**International Islamic University Chittagong (IIUC)**

**Department of Computer and Communication Engineering**

|  |  |
| --- | --- |
| Program: **B.sc (Eng.)**  Course Code: **CCE–3506**  Type: **Lab**  Segment: **Lab work** | **Baizid MD Ashadzzaman**  Designation: **Adjunct Lecturer**  Email: **baizid.md.ashadzzaman@gmail.com**  Phone: **8801862420119** |

**Introduction to Databases and SQL :**

A database is a collection of data that is organized and stored in a structured way. Databases are used to store and manage large amounts of data, and are used in a variety of applications such as online stores, financial systems, and customer relationship management systems.

SQL (Structured Query Language) is a programming language that is used to interact with databases. SQL is used to create, modify, and query databases.

**Types of Databases :**

There are several types of databases, including:

|  |  |  |
| --- | --- | --- |
| Relational databases | NoSQL databases | In-memory databases |
| These are the most common type of databases, and they store data in tables with rows and columns. The tables are related to each other through keys, and data can be queried and joined across multiple tables. | These databases are designed to handle large amounts of data that is not structured in a traditional relational way. Examples include MongoDB, Cassandra, and Redis. | These databases store data in memory, which allows for faster access and processing of data. However, the data is lost when the database is shut down. |

**SQL Basics :**

SQL is used to create and modify databases, as well as to query and manipulate data within a database. Here are some basic SQL commands:

|  |  |
| --- | --- |
| **SELECT** | Used to retrieve data from a database. |
| **INSERT INTO** | Used to add new rows to a table. |
| **UPDATE** | Used to modify existing data in a table. |
| **DELETE** | Used to delete rows from a table. |
| **CREATE TABLE** | Used to create a new table in a database. |
| **DROP TABLE** | Used to delete a table from a database. |

**Data Types and Data Definition :**

In a database, data is stored in various formats, known as data types. Some common data types include:

|  |  |
| --- | --- |
| **INTEGER** | A numeric data type that stores whole numbers. |
| **FLOAT** | A numeric data type that stores decimal numbers. |
| **VARCHAR** | A character data type that stores a variable number of characters. |
| **DATE** | A data type that stores date and time values. |

Data definition is the process of creating and modifying the structure of a database. This includes defining the data types and other properties of the data that will be stored in the database.

**Creating a Table :**

To create a table in a database, the CREATE TABLE command is used. For example:

|  |  |
| --- | --- |
| **Create a table users** | CREATE TABLE users   (      id         *INTEGER* PRIMARY KEY,      username   *VARCHAR*(255) NOT NULL,      password   *VARCHAR*(255) NOT NULL,      created\_at *DATE* NOT NULL   ); |
| **Create table with relation** | CREATE TABLE orders   (      order\_id    *INT* PRIMARY KEY,      customer\_id *INT*,      order\_date  *DATE*,      FOREIGN KEY (customer\_id)  REFERENCES customers(id)   ); |
|  | CREATE SEQUENCE student\_infos\_seq START WITH 1  CREATE OR replace TRIGGER student\_infos\_trigger   BEFORE INSERT ON student\_infos FOR EACH ROW BEGIN  :NEW.id := student\_infos\_seq.nextval;END; |

This creates a table called users with four columns: id, username, password, and created\_at. The data type for each column is specified, as well as any additional properties such as PRIMARY KEY or NOT NULL.

**Modifying a Table :**

To modify an existing table, the ALTER TABLE command is used. For example:

|  |  |
| --- | --- |
| **Add new col in table** | ALTER TABLE users   ADD email *VARCHAR*(255); |

This adds a new column called email to the users table.

**Data Constraints :**

Data constraints are rules that are applied to the data in a table to ensure data integrity. Some common data constraints include:

|  |  |
| --- | --- |
| **NOT NULL** | This constraint ensures that a column cannot contain a NULL value. |
| **UNIQUE** | This constraint ensures that all values in a column are unique. |
| **PRIMARY KEY** | This constraint defines a column as the primary key for a table. The primary key is a unique identifier for each row in the table. |
| **FOREIGN KEY** | This constraint defines a column as a foreign key, which is a reference to the primary key of another table. This is used to create a relationship between tables. |

**Creating Indexes**

An index is a data structure that is used to improve the performance of database queries. To create an index on a table, the CREATE INDEX command is used. For example:

|  |  |
| --- | --- |
| **Create index** | CREATE INDEX username\_index   ON users (username); |

This creates an index on the username column of the users table.

**Modifying Indexes**

To modify an existing index, the ALTER INDEX command is used. For example:

|  |  |
| --- | --- |
| **Modify index** | ALTER INDEX username\_index rename TO user\_index; |

This renames the index from username\_index to user\_index.

**Dropping Objects**

To delete an object from a database, the DROP command is used. For example:

|  |  |
| --- | --- |
| **Remove a table** | DROP TABLE users; |

This deletes the users table from the database. The DROP command can also be used to delete indexes and other database objects.

It is important to be careful when using the DROP command, as it permanently deletes the object and the data it contains.

**Data Manipulation**

In a database, data manipulation refers to the process of adding, modifying, and deleting data. This is typically done using SQL commands known as Data Manipulation Language (DML).

**Inserting Data**

To insert new rows into a table, the INSERT INTO command is used. For example:

INSERT INTO users  
            (username,  
             password,  
             created\_at)  
VALUES      ('john',  
             'password123',  
             '2022-01-01');

This inserts a new row into the users table with the values 'john', 'password123', and '2022-01-01' for the username, password, and created\_at columns, respectively.

**Updating Data**

To modify existing data in a table, the UPDATE command is used. For example:

UPDATE users  
SET    password = 'newpassword123'  
WHERE  username = 'john';

This updates the password column for all rows where the username is 'john' to 'newpassword123'.

**Deleting Data**

To delete rows from a table, the DELETE command is used. For example:

DELETE FROM users  
WHERE  username = 'john';

This deletes all rows from the users table where the username is 'john'.

**Selecting Data**

To retrieve data from a table, the SELECT command is used. For example:

SELECT \*  
FROM   users  
WHERE  username = 'john'  
ORDER  BY salary;

This retrieves all columns and rows from the users table where the username is 'john'. The SELECT command can also be used to retrieve specific columns, and to sort and filter the data using clauses such as ORDER BY and WHERE.

**Basic SQL query practice :**

CREATE database database\_name;

RENAME DATABASE old\_database\_name TO new\_database\_name;(mysql)

ALTER DATABASE old\_database\_name RENAME TO new\_database\_name;(oracle)

DROP DATABASE database\_name;

CREATE TABLE customers  
             (  
                          customer\_id *INT* PRIMARY KEY,  
                          name        *VARCHAR*(50),  
                          city        *VARCHAR*(30),  
                          created\_at  *DATE* NOT NULL,  
             );CREATE TABLE orders  
             (  
                          order\_id    *INT* PRIMARY KEY,  
                          customer\_id *INT*,  
                          order\_date  *DATE*,  
                          FOREIGN KEY (customer\_id) REFERENCES customers(customer\_id)  
             );

ALTER TABLE old\_table\_name RENAME TO new\_table\_name;(mysql  
AND  
oracle)

RENAME TABLE old\_table\_name TO new\_table\_name;(mysql)

INSERT INTO customers  
            (  
                        customer\_id,  
                        name,  
                        city  
            )  
            VALUES  
            (  
                        2,  
                        'Jane Smith',  
                        'Los Angeles'  
            )  
            ,  
            (  
                        3,  
                        'Michael Johnson',  
                        'Chicago'  
            );

DROP TABLE table\_name;

SELECT \*  
FROM   table\_name;

SELECT \*  
FROM   customers  
WHERE  city = 'New York';

SELECT \*  
FROM   customers  
WHERE  city = 'New York'  
AND    salary > 50000;

SELECT \*  
FROM   customers  
WHERE  city = 'New York'  
OR     salary > 50000;

SELECT \*  
FROM   customers  
WHERE  city != 'New York';

SELECT \*  
FROM   customers  
WHERE  birth\_date BETWEEN '1980-01-01' AND '1990-12-31';

SELECT \*  
FROM   customers  
WHERE  city IN ('New York',  
                'Los Angeles',  
                'Chicago');

SELECT \*  
FROM   customers  
WHERE  last\_name LIKE 'Smith%';( last name starts WITH)

SELECT \*  
FROM   customers  
WHERE  last\_name LIKE '%son%';( last name contains)

SELECT \*  
FROM   student\_infos  
LIMIT  1; (Mysql)

SELECT \*  
FROM   student\_infos  
WHERE  ROWNUM = 1; (Oracle)

SELECT student\_id,  
       **SUM**(marks) AS total\_marks  
FROM   student\_result\_info  
GROUP  BY student\_id;

DELETE  
FROM   customers  
WHERE  customer\_id = 123;

DELETE  
FROM   table\_name;

TRUNCATE TABLE table\_name;(faster BY restrictive)

**Types of Joins**

There are several types of joins:

* **INNER JOIN:** Returns rows that have matching values in both tables.
* **LEFT JOIN:** Returns all rows from the left table, and the matched rows from the right table.
* **RIGHT JOIN:** Returns all rows from the right table, and the matched rows from the left table.
* **FULL OUTER JOIN:** Returns all rows when there is a match in either left or right table.

Let's say we have two tables:

* **Customers:** customer\_id, first\_name, last\_name, city
* **Orders:** order\_id, customer\_id, order\_date, order\_amount

To find customer names and their corresponding order amounts:

SELECT     c.first\_name,  
           c.last\_name,  
           o.order\_amount  
FROM       customers c  
inner join orders o  
ON         c.customer\_id = o.customer\_id;

-- Find all customers and their orders (if any)

SELECT c.first\_name,  
       c.last\_name,  
       o.order\_amount  
FROM   customers c  
       left join orders o  
              ON c.customer\_id = o.customer\_id;

-- Find all orders and their corresponding customers (if any)

SELECT c.first\_name,  
       c.last\_name,  
       o.order\_amount  
FROM   customers c  
       right join orders o  
               ON c.customer\_id = o.customer\_id;

-- Find all customers and orders, regardless of whether there's a match

SELECT c.first\_name,  
       c.last\_name,  
       o.order\_amount  
FROM   customers c  
       full outer join orders o  
                    ON c.customer\_id = o.customer\_id;

UPDATE customers  
SET    city = 'New City',  
       name = 'New Name'  
WHERE  customer\_id = 123;

INSERT INTO beauty\_saloon\_inventory.employee\_categories  
            (  
                        column1,  
                        column2,  
                        column3,  
                        ...  
            )  
SELECT column1,  
       column2,  
       column3,  
       ...  
FROM   beauty\_saloon\_inventory\_old.employee\_categories;

INSERT INTO beauty\_saloon\_inventory.employee\_categories

SELECT \*  
FROM   beauty\_saloon\_inventory\_old.employee\_categories;INSERT INTO beauty\_saloon\_inventory.products  
            (  
                        column1,  
                        column2,  
                        column3,  
                        new\_column  
            )  
SELECT column1,  
       column2,  
       column3,  
       'default\_value'  
FROM   beauty\_saloon\_inventory\_old.products;

SELECT DISTINCT project  
FROM   employeesalary;  
  
SELECT **Max**(salary) AS MaxSalary,  
       **Min**(salary) AS MinSalary,  
       **Avg**(salary) AS AvgSalary  
FROM   employeesalary;  
  
SELECT \*  
FROM   employeedetails  
WHERE  city = 'California'  
        OR managerid = 321;  
  
SELECT empid,  
       project,  
       salary + variable AS TotalSalary  
FROM   employeesalary;  
  
SELECT \*  
FROM   employeedetails  
WHERE  fullname LIKE '\_\_hn%';  
  
SELECT ord\_no,  
       purch\_amt,  
       ord\_date,  
       customer\_id,  
       salesman\_id  
FROM   orders  
WHERE  salesman\_id = (SELECT salesman\_id  
                      FROM   salesman  
                      WHERE  name = 'Abir');  
  
SELECT name,  
       commission  
FROM   salesman  
WHERE  city = 'Paris';  
  
SELECT grade,  
       **Count**(\*) AS CustomerCount  
FROM   customer  
WHERE  city = 'New York'  
       AND grade > (SELECT **Avg**(grade)  
                    FROM   customer  
                    WHERE  city = 'New York')  
GROUP  BY grade;

**Different joining with example:**

**Employees Table**

| **employee\_id** | **name** | **department\_id** |
| --- | --- | --- |
| 1 | Alice | 101 |
| 2 | Bob | 102 |
| 3 | Charlie | NULL |

**Departments Table**

| **department\_id** | **department\_name** |
| --- | --- |
| 101 | Sales |
| 102 | Marketing |
| 103 | HR |

**Inner join example**

SELECT employees.name,  
       departments.department\_name  
FROM   employees  
       inner join departments  
               ON employees.department\_id = departments.department\_id;

| **name** | **department\_name** |
| --- | --- |
| Alice | Sales |
| Bob | Marketing |

**Left join example**

SELECT employees.name,  
       departments.department\_name  
FROM   employees  
       left join departments  
              ON employees.department\_id = departments.department\_id;

| **name** | **department\_name** |
| --- | --- |
| Alice | Sales |
| Bob | Marketing |
| Charlie | NULL |

**Right join example**

SELECT employees.name,  
       departments.department\_name  
FROM   employees  
       right join departments  
               ON employees.department\_id = departments.department\_id;

| **name** | **department\_name** |
| --- | --- |
| Alice | Sales |
| Bob | Marketing |
| NULL | HR |

**Full join example**

SELECT employees.name,  
       departments.department\_name  
FROM   employees  
       full join departments  
              ON employees.department\_id = departments.department\_id;

| **name** | **department\_name** |
| --- | --- |
| Alice | Sales |
| Bob | Marketing |
| Charlie | NULL |
| NULL | HR |

**Cross join example**

SELECT employees.name,  
       departments.department\_name  
FROM   employees  
       cross join departments;

| **name** | **department\_name** |
| --- | --- |
| Alice | Sales |
| Alice | Marketing |
| Alice | HR |
| Bob | Sales |
| Bob | Marketing |
| Bob | HR |
| Charlie | Sales |
| Charlie | Marketing |
| Charlie | HR |

**JOINING AND RETRIVE DATA:**

CREATE TABLE parents\_infos  
 (  
     id          *NUMBER* PRIMARY KEY,  
     father\_name *VARCHAR2*(50) NULL,  
     mother\_name *VARCHAR2*(50) NULL,  
     student\_id  *NUMBER*,  
     CONSTRAINT fk\_student FOREIGN KEY (student\_id) REFERENCES student\_infos(  
     id) ON DELETE CASCADE  
  );

**After creating table**

ALTER TABLE parents\_infos  
  ADD CONSTRAINT fk\_student FOREIGN KEY (student\_id) REFERENCES  
  student\_infos(id) ON DELETE SET NULL;

**Get data after joining table**

SELECT student\_infos.id             AS student\_id,  
       student\_infos.NAME           AS student\_name,  
       student\_infos.class          AS student\_class,  
       parents\_infos.NAME           AS parent\_name,  
       parents\_infos.contact\_number AS parent\_contact  
FROM   student\_infos  
       JOIN parents\_infos  
         ON student\_infos.id = parents\_infos.student\_id;

**Filter data after joining**

SELECT student\_infos.NAME AS student\_name,  
       parents\_infos.NAME AS parent\_name  
FROM   student\_infos  
       JOIN parents\_infos  
         ON student\_infos.id = parents\_infos.student\_id  
WHERE  student\_infos.NAME = 'Asadzaman';

**For multiple table joining**

SELECT \*  
FROM   student\_infos  
       JOIN parents\_infos  
         ON student\_infos.id = parents\_infos.student\_id  
       JOIN student\_result\_info  
         ON student\_infos.id = student\_result\_info.student\_id;

**NESTED QUERIES:**

Find student who have highest mark

SELECT NAME  
FROM   student\_infos  
WHERE  student\_id = (SELECT student\_id  
                     FROM   student\_result\_info  
                     WHERE  marks = (SELECT *Max*(marks)  
                                     FROM   student\_result\_info));

Find students list who get mark above average

SELECT NAME  
FROM   student\_infos  
WHERE  student\_id IN (SELECT student\_id  
                      FROM   student\_result\_info  
                      WHERE  marks > (SELECT *Avg*(marks)  
                                      FROM   student\_result\_info));

Find subject list of class 10

SELECT DISTINCT subject  
FROM   student\_result\_info  
WHERE  student\_id IN (SELECT student\_id  
                      FROM   student\_infos  
                      WHERE  class = '10');

**Views, Synonyms, and Sequences in Oracle:**

**view** is a virtual table based on the result of a SQL query. It does not store data but fetches it dynamically from the base tables.

**Create view**

CREATE VIEW student\_details  
AS  
  SELECT student\_id,  
         NAME,  
         class  
  FROM   student\_infos  
  WHERE  class = '10';

**Call view**

SELECT \*  
FROM   student\_details;

**Update view**

UPDATE student\_details  
SET    class = '11'  
WHERE  student\_id = 101;

**synonym** is an alias for a database object such as a table, view, sequence, or procedure. It simplifies access to these objects, especially when they are in different schemas.

**Create synonym**

CREATE synonym student\_synonym FOR student\_infos;

**Call using synonym**

SELECT \*  
FROM   student\_synonym;

**sequence** generates unique numbers, often used for primary key values.

**Create sequence**

CREATE SEQUENCE student\_seq  
  START WITH 1  
  INCREMENT BY 1  
  NOCACHE;

**Check sequence**

SELECT student\_seq.nextval  
FROM   dual; 

SELECT student\_seq.currval  
FROM   dual;

**Call sequence**

INSERT INTO student\_infos  
            (student\_id,  
             NAME,  
             class)  
VALUES      (student\_seq.nextval,  
             'Asadzaman',  
             '10');

**Cursors in Database Programming(implicit and explicit cursor):**

Cursors are used in database programming to handle query results row-by-row. There are two types of cursors in PL/SQL: **implicit cursors** and **explicit cursors**.

Enable this first to show result : **SET SERVEROUTPUT ON;**

**implicit cursors**

An implicit cursor is automatically created by Oracle whenever a SELECT, INSERT, UPDATE, or DELETE statement is executed.It is managed internally by Oracle, so you don't need to declare or control it.

DECLARE  
    s\_name *VARCHAR2*(50);  
BEGIN  
    SELECT student\_name  
    INTO   s\_name  
    FROM   student\_infos  
    WHERE  id = 1;  
  
    dbms\_output.**Put\_line**('Student Name: '  
                         || s\_name);  
END;  
/

**explicit cursors**

An explicit cursor is manually declared and controlled by the programmer .It is used when a query returns multiple rows, allowing you to process each row individually.

DECLARE  
    CURSOR student\_cursor IS  
      SELECT id,  
             student\_name  
      FROM   student\_infos;  
    s\_id   *NUMBER*;  
    s\_name *VARCHAR2*(50);  
BEGIN  
    OPEN student\_cursor ; *-- Open the cursor*  
    LOOP  
        FETCH student\_cursor  INTO s\_id, s\_name; *-- Fetch rows*  
  
        EXIT WHEN student\_cursor%NOTFOUND; *-- Exit loop when no more rows*  
        dbms\_output.**Put\_line**('Student ID: '  
                             || s\_id  
                             || ', Name: '  
                             || s\_name);  
    END LOOP;  
  
    CLOSE student\_cursor; *-- Close the cursor*  
END;  
  
/

**Procedures and Functions in PL/SQL:**

Procedures and functions are reusable subprograms in PL/SQL. They allow encapsulating code logic for tasks that can be executed multiple times. However, they have distinct purposes and differences.

**Procedure**

A procedure is a PL/SQL subprogram that performs a specific task. It may or may not return a value, but it can pass multiple values through OUT parameters.

CREATE OR replace PROCEDURE **Calculate\_total\_marks** (inp\_student\_id  IN *NUMBER*,  
                                                   total\_marks OUT *NUMBER*)  
AS  
BEGIN  
    SELECT **SUM**(marks)  
    INTO   total\_marks  
    FROM   student\_result\_info  
    WHERE  student\_id = inp\_student\_id;  
END calculate\_total\_marks;  
  
/

DECLARE  
    total\_marks *NUMBER*;  
BEGIN  
    **Calculate\_total\_marks**(1, total\_marks);  
  
    dbms\_output.**Put\_line**('Total Marks: '  
                         || total\_marks);  
END;

**Function**

A function is a PL/SQL subprogram that must return a single value using the RETURN keyword. Functions are often used in SQL statements.

**Create function**

CREATE OR replace FUNCTION **Calculate\_avg\_marks** (inp\_student\_id IN *NUMBER*)  
RETURN *NUMBER*  
AS  
  avg\_marks *NUMBER*;  
BEGIN  
    SELECT **Avg**(marks)  
    INTO   avg\_marks  
    FROM   student\_result\_info  
    WHERE  student\_id = inp\_student\_id;  
  
    RETURN avg\_marks;  
END calculate\_avg\_marks;  
  
/

**Call function(PL)**

DECLARE  
    avg\_marks *NUMBER*;  
BEGIN  
    avg\_marks := **Calculate\_avg\_marks**(101);  
  
    dbms\_output.**Put\_line**('Average Marks: '  
                         || avg\_marks);  
END;  
  
/

**Call function(SQL)**

SELECT **Calculate\_avg\_marks**(101)  
FROM dual;

**Triggers in PL/SQL:**

A trigger is a stored program in a database that automatically executes (or "fires") in response to certain events on a table or view. It is used to enforce business rules, maintain audit trails, and automate certain database tasks.

This example that keep record of changes of student name in different table

**Student table**

CREATE TABLE student\_infos  
  (  
     student\_id *NUMBER* PRIMARY KEY,  
     name       *VARCHAR2*(50)  
  );

**Student record**

CREATE TABLE student\_audit  
             (  
                          audit\_id    NUMBER generated BY DEFAULT AS identity,  
                          student\_id  NUMBER,  
                          old\_name    VARCHAR2(50),  
                          new\_name    VARCHAR2(50),  
                          change\_date DATE  
             );

**Call trigger**

CREATE OR replace TRIGGER audit\_student\_update  
  AFTER UPDATE ON student\_infos  
  FOR EACH ROW  
BEGIN  
    INSERT INTO student\_audit  
                (student\_id,  
                 old\_name,  
                 new\_name,  
                 change\_date)  
    VALUES      (:OLD.id,  
                 :OLD.student\_name,  
                 :NEW.student\_name,  
                 SYSDATE);  
END;

**Exception Handling in PL/SQL:**

Exception handling in PL/SQL allows you to gracefully manage and respond to runtime errors or unexpected conditions in your program. PL/SQL provides a robust mechanism for handling such errors using the EXCEPTION block.

**Syntax**

DECLARE  
  *-- Declarations (variables, constants, etc.)*  
BEGIN  
  *-- Executable statements*  
EXCEPTION  
WHEN exception\_name THEN  
  *-- Error-handling statements*  
WHEN OTHERS THEN  
  *-- Handle all other exceptions*  
END;  
/

**Divide by Zero**

DECLARE  
    num    *NUMBER* := 10;  
    denom  *NUMBER* := 0;  
    result *NUMBER*;  
BEGIN  
    result := num / denom; *-- This will raise a ZERO\_DIVIDE exception*  
EXCEPTION  
    WHEN zero\_divide THEN  
      dbms\_output.**Put\_line**('Error: Division by zero is not allowed.');  
END;  
  
/

**DATABASE CONNECTIVITY WITH FRONTEND TOOL:**

**Install Flask and MySQL library:**

pip install flask flask-mysql-connector

**Setup database:**

CREATE DATABASE user\_management;  
  
USE user\_management;  
  
CREATE TABLE users  
  (  
     id    *INT* auto\_increment PRIMARY KEY,  
     name  *VARCHAR*(255) NOT NULL,  
     email *VARCHAR*(255) UNIQUE NOT NULL  
  );

**Write python logical code :**

from flask import Flask, render\_template, request, redirect, url\_for, flash

import mysql.connector

app = Flask(\_\_name\_\_)

app.secret\_key = "your\_secret\_key"

# Configure MySQL connection

db\_config = {

    "host": "localhost",

    "user": "root",

    "password": "",

    "database": "user\_management",

}

# Database connection

def get\_db\_connection():

    return mysql.connector.connect(\*\*db\_config)

# Route: Home (Read all users)

@app.route("/")

def index():

    conn = get\_db\_connection()

    cursor = conn.cursor(dictionary=True)

    cursor.execute("SELECT \* FROM users")

    users = cursor.fetchall()

    conn.close()

    return render\_template("index.html", users=users)

# Route: Add User (Create)

@app.route("/add", methods=["POST"])

def add\_user():

    name = request.form["name"]

    email = request.form["email"]

    conn = get\_db\_connection()

    cursor = conn.cursor()

    try:

        cursor.execute("INSERT INTO users (name, email) VALUES (%s, %s)", (name, email))

        conn.commit()

        flash("User added successfully!", "success")

    except Exception as e:

        flash(f"Error: {str(e)}", "danger")

    finally:

        conn.close()

    return redirect(url\_for("index"))

# Route: Edit User (Read specific user)

@app.route("/edit/<int:id>")

def edit\_user(id):

    conn = get\_db\_connection()

    cursor = conn.cursor(dictionary=True)

    cursor.execute("SELECT \* FROM users WHERE id = %s", (id,))

    user = cursor.fetchone()

    conn.close()

    return render\_template("edit.html", user=user)

# Route: Update User

@app.route("/update/<int:id>", methods=["POST"])

def update\_user(id):

    name = request.form["name"]

    email = request.form["email"]

    conn = get\_db\_connection()

    cursor = conn.cursor()

    try:

        cursor.execute(

            "UPDATE users SET name = %s, email = %s WHERE id = %s", (name, email, id)

        )

        conn.commit()

        flash("User updated successfully!", "success")

    except Exception as e:

        flash(f"Error: {str(e)}", "danger")

    finally:

        conn.close()

    return redirect(url\_for("index"))

# Route: Delete User

@app.route("/delete/<int:id>", methods=["POST"])

def delete\_user(id):

    conn = get\_db\_connection()

    cursor = conn.cursor()

    try:

        cursor.execute("DELETE FROM users WHERE id = %s", (id,))

        conn.commit()

        flash("User deleted successfully!", "success")

    except Exception as e:

        flash(f"Error: {str(e)}", "danger")

    finally:

        conn.close()

    return redirect(url\_for("index"))

if \_\_name\_\_ == "\_\_main\_\_":

    app.run(debug=True)

**Write frontend code (html css js):**

Index.html pages

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>User Management</title>

</head>

<body>

    <h1>User Management</h1>

    <form method="POST" action="/add">

        <input type="text" name="name" placeholder="Name" required>

        <input type="email" name="email" placeholder="Email" required>

        <button type="submit">Add User</button>

    </form>

    <table border="1">

        <thead>

            <tr>

                <th>ID</th>

                <th>Name</th>

                <th>Email</th>

                <th>Actions</th>

            </tr>

        </thead>

        <tbody>

            {% for user in users %}

            <tr>

                <td>{{ user.id }}</td>

                <td>{{ user.name }}</td>

                <td>{{ user.email }}</td>

                <td>

                    <a href="/edit/{{ user.id }}">Edit</a>

                    <form method="POST" action="/delete/{{ user.id }}" style="display:inline;">

                        <button type="submit">Delete</button>

                    </form>

                </td>

            </tr>

            {% endfor %}

        </tbody>

    </table>

    {% with messages = get\_flashed\_messages(with\_categories=True) %}

        {% if messages %}

            {% for category, message in messages %}

                <p class="{{ category }}">{{ message }}</p>

            {% endfor %}

        {% endif %}

    {% endwith %}

</body>

</html>

edit.html pages

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Edit User</title>

</head>

<body>

    <h1>Edit User</h1>

    <form method="POST" action="/update/{{ user.id }}">

        <input type="text" name="name" value="{{ user.name }}" required>

        <input type="email" name="email" value="{{ user.email }}" required>

        <button type="submit">Update User</button>

    </form>

</body>

</html>

**Run code:**

python app.py