Advanced Database Systems Design

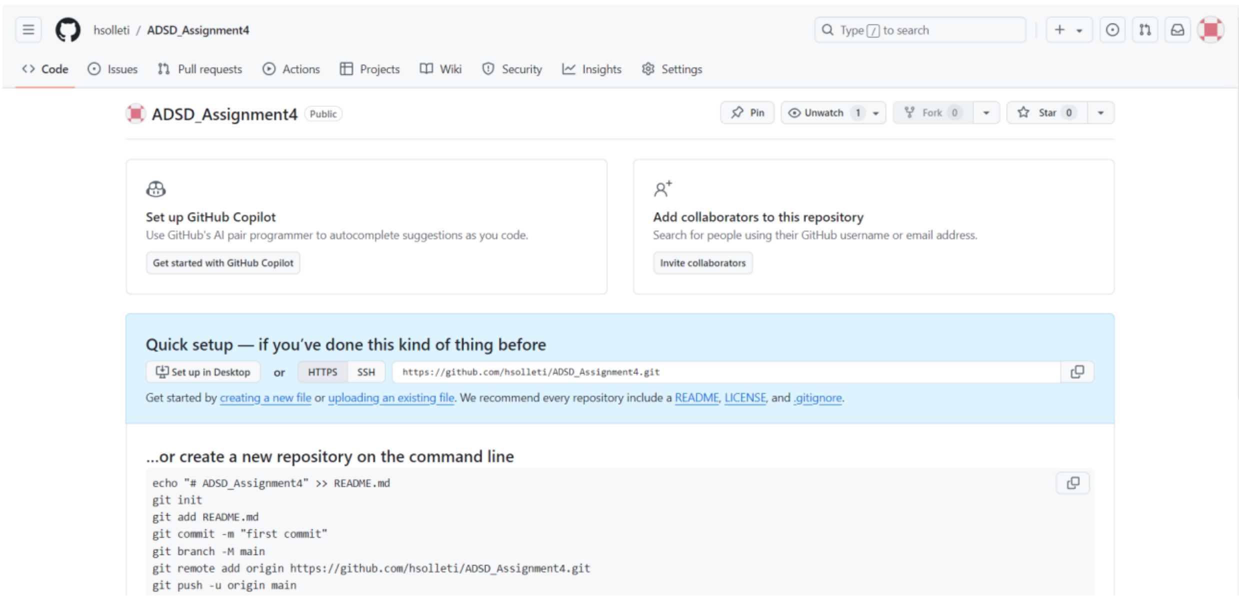
Assignment-4

Harini Padmaja Solleti

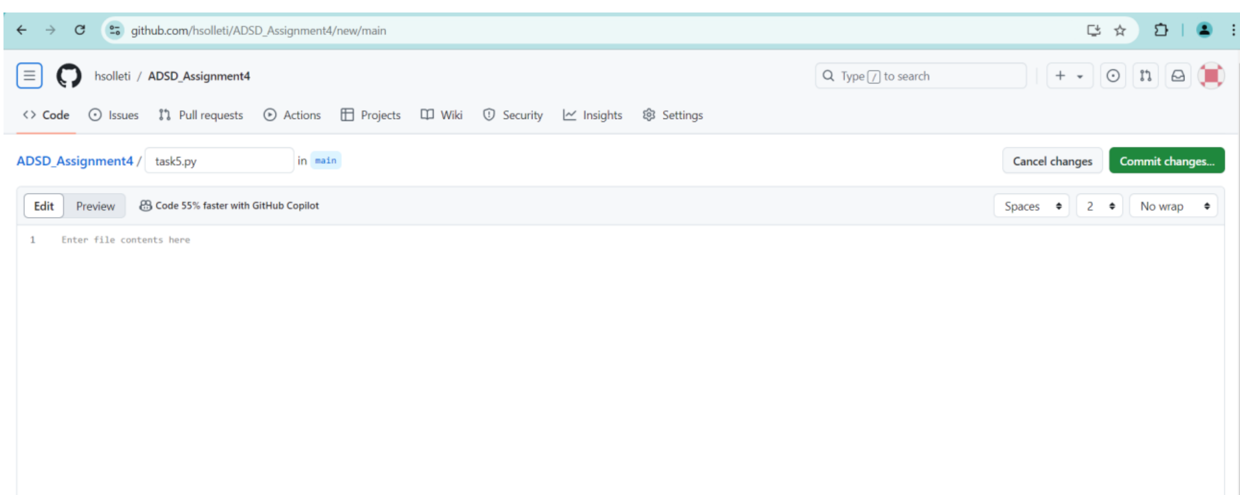
811288992

Firstly, I have created a new repository for this assignment and named it as ADSD_Assignment4.

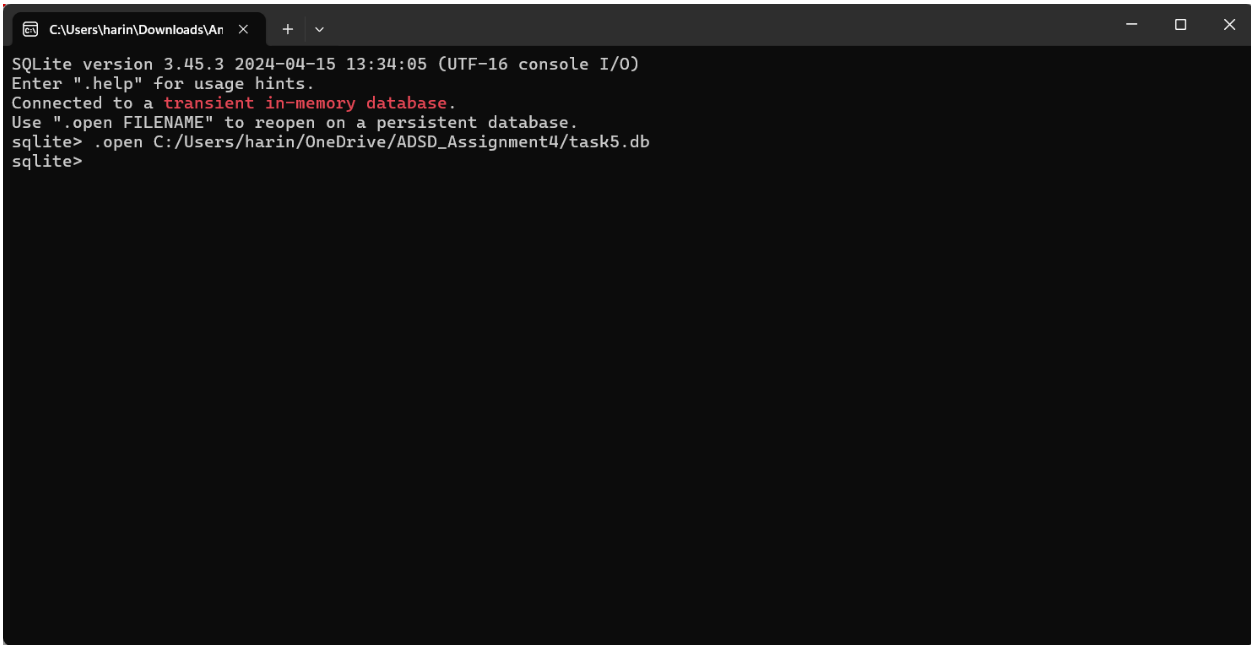
https://github.com/hsolleti/ADSD_Assignment4



I have created a new file and named as task5.py.



Now, I have created a new database named task5.db.



```
C:\Users\harin\Downloads\Ar
SQLite version 3.45.3 2024-04-15 13:34:05 (UTF-16 console I/O)
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> .open C:/Users/harin/OneDrive/ADSD_Assignment4/task5.db
sqlite>
```

(1) SQL Joins:

<u>a) Inner Join:</u>

Here I have created customers table and orders table. I have entered a sample data of names and orders in both the tables. And to retrieve this information I have run the following query and got the desired output.

SELECT customers.customer_name, orders.product_name

FROM customers

INNER JOIN orders

ON customers.customer_id = orders.customer_id;

```
sqlite> CREATE TABLE customers (
(x1...>   customer_id INTEGER PRIMARY KEY,
(x1...>   customer_name TEXT
(x1...> );
sqlite> CREATE TABLE orders (
(x1...>   order_id INTEGER PRIMARY KEY,
(x1...>   customer_id INTEGER,
(x1...>   product_name TEXT,
(x1...>   FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
(x1...> );
sqlite> INSERT INTO customers (customer_name) VALUES ('Alice');
sqlite> INSERT INTO customers (customer_name) VALUES ('Bob');
sqlite> INSERT INTO customers (customer_name) VALUES ('Charlie');
sqlite> INSERT INTO orders (customer_id, product_name) VALUES (1, 'Laptop');
sqlite> INSERT INTO orders (customer_id, product_name) VALUES (2, 'Smartphone');
sqlite> INSERT INTO orders (customer_id, product_name) VALUES (1, 'Tablet');
sqlite> SELECT customers.customer_name, orders.product_name
   ...> FROM customers
   ...> INNER JOIN orders
   ...> ON customers.customer_id = orders.customer_id;
Alice|Laptop
Bob|Smartphone
Alice|Tablet
sqlite>
```

b) Left Join: Left join ensures that even if a customer does not have an order, their name will be still visible with NULL value.

Using the code below I have have done the left join.

SELECT customers.customer_name, orders.product_name
FROM customers
LEFT JOIN orders
ON customers.customer_id = orders.customer_id;

```
sqlite> SELECT customers.customer_name, orders.product_name
   ...> FROM customers
   ...> LEFT JOIN orders
   ...> ON customers.customer_id = orders.customer_id;
Alice|Laptop
Alice|Tablet
Bob|Smartphone
Charlie|
sqlite> |
```

c) Right Join: In sqlite we cannot do the right join, so we must basically swap the left join.

So now, we have created products table, suppliers table, product_supplier table and inserted data in these tables of customers, orders, product and the supplier. I have used the following query and we got the output of which supplier is selling which product.

SELECT products.product_name, suppliers.supplier_name
FROM products
LEFT JOIN product_suppliers
ON products.product_id = product_suppliers.product_id
LEFT JOIN suppliers
ON product_suppliers.supplier_id = suppliers.supplier_id;

```
sqlite> SELECT products.product_name, suppliers.supplier_name
   ...> FROM products
   ...> LEFT JOIN product_suppliers
   ...> ON products.product_id = product_suppliers.product_id
   ...> LEFT JOIN suppliers
   ...> ON product_suppliers.supplier_id = suppliers.supplier_id;
Parse error: no such table: products
sqlite> CREATE TABLE products (
(x1...>   product_id INTEGER PRIMARY KEY,
(x1...>   product_name TEXT
(x1...> );
sqlite> CREATE TABLE suppliers (
(x1...>   supplier_id INTEGER PRIMARY KEY,
(x1...>   supplier_name TEXT
(x1...> );
sqlite> CREATE TABLE product_suppliers (
(x1...>   product_id INTEGER,
(x1...>   supplier_id INTEGER,
(x1...>   FOREIGN KEY (product_id) REFERENCES products(product_id),
(x1...>   FOREIGN KEY (supplier_id) REFERENCES suppliers(supplier_id)
(x1...> );
sqlite> INSERT INTO products (product_name) VALUES ('Laptop');
sqlite> INSERT INTO products (product_name) VALUES ('Tablet');
sqlite> INSERT INTO products (product_name) VALUES ('Smartphone');
sqlite> INSERT INTO suppliers (supplier_name) VALUES ('TechCorp');
sqlite> INSERT INTO suppliers (supplier_name) VALUES ('GadgetCo');
sqlite> INSERT INTO product_suppliers (product_id, supplier_id) VALUES (1, 1);  -- Laptop from TechCorp
sqlite> INSERT INTO product_suppliers (product_id, supplier_id) VALUES (2, 2);  -- Tablet from GadgetCo
sqlite> SELECT products.product_name, suppliers.supplier_name
   ...> FROM products
   ...> LEFT JOIN product_suppliers
   ...> ON products.product_id = product_suppliers.product_id
   ...> LEFT JOIN suppliers
   ...> ON product_suppliers.supplier_id = suppliers.supplier_id;
Laptop|TechCorp
Tablet|GadgetCo
Smartphone|
sqlite> |
```

d) Full Outer Join: This is the union of both the left join and the right join.

I have created employees and department tables, inserted sample data and run the full outer join query and got the output as shown in the below screenshot where we can see that each employee is assigned in particular departments.

```
sqlite> -- First part: Left Join to get employees (even without departments)
sqlite> SELECT employees.employee_name, departments.department_name
   ...> FROM employees
   ...> LEFT JOIN departments
   ...> ON employees.department_id = departments.department_id
   ...>
   ...> UNION
   ...>
   ...> -- Second part: Left Join to get departments (even without employees)
   ...> SELECT employees.employee_name, departments.department_name
   ...> FROM departments
   ...> LEFT JOIN employees
   ...> ON employees.department_id = departments.department_id;
Parse error: no such table: departments
sqlite> CREATE TABLE employees (
(x1...>   employee_id INTEGER PRIMARY KEY,
(x1...>   employee_name TEXT,
(x1...>   department_id INTEGER,
(x1...>   FOREIGN KEY (department_id) REFERENCES departments(department_id)
(x1...> );
sqlite> CREATE TABLE departments (
(x1...>   department_id INTEGER PRIMARY KEY,
(x1...>   department_name TEXT
(x1...> );
sqlite> INSERT INTO departments (department_name) VALUES ('HR');
sqlite> INSERT INTO departments (department_name) VALUES ('Engineering');
sqlite> INSERT INTO departments (department_name) VALUES ('Sales');
sqlite> INSERT INTO employees (employee_name, department_id) VALUES ('Alice', 1);  -- HR
sqlite> INSERT INTO employees (employee_name, department_id) VALUES ('Bob', 2);    -- Engineering
sqlite> INSERT INTO employees (employee_name, department_id) VALUES ('Charlie', NULL);  -- No Department
sqlite> -- First part: Left Join to get employees (even without departments)
sqlite> SELECT employees.employee_name, departments.department_name
   ...> FROM employees
   ...> LEFT JOIN departments
   ...> ON employees.department_id = departments.department_id
   ...>
   ...> UNION
   ...>
   ...> -- Second part: Left Join to get departments (even without employees)
   ...> SELECT employees.employee_name, departments.department_name
   ...> FROM departments
   ...> LEFT JOIN employees
   ...> ON employees.department_id = departments.department_id;
|Sales
Alice|HR
Bob|Engineering
Charlie|
sqlite> |
```

e) Self Join: Self join is a join in which the table is joined with itself. This is useful to compare rows within the same table.

To add manager_id column in place of employee table I have used **ALTER TABLE**. I have dropped the existing employee table and created a new one along with manager_id column. And inserted data into the table and run the following query:

SELECT e.employee_name AS employee, m.employee_name AS manager
FROM employees e
LEFT JOIN employees m
ON e.manager_id = m.employee_id;

```
sqlite> ALTER TABLE employees ADD COLUMN manager_id INTEGER;
sqlite> DROP TABLE employees;
sqlite> CREATE TABLE employees (
(x1...>   employee_id INTEGER PRIMARY KEY,
(x1...>   employee_name TEXT,
(x1...>   manager_id INTEGER,
(x1...>   FOREIGN KEY (manager_id) REFERENCES employees(employee_id)
(x1...> );
sqlite> INSERT INTO employees (employee_name, manager_id) VALUES ('Alice', NULL);   -- Alice has no manager
sqlite> INSERT INTO employees (employee_name, manager_id) VALUES ('Bob', 1);        -- Bob's manager is Alice
sqlite> INSERT INTO employees (employee_name, manager_id) VALUES ('Charlie', 1);    -- Charlie's manager is Alice
sqlite> INSERT INTO employees (employee_name, manager_id) VALUES ('David', 2);      -- David's manager is Bob
sqlite> SELECT e.employee_name AS employee, m.employee_name AS manager
   ...> FROM employees e
   ...> LEFT JOIN employees m
   ...> ON e.manager_id = m.employee_id;
Alice|
Bob|Alice
Charlie|Alice
David|Bob
sqlite>
```

<u>f) Cross Join:</u> Cross join is a cartesian product of two tables.

Here, when we cross join products table and customers table, this will result in combination of every product of every customer.

SELECT products.product_name, customers.customer_name
FROM products
CROSS JOIN customers;

```
sqlite> SELECT products.product_name, customers.customer_name
   ...> FROM products
   ...> CROSS JOIN customers;
Laptop|Alice
Laptop|Bob
Laptop|Charlie
Tablet|Alice
Tablet|Bob
Tablet|Charlie
Smartphone|Alice
Smartphone|Bob
Smartphone|Charlie
sqlite> |
```

<u>g) Natural Join:</u> This will automatically join the customers and orders table on customer_id column since it is the only common column they share.

```
sqlite> SELECT *
   ...> FROM customers
   ...> NATURAL JOIN orders;
1|Alice|1|Laptop
2|Bob|2|Smartphone
1|Alice|3|Tablet
sqlite> |
```

<u>h) Join with Aggregation</u>: Firstly, we checked what tables are there currently and dropped the existing tables. And run the following query:

SELECT c.customer_name, COUNT(o.product_id) AS total_products_ordered
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.customer_name;

```
sqlite> .tables
customers          employees          product_suppliers  suppliers
departments        orders             products
sqlite> PRAGMA table_info(orders);
0|order_id|INTEGER|0||1
1|customer_id|INTEGER|0||0
2|product_name|TEXT|0||0
sqlite> DROP TABLE IF EXISTS orders;
sqlite> DROP TABLE IF EXISTS customers;
sqlite> CREATE TABLE customers (
(x1...>   customer_id INTEGER PRIMARY KEY,
(x1...>   customer_name TEXT
(x1...> );
sqlite> CREATE TABLE orders (
(x1...>   order_id INTEGER PRIMARY KEY,
(x1...>   customer_id INTEGER,
(x1...>   product_id INTEGER,
(x1...>   FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
(x1...> );
sqlite> INSERT INTO customers (customer_name) VALUES ('Alice');
sqlite> INSERT INTO customers (customer_name) VALUES ('Bob');
sqlite> INSERT INTO customers (customer_name) VALUES ('Charlie');
sqlite>
sqlite> INSERT INTO orders (customer_id, product_id) VALUES (1, 101);  -- Order by Alice
sqlite> INSERT INTO orders (customer_id, product_id) VALUES (1, 102);  -- Another order by Alice
sqlite> INSERT INTO orders (customer_id, product_id) VALUES (2, 103);  -- Order by Bob
sqlite> INSERT INTO orders (customer_id, product_id) VALUES (3, 104);  -- Order by Charlie
sqlite> INSERT INTO orders (customer_id, product_id) VALUES (2, 105);  -- Another order by Bob
sqlite> SELECT c.customer_name, COUNT(o.product_id) AS total_products_ordered
   ...> FROM customers c
   ...> JOIN orders o ON c.customer_id = o.customer_id
   ...> GROUP BY c.customer_name;
Alice|2
Bob|2
Charlie|1
sqlite> |
```

i) Multiple Joins: To get the complete list of order details including customer names, product names and the order dates we use multiple join. This is only example of what tables I have taken.

```
sqlite> -- List all tables in the database
sqlite> .tables
customers         employees         product_suppliers  suppliers
departments       orders            products
sqlite>
sqlite> -- Show the schema for the orders table
sqlite> .schema orders
CREATE TABLE orders (
  order_id INTEGER PRIMARY KEY,
  customer_id INTEGER,
  product_id INTEGER,
  FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);
sqlite> DROP TABLE IF EXISTS orders;
sqlite> DROP TABLE IF EXISTS products;
sqlite> DROP TABLE IF EXISTS customers;
sqlite> CREATE TABLE customers (
(x1...>   customer_id INTEGER PRIMARY KEY,
(x1...>   customer_name TEXT
(x1...> );
sqlite>
sqlite> CREATE TABLE products (
(x1...>   product_id INTEGER PRIMARY KEY,
(x1...>   product_name TEXT
(x1...> );
sqlite>
sqlite> CREATE TABLE orders (
(x1...>   order_id INTEGER PRIMARY KEY,
(x1...>   customer_id INTEGER,
(x1...>   product_id INTEGER,
(x1...>   order_date TEXT,
(x1...>   FOREIGN KEY (customer_id) REFERENCES customers(customer_id),
(x1...>   FOREIGN KEY (product_id) REFERENCES products(product_id)
(x1...> );
sqlite> -- Inserting data into customers
sqlite> INSERT INTO customers (customer_name) VALUES ('Alice');
sqlite> INSERT INTO customers (customer_name) VALUES ('Bob');
sqlite> INSERT INTO customers (customer_name) VALUES ('Charlie');
sqlite>
sqlite> -- Inserting data into products
sqlite> INSERT INTO products (product_name) VALUES ('Product A');
sqlite> INSERT INTO products (product_name) VALUES ('Product B');
sqlite> INSERT INTO products (product_name) VALUES ('Product C');
sqlite>
sqlite> -- Inserting data into orders
sqlite> INSERT INTO orders (customer_id, product_id, order_date) VALUES (1, 1, '2024-10-01');  -- Alice ordered Product A
sqlite> INSERT INTO orders (customer_id, product_id, order_date) VALUES (1, 2, '2024-10-02');  -- Alice ordered Product B
sqlite> INSERT INTO orders (customer_id, product_id, order_date) VALUES (2, 1, '2024-10-01');  -- Bob ordered Product A
sqlite> INSERT INTO orders (customer_id, product_id, order_date) VALUES (3, 3, '2024-10-03');  -- Charlie ordered Product C
sqlite> SELECT
   ...>     c.customer_name,
   ...>     p.product_name,
   ...>     o.order_date
   ...> FROM
   ...>     customers c
   ...> JOIN
   ...>     orders o ON c.customer_id = o.customer_id
   ...> JOIN
   ...>     products p ON o.product_id = p.product_id;
Alice|Product A|2024-10-01
Alice|Product B|2024-10-02
Bob|Product A|2024-10-01
Charlie|Product C|2024-10-03
sqlite> |
```

(2) Foreign Keys:

a) Foreign key definition: Foreign key establishes a foreign key relation in other table.

Here, we created authors table and books table and entered sample data.

```
sqlite> CREATE TABLE authors (
(x1...>     author_id INTEGER PRIMARY KEY,
(x1...>     author_name TEXT NOT NULL
(x1...> );
sqlite> CREATE TABLE books (
(x1...>     book_id INTEGER PRIMARY KEY,
(x1...>     book_title TEXT NOT NULL,
(x1...>     author_id INTEGER,
(x1...>     FOREIGN KEY (author_id) REFERENCES authors(author_id)
(x1...> );
sqlite> INSERT INTO authors (author_name) VALUES ('Agatha Christie');
sqlite> INSERT INTO authors (author_name) VALUES ('Mark Twain');
sqlite> INSERT INTO books (book_title, author_id) VALUES ('Murder on the Orient Express', 1);   -- References Agatha Christie
sqlite> INSERT INTO books (book_title, author_id) VALUES ('The Adventures of Tom Sawyer', 2);   -- References Mark Twain
sqlite> INSERT INTO books (book_title, author_id) VALUES ('And Then There Were None', 1);       -- References Agatha Christie
sqlite> INSERT INTO books (book_title, author_id) VALUES ('Unknown Book', 999);
sqlite> SELECT
   ...>     b.book_title,
   ...>     a.author_name
   ...> FROM
   ...>     books b
   ...> JOIN
   ...>     authors a ON b.author_id = a.author_id;
Murder on the Orient Express|Agatha Christie
The Adventures of Tom Sawyer|Mark Twain
And Then There Were None|Agatha Christie
sqlite> |
```

b) Cascading Deletes: To simply explain, cascading delete means when you delete a record in the parent table, it will delete the record from child tables automatically.

So, below when you deleted electronics, in the output electronics related are deleted.

```
sqlite> PRAGMA foreign_keys = OFF;
sqlite> DELETE FROM products;
sqlite> DROP TABLE IF EXISTS products;
sqlite> CREATE TABLE products (
(x1...>     product_id INTEGER PRIMARY KEY,
(x1...>     product_name TEXT NOT NULL,
(x1...>     category_id INTEGER,
(x1...>     FOREIGN KEY (category_id) REFERENCES categories(category_id) ON DELETE CASCADE
(x1...> );
sqlite> PRAGMA foreign_keys = ON;
sqlite> -- Insert into categories
sqlite> INSERT INTO categories (category_name) VALUES ('Electronics');
sqlite> INSERT INTO categories (category_name) VALUES ('Furniture');
sqlite>
sqlite> -- Insert into products
sqlite> INSERT INTO products (product_name, category_id) VALUES ('Smartphone', 1);
Runtime error: FOREIGN KEY constraint failed (19)
sqlite> INSERT INTO products (product_name, category_id) VALUES ('Laptop', 1);
Runtime error: FOREIGN KEY constraint failed (19)
sqlite> INSERT INTO products (product_name, category_id) VALUES ('Sofa', 2);
sqlite> INSERT INTO products (product_name, category_id) VALUES ('Table', 2);
sqlite>
sqlite> -- Delete Electronics category
sqlite> DELETE FROM categories WHERE category_id = 1;
sqlite>
sqlite> -- Check remaining products (Furniture products should remain)
sqlite> SELECT * FROM products;
1|Sofa|2
2|Table|2
sqlite> |
```

c) Violating foreign key constraint: This helps in maintaining no issues within tables if they have foreign keys.

```
sqlite> CREATE TABLE orders (
(x1...>     order_id INTEGER PRIMARY KEY,
(x1...>     customer_id INTEGER,
(x1...>     order_date TEXT,
(x1...>     FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
(x1...> );
sqlite> INSERT INTO customers (customer_name) VALUES ('Alice');  -- customer_id = 1
sqlite> INSERT INTO customers (customer_name) VALUES ('Bob');    -- customer_id = 2
sqlite>
sqlite> INSERT INTO orders (customer_id, order_date) VALUES (1, '2024-10-07');  -- Valid insertion for Alice
sqlite> INSERT INTO orders (customer_id, order_date) VALUES (3, '2024-10-08');  -- This will cause an error since customer_id = 3 doesn't exist
Runtime error: FOREIGN KEY constraint failed (19)
sqlite> -- Insert a valid order for Alice (customer_id = 1)
sqlite> INSERT INTO orders (customer_id, order_date) VALUES (1, '2024-10-07');  -- Valid insertion for Alice
sqlite>
sqlite> -- Insert a valid order for Bob (customer_id = 2)
sqlite> INSERT INTO orders (customer_id, order_date) VALUES (2, '2024-10-08');  -- Valid insertion for Bob
sqlite>
sqlite> -- The following line will fail because customer_id = 3 does not exist in the customers table
sqlite> -- INSERT INTO orders (customer_id, order_date) VALUES (3, '2024-10-09');  -- Invalid insertion
sqlite> |
```

(3) Consistency constraints:

a) Unique constraint: This ensures that there are no same values in columns and the rows. It helps maintain data integrity by preventing duplicates in a column and contain unique values.

```
sqlite> -- Insert another user with a unique email
sqlite> INSERT INTO users (user_name, email) VALUES ('Charlie', 'charlie@example.com');  -- This should succeed
sqlite> SELECT * FROM users;
1|Alice|alice@example.com
2|Bob|bob@example.com
3|Charlie|charlie@example.com
sqlite> |
```

b) Check constraint: It ensures that the data entered in the table should meet some criteria. In the following code, we made sure that the price must be greater than 0(Price>0).

```
sqlite> PRAGMA foreign_key_list(orders);           -- Check if this table references products
0|0|customers|customer_id|customer_id|NO ACTION|NO ACTION|NONE
sqlite> PRAGMA foreign_key_list(order_items);      -- Check if this table references products
sqlite> PRAGMA foreign_key_list(inventory);        -- Check if this table references products
sqlite> PRAGMA foreign_key_list(product_suppliers);-- Check if this table references products
0|0|suppliers|supplier_id|supplier_id|NO ACTION|NO ACTION|NONE
1|0|products|product_id|product_id|NO ACTION|NO ACTION|NONE
sqlite> DROP TABLE IF EXISTS order_items;        -- Drop dependent table first
sqlite> DROP TABLE IF EXISTS product_suppliers;  -- Drop dependent table first
sqlite> DROP TABLE IF EXISTS inventory;           -- Drop dependent table first
sqlite> DROP TABLE IF EXISTS products;
sqlite> CREATE TABLE products (
(x1...>     product_id INTEGER PRIMARY KEY,
(x1...>     product_name TEXT NOT NULL,
(x1...>     price REAL CHECK (price > 0)  -- This ensures that the price must be greater than 0
(x1...> );
sqlite>
```

c) Primary key and consistency: Primary key is the unique identifier for each table in the table. And consistency refers to accuracy and validity across the database.
The following command will display all the rows in the table: **SELECT * FROM courses;**

```
sqlite> CREATE TABLE courses (
(x1...>     course_id INTEGER,
(x1...>     course_name TEXT NOT NULL,
(x1...>     department_id INTEGER,
(x1...>     PRIMARY KEY (course_id, department_id)  -- Composite primary key
(x1...> );-- Insert courses into the table
sqlite> INSERT INTO courses (course_id, course_name, department_id) VALUES (101, 'Database Systems', 1);
sqlite> INSERT INTO courses (course_id, course_name, department_id) VALUES (102, 'Data Structures', 1);
sqlite> INSERT INTO courses (course_id, course_name, department_id) VALUES (101, 'Introduction to AI', 2);  -- Same course_id, different departm
ent
sqlite> SELECT * FROM courses;
101|Database Systems|1
102|Data Structures|1
101|Introduction to AI|2
sqlite>
```

d) Foreign key and consistency: In a relational database foreign key creates a link between two or more tables. These prevent insertion of invalid data ensuring the every relationship is valid and consistent.

```
sqlite> DROP TABLE IF EXISTS courses;
sqlite> CREATE TABLE courses (
(x1...>     course_id INTEGER PRIMARY KEY,
(x1...>     course_name TEXT NOT NULL
(x1...> );
sqlite> CREATE TABLE student_courses (
(x1...>     student_id INTEGER,
(x1...>     course_id INTEGER,
(x1...>     PRIMARY KEY (student_id, course_id),
(x1...>     FOREIGN KEY (student_id) REFERENCES students(student_id) ON DELETE CASCADE,
(x1...>     FOREIGN KEY (course_id) REFERENCES courses(course_id) ON DELETE CASCADE
(x1...> );-- Insert students
sqlite> INSERT INTO students (student_name) VALUES ('Alice');
sqlite> INSERT INTO students (student_name) VALUES ('Bob');
sqlite>
sqlite> -- Insert courses
sqlite> INSERT INTO courses (course_name) VALUES ('Database Systems');
sqlite> INSERT INTO courses (course_name) VALUES ('Data Structures');
sqlite>
sqlite> -- Insert valid student_courses entries
sqlite> INSERT INTO student_courses (student_id, course_id) VALUES (1, 1);  -- Alice enrolled in Database Systems
sqlite> INSERT INTO student_courses (student_id, course_id) VALUES (2, 2);  -- Bob enrolled in Data Structures
```

```
sqlite> SELECT * FROM users;
1|Alice|alice@example.com
2|Bob|bob@example.com
sqlite> SELECT * FROM students;
1|Alice
2|Bob
sqlite> SELECT * FROM courses;
1|Database Systems
2|Data Structures
sqlite> SELECT * FROM student_courses;
1|1
2|2
sqlite>
```

e) Not null constraint: It ensures that the column should not contain null values.
This table now shows usernames and their respective email addresses.

```
sqlite> -- Create the users table with NOT NULL constraints on username and email
sqlite> CREATE TABLE users (
(x1...>     user_id INTEGER PRIMARY KEY,
(x1...>     username TEXT NOT NULL,  -- This ensures that the username cannot be null
(x1...>     email TEXT NOT NULL      -- This ensures that the email cannot be null
(x1...> );
sqlite> INSERT INTO users (username, email) VALUES ('Alice', 'alice@example.com');
sqlite> INSERT INTO users (username, email) VALUES ('Bob', 'bob@example.com');
sqlite> SELECT * FROM users;
1|Alice|alice@example.com
2|Bob|bob@example.com
sqlite> |
```

f) Adding a check constraint to an existing table:
The following output reflects only the valid employee salaries whose salaries are greater than 0 as we gave the constraint to the table.

```
sqlite> -- Step 1: Drop the old employees table if it exists
sqlite> DROP TABLE IF EXISTS employees;
sqlite>
sqlite> -- Step 2: Create the employees table without CHECK constraint
sqlite> CREATE TABLE employees (
(x1...>     employee_id INTEGER PRIMARY KEY,
(x1...>     employee_name TEXT NOT NULL,
(x1...>     salary REAL
(x1...> );
sqlite>
sqlite> -- Step 3: Insert some valid sample data into the employees table
sqlite> INSERT INTO employees (employee_name, salary) VALUES ('Alice', 50000);
sqlite> INSERT INTO employees (employee_name, salary) VALUES ('Bob', 60000);
sqlite> -- Do not insert an invalid salary
sqlite>
sqlite> -- Step 4: Create the new employees table with the CHECK constraint
sqlite> CREATE TABLE employees_new (
(x1...>     employee_id INTEGER PRIMARY KEY,
(x1...>     employee_name TEXT NOT NULL,
(x1...>     salary REAL CHECK (salary > 0)
(x1...> );
sqlite>
sqlite> -- Step 5: Copy valid data from the old employees table to the new table
sqlite> INSERT INTO employees_new (employee_id, employee_name, salary)
   ...> SELECT employee_id, employee_name, salary FROM employees WHERE salary > 0;
sqlite>
sqlite> -- Step 6: Drop the old employees table
sqlite> DROP TABLE employees;
sqlite>
sqlite> -- Step 7: Rename the new table to the original table name
sqlite> ALTER TABLE employees_new RENAME TO employees;
sqlite>
sqlite> -- Step 8: Check the final output
sqlite> SELECT * FROM employees;
1|Alice|50000.0
2|Bob|60000.0
sqlite> |
```

g) Composite key constraint: It is a combination of two or more tables in a database table which identifies each row in the table uniquely.
As desired we have got the combination of student id and course id.

```
sqlite> DROP TABLE IF EXISTS student_courses;
sqlite> CREATE TABLE student_courses (
(x1...>     student_id INTEGER,
(x1...>     course_id INTEGER,
(x1...>     PRIMARY KEY (student_id, course_id),
(x1...>     FOREIGN KEY (student_id) REFERENCES students(student_id) ON DELETE CASCADE,
(x1...>     FOREIGN KEY (course_id) REFERENCES courses(course_id) ON DELETE CASCADE
(x1...> );
sqlite> .tables
authors          customers        products         users
books            departments      student_courses
categories       employees        students
courses          orders           suppliers
sqlite> -- Insert sample students
sqlite> INSERT INTO students (student_name) VALUES ('Alice');
sqlite> INSERT INTO students (student_name) VALUES ('Bob');
sqlite>
sqlite> -- Insert sample courses
sqlite> INSERT INTO courses (course_name) VALUES ('Database Systems');
sqlite> INSERT INTO courses (course_name) VALUES ('Data Structures');
sqlite> INSERT INTO student_courses (student_id, course_id) VALUES (1, 1);  -- Alice in Database Systems
sqlite> INSERT INTO student_courses (student_id, course_id) VALUES (2, 2);  -- Bob in Data Structures
sqlite> SELECT * FROM students;       -- Check students table
1|Alice
2|Bob
3|Alice
4|Bob
sqlite> SELECT * FROM courses;        -- Check courses table
1|Database Systems
2|Data Structures
3|Database Systems
4|Data Structures
sqlite> SELECT * FROM student_courses; -- Check student_courses table
1|1
2|2
sqlite> |
```