees
WHERE hourly_pay >= 15;

SELECT * FROM employees
WHERE hire_date IS NULL;

SELECT * FROM employees
WHERE hire_date IS NOT NULL;
```

LIMIT, ORDER BY, GROUP BY, HAVING

- **LIMIT:** Restrict the **number of rows returned** by a query.

```
SELECT * FROM students LIMIT 5;
```

Returns only first 5 rows of the result.

```
SELECT * FROM employees LIMIT 10,10;
SELECT * FROM employees LIMIT 20,10;
SELECT * FROM employees LIMIT 30,10;
```

This is called the offset where the 10 records are counted form the offset (i.e 10, 20, 30), this is useful if we want to show 10 results per page.

- **ORDER BY:** used to sort the result set of a query.

  `ASC` → ascending order (default).

  `DESC` → descending order.

  ```
  SELECT * FROM employees
  ORDER BY last_name;

  SELECT * FROM employees
  ORDER BY first_name DESC;
  ```

  We can use ORDER BY using 1 or more columns, the result will depend upon all the columns for the order.

- **AGGREGATE FUNCTIONS & GROUP BY:** Perform calculations on a set of values and return a **single value**.

  Common ones:

  - `COUNT()` → number of rows
  - `SUM()` → total of numeric values
  - `AVG()` → average
  - `MAX()` → maximum value
  - `MIN()` → minimum value

  GROUP BY Clause: Groups rows that have the same value in one or more columns so **aggregate functions** can be applied to each group.

  1. Count students in each age group

  ```
  SELECT age, COUNT(*) FROM students
  GROUP BY age;
  ```

  2. Total salary in each department

  ```
  SELECT department_id, SUM(salary) FROM employees
  GROUP BY department_id;
  ```

3. Average marks per class

```
SELECT class, AVG(marks) FROM results
GROUP BY class;
```

4. Maximum salary in each department

```
SELECT department_id, MAX(salary) FROM employees
GROUP BY department_id;
```

5. Number of orders per customer

```
SELECT customer_id, COUNT(order_id) FROM orders
GROUP BY customer_id;
```

- **HAVING CLAUSE:** It is used to filter groups created by GROUP BY based on a condition. It is used with SELECT + GROUP BY + aggregate functions.

```
-- Only shows departments with more than 5 employees.
SELECT department_id, COUNT(*) AS total_employees
FROM employees
GROUP BY department_id
HAVING COUNT(*) > 5;
```

DIFFERENCE BETWEEN WHERE & HAVING:

Feature	WHERE	HAVING
Purpose	Filters **rows** before grouping	Filters **groups** after grouping
Used with	`SELECT`, `UPDATE`, `DELETE`	`SELECT` with `GROUP BY`
Applies to	Individual rows	Aggregated data (like COUNT, SUM)
Example	`SELECT * FROM employees WHERE age > 30;`	`SELECT department_id, COUNT(*) FROM employees GROUP BY department_id HAVING COUNT(*) > 5;`

JOINS (INNER, LEFT, RIGHT, FULL, SELF)

JOIN is used to combine rows from two or more tables based on a related column between them. It helps to fetch data that is spread across multiple tables.

students table

student_id	name	dept_id
1	Alice	101
2	Bob	102
3	Charlie	103
4	David	101

departments table

dept_id	dept_name
101	Science
102	Math
104	History

- **INNER JOIN:** Returns only the rows where there is a **match in both tables**.

```
SELECT s.name, d.dept_name
FROM students s
INNER JOIN departments d
ON s.dept_id = d.dept_id;
```

name	dept_name
Alice	Science
Bob	Math
David	Science

- **LEFT JOIN:** Returns **all rows from the left table** (students), and matching rows from the right table (departments). If no match, **NULL** appears.

```
SELECT s.name, d.dept_name
FROM students s
LEFT JOIN departments d
ON s.dept_id = d.dept_id;
```

name	dept_name
Alice	Science
Bob	Math
Charlie	NULL
David	Science

- RIGHT JOIN: Returns **all rows from the right table** (departments) and matching rows from the left (students). Missing left-side rows appear as **NULL**.

```
SELECT s.name, d.dept_name
FROM students s
RIGHT JOIN departments d
ON s.dept_id = d.dept_id;
```

name	dept_name
Alice	Science
David	Science
Bob	Math
NULL	History

- FULL JOIN: Returns **all rows from both tables**. Missing values appear as **NULL**. MySQL does not support FULL JOIN directly; can be done with `UNION` of LEFT and RIGHT joins.

```
SELECT s.name, d.dept_name
FROM students s
LEFT JOIN departments d ON s.dept_id = d.dept_id
UNION
SELECT s.name, d.dept_name
FROM students s
RIGHT JOIN departments d ON s.dept_id = d.dept_id;
```

name	dept_name
Alice	Science
David	Science

name	dept_name
Bob	Math
Charlie	NULL
NULL	History

- SELF JOIN: Join a table **with itself**. Useful for comparing rows in the same table.

```
SELECT a.name AS student1, b.name AS student2, a.dept_id
FROM students a
JOIN students b
ON a.dept_id = b.dept_id AND a.student_id < b.student_id;
```

student1	student2	dept_id
Alice	David	101

Analogy

- Think of **tables as groups of people**:
  - **INNER JOIN** → Only people who are in **both groups**.
  - **LEFT JOIN** → Everyone from **first group**, show info from second if available.
  - **RIGHT JOIN** → Everyone from **second group**, show info from first if available.
  - **FULL JOIN** → Everyone from **both groups**, match where possible.
  - **SELF JOIN** → Comparing people **within the same group**.

Logical Query Processing Order

When SQL executes a `SELECT` query, it **processes clauses in a specific logical order**, not the order you write them:

- **FROM** → Identify tables and join them.
- **WHERE** → Filter rows based on conditions.
- **GROUP BY** → Group rows for aggregation.
- **HAVING** → Filter groups based on aggregate conditions.

- **SELECT** → Choose columns or compute expressions.
- **DISTINCT** → Remove duplicate rows (if used).
- **ORDER BY** → Sort the final result.
- **LIMIT** → Return only a subset of rows (if used).

```
SELECT dept_id, COUNT(*) AS total
FROM employees
WHERE salary > 3000
GROUP BY dept_id
HAVING COUNT(*) > 2
ORDER BY total DESC
LIMIT 3;
```

SQL first picks rows from `employees` (`FROM`), filters salaries > 3000 (`WHERE`), groups by department (`GROUP BY`), filters groups with more than 2 employees (`HAVING`), selects `dept_id` and count (`SELECT`), sorts by count (`ORDER BY`), and returns top 3 (`LIMIT`).

KEYS

Types of Keys

1. **Primary Key**
    - Uniquely identifies each row in a table.
    - Cannot have `NULL` values.
    - Example: `student_id` in a student table.
2. **Foreign Key**
    - Links one table to another.
    - Ensures **referential integrity**.
    - Example: `dept_id` in students referencing `dept_id` in departments.
3. **Unique Key**
    - Ensures **all values are unique** in a column.
    - Can have **one NULL**.

4. **Candidate Key**
    - A column (or set of columns) that **could be chosen as primary key**.
5. **Composite Key**
    - **Combination of two or more columns** to uniquely identify a row.
6. **Super Key**
    - A set of columns that **uniquely identifies a row**, may include extra columns.
7. **Alternate Key**
    - Candidate key that **was not chosen as the primary key**.

TRANSACTIONS - START, COMMIT, ROLLBACK, SAVEPOINT

a group of logically related statements which will either be executed completely or no statement execution will occur. Atomicity, Consistency, Isolation, and Durability (ACID) are the properties of a MySQL transaction. Transactions are only available if the **engine supports them**. Eg, ndb clusters.

- **START TRANSACTION :** starting a transaction. The next statements belong together if power fails, whole thing will rollback
- **COMMIT :** saves permenantly
- ROLLBACK : *Undo everything since the start of the transaction (or since last SAVEPOINT)."*
    - It cancels the changes and restores the previous state.
    - InnoDB can do this because it maintains undo logs.
- **SAVEPOINT:** partial rollback in transaction

ENGINES

Software component that a database management system (DBMS) uses to decide how data is stored, indexed, locked, retrieve, support for transaction, foreign key, locking, crash recovery

Types are transactional and non transactional, transactional provides with rollback and commits modern has transactional, innodb by default also myISAM

Engine examples :

Innodb - banking, e-commerce

Myisam - read heavy apps like reading blog

Memory - temp and fast data like cashing

Archive - historical data like audit

ACID PROPERTIES

Atomicity, Consistency, Isolation, and Durability. These are **rules every transaction should follow** to ensure database reliability, correctness, and safety

Atomicity - do everything or do nothing, eg bank transfers, e-commerce orders and inventory management

Consistency - database must remain in a valid state before and after a transaction.

Isolation - **Isolation** ensures that transactions do not interfere with each other.

problems - **Dirty reads**: Transaction A reads uncommitted changes from Transaction B

eg - ticket booking system

Durability - Once a transaction is committed, the changes are **permanent**, even if the system crashes immediately after. Eg-financial sector

Think of a **banking ATM transaction**:

1. **Atomicity:** Either money is deducted and given to you, or nothing happens.
2. **Consistency:** Your account never goes negative; bank rules are enforced.
3. **Isolation:** Other users don't see your partial transaction until it's finished.
4. **Durability:** Once the cash is dispensed and logged, it is permanent, even if the bank server crashes.

LOCKING

Prevents unsafe simultaneous access to the same data

InnoDB → Row-level locking, transactions supported → safest (ACID compliant).

MyISAM → Table-level locking, no transactions → fast for reads but unsafe for concurrent writes.

LOCKING AND TRANSACTION ENSURE ACID PROPERTIES

Eg : movie ticket booking for 1 seat between 2 people

If not coordinated , deadlocks, dirty reads, lost updates