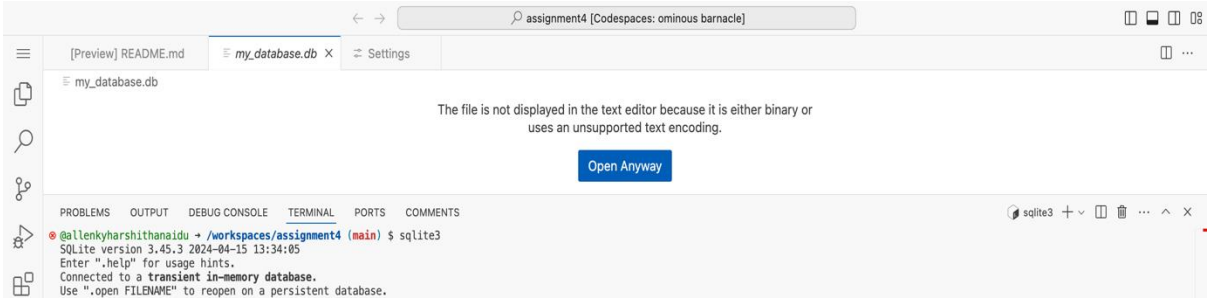
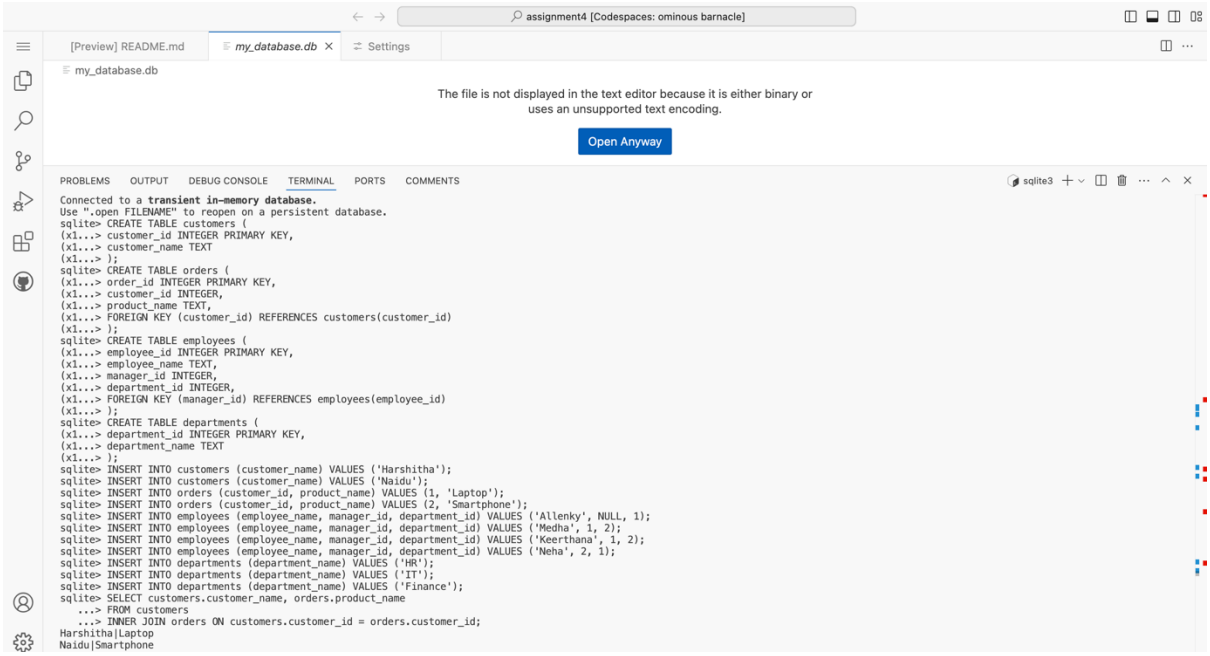
ASSIGNMENT-4

SQL Joins:

a) Inner Join:

An Inner Join is one of the most commonly used types of joins in SQL. It is used to combine rows from two or more tables based on a related column between them. The Inner Join returns only the rows where there is a match in both tables involved in the join. If no match is found, those rows are excluded from the result set.





b) Left Join:

A Left Join also known as a Left Outer Join is a type of SQL join that retrieves all the rows from the left table and the matched rows from the right table. If there is no match, the result will still include all rows from the left table, but the columns from the right table will contain NULL for those unmatched rows.

```
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 ('Harshitha');
sqlite> INSERT INTO customers (customer_name) VALUES ('Naidu');
sqlite> INSERT INTO customers (customer_name) VALUES ('allenky');
sqlite> INSERT INTO orders (customer_id, product_name) VALUES (1, 'Laptop');
INSERT INTO orders (customer_id, product_name) VALUES (2, 'Smartphone');
sqlite> SELECT customers.customer_name, orders.product_name
   ...> FROM customers
   ...> LEFT JOIN orders ON customers.customer_id = orders.customer_id;
Harshitha|Laptop
Naidu|Smartphone
allenky|
sqlite>
```

c) Right Join:

A Right Join also known as a Right Outer Join is a type of SQL join that retrieves all rows from the right table and the matched rows from the left table .If there is no match, the result will still include all rows from the right table, but the columns from the left table will contain NULL for those unmatched rows.

```
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),
    FOREIGN KEY (supplier_id) REFERENCES suppliers(supplier_id)
);
sqlite> INSERT INTO products (product_name) VALUES ('Laptop');
INSERT INTO products (product_name) VALUES ('Smartphone');
INSERT INTO products (product_name) VALUES ('Tablet');
sqlite> INSERT INTO suppliers (supplier_name) VALUES ('Supplier A');
sqlite> INSERT INTO suppliers (supplier_name) VALUES ('Supplier B');
sqlite> INSERT INTO product_suppliers (product_id, supplier_id) VALUES (1, 1);
sqlite> INSERT INTO product_suppliers (product_id, supplier_id) VALUES (2, 2);
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|Supplier A
Smartphone|Supplier B
Tablet|
sqlite>
```

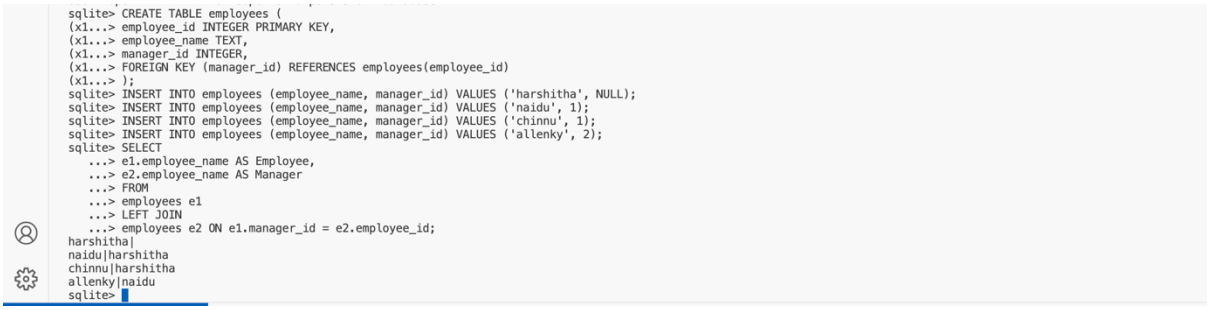
d) Full Outer Join:

A Full Outer Join is a type of SQL join that combines the results of both Left Join and Right Join. It returns all rows from both tables being joined, with NULL values in the result set where there are no matches between the two tables.

```
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 employees (employee_name, department_id) VALUES ('Medha', 1);
sqlite> INSERT INTO employees (employee_name, department_id) VALUES ('keerthana', 2);
sqlite> INSERT INTO employees (employee_name, department_id) VALUES ('Chinnu', NULL);
sqlite> INSERT INTO departments (department_name) VALUES ('HR');
sqlite> INSERT INTO departments (department_name) VALUES ('IT');
sqlite> INSERT INTO departments (department_name) VALUES ('Marketing');
sqlite> SELECT employees.employee_name, departments.department_name
FROM employees
   ...> LEFT JOIN departments ON employees.department_id = departments.department_id
   ...> UNION
   ...> SELECT employees.employee_name, departments.department_name
   ...> FROM departments
   ...> LEFT JOIN employees ON employees.department_id = departments.department_id;
|Marketing
Chinnu|
Medha|HR
keerthana|IT
sqlite>
```

e) Self Join:

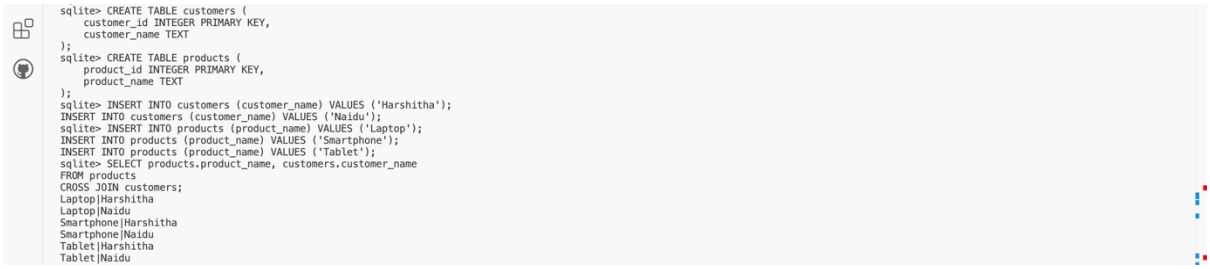
A Self Join is a special type of join in SQL where a table is joined with itself. This can be useful for comparing rows within the same table or for retrieving hierarchical data from a single table.



```
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 ('harshitha', NULL);
sqlite> INSERT INTO employees (employee_name, manager_id) VALUES ('naidu', 1);
sqlite> INSERT INTO employees (employee_name, manager_id) VALUES ('chinnu', 1);
sqlite> INSERT INTO employees (employee_name, manager_id) VALUES ('allenky', 2);
sqlite> SELECT
   ...> e1.employee_name AS Employee,
   ...> e2.employee_name AS Manager
   ...> FROM
   ...> employees e1
   ...> LEFT JOIN
   ...> employees e2 ON e1.manager_id = e2.employee_id;
harshitha|
naidu|harshitha
chinnu|harshitha
allenky|naidu
sqlite>
```

f) Cross Join:

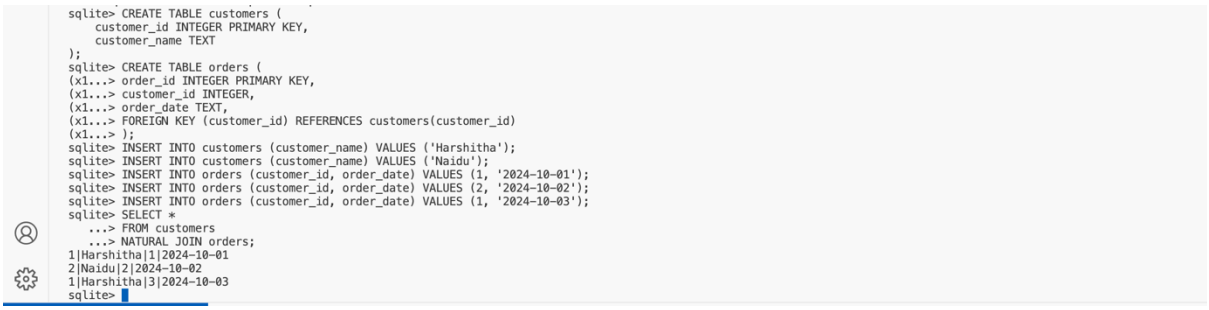
A Cross Join is a type of SQL join that returns the Cartesian product of two tables. This means that every row from the first table is combined with every row from the second table. Cross joins are useful when you want to combine all possible combinations of rows from two tables.



```
sqlite> CREATE TABLE customers (
    customer_id INTEGER PRIMARY KEY,
    customer_name TEXT
);
sqlite> CREATE TABLE products (
    product_id INTEGER PRIMARY KEY,
    product_name TEXT
);
sqlite> INSERT INTO customers (customer_name) VALUES ('Harshitha');
INSERT INTO customers (customer_name) VALUES ('Naidu');
sqlite> INSERT INTO products (product_name) VALUES ('Laptop');
INSERT INTO products (product_name) VALUES ('Smartphone');
INSERT INTO products (product_name) VALUES ('Tablet');
sqlite> SELECT products.product_name, customers.customer_name
FROM products
CROSS JOIN customers;
Laptop|Harshitha
Laptop|Naidu
Smartphone|Harshitha
Smartphone|Naidu
Tablet|Harshitha
Tablet|Naidu
```

g) Natural Join:

A Natural Join is a type of SQL join that automatically joins two tables based on all columns with the same names and compatible data types in both tables. It simplifies the process of joining tables by eliminating the need to specify the join condition explicitly.



```
sqlite> CREATE TABLE customers (
    customer_id INTEGER PRIMARY KEY,
    customer_name TEXT
);
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 ('Harshitha');
sqlite> INSERT INTO customers (customer_name) VALUES ('Naidu');
sqlite> INSERT INTO orders (customer_id, order_date) VALUES (1, '2024-10-01');
sqlite> INSERT INTO orders (customer_id, order_date) VALUES (2, '2024-10-02');
sqlite> INSERT INTO orders (customer_id, order_date) VALUES (1, '2024-10-03');
sqlite> SELECT *
   ...> FROM customers
   ...> NATURAL JOIN orders;
1|Harshitha|1|2024-10-01
2|Naidu|2|2024-10-02
1|Harshitha|3|2024-10-03
sqlite>
```

h) Join with Aggregation:

A Join with Aggregation is a SQL operation that combines the results of two or more tables and applies an aggregation function to summarize data. This type of join is particularly useful when you want to perform

calculations (like sums, counts, averages, etc.) on grouped data from the joined tables.

```
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 ('Harshitha');
sqlite> INSERT INTO customers (customer_name) VALUES ('Naidu');
sqlite> INSERT INTO orders (customer_id, product_id) VALUES (1, 101);
sqlite> INSERT INTO orders (customer_id, product_id) VALUES (2, 102);
sqlite> INSERT INTO orders (customer_id, product_id) VALUES (1, 103);
sqlite> SELECT
   ...> customers.customer_name,
   ...> COUNT(orders.order_id) AS total_products_ordered
   ...> FROM
   ...> customers
   ...> INNER JOIN
   ...> orders ON customers.customer_id = orders.customer_id
   ...> GROUP BY
   ...> customers.customer_name;
Harshitha|2
Naidu|1
sqlite>
```

i) Multiple Joins:

Multiple Joins refer to the SQL operation of joining more than two tables in a single query. This is useful when you need to retrieve related data from multiple tables based on defined relationships. You can use various types of joins (INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL OUTER JOIN, etc.) when performing multiple joins.

```
sqlite> CREATE TABLE customers (
    customer_id INTEGER PRIMARY KEY,
    customer_name TEXT
);
sqlite> CREATE TABLE products (
    product_id INTEGER PRIMARY KEY,
    product_name TEXT
);
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> INSERT INTO customers (customer_name) VALUES ('Harshitha');
INSERT INTO customers (customer_name) VALUES ('Naidu');
sqlite> INSERT INTO products (product_name) VALUES ('Laptop');
INSERT INTO products (product_name) VALUES ('Smartphone');
INSERT INTO products (product_name) VALUES ('Tablet');
sqlite> INSERT INTO products (product_name) VALUES ('Laptop');
INSERT INTO products (product_name) VALUES ('Smartphone');
INSERT INTO products (product_name) VALUES ('Tablet');
sqlite> INSERT INTO orders (customer_id, product_id, order_date) VALUES (1, 1, '2024-10-01');
sqlite> INSERT INTO orders (customer_id, product_id, order_date) VALUES (2, 2, '2024-10-02');
sqlite> INSERT INTO orders (customer_id, product_id, order_date) VALUES (1, 3, '2024-10-03');
sqlite> SELECT
   ...> customers.customer_name,
   ...> products.product_name,
   ...> orders.order_date
   ...> FROM
   ...>  orders
   ...> INNER JOIN
   ...> customers ON orders.customer_id = customers.customer_id
   ...> INNER JOIN
   ...> products ON orders.product_id = products.product_id;
Harshitha|Laptop|2024-10-01
Naidu|Smartphone|2024-10-02
Harshitha|Tablet|2024-10-03
sqlite>
```

2. Foreign Keys:

a) Foreign Key:

A Foreign Key is a field in one table that uniquely identifies a row of another table or the same table. It is a key used to link two tables together. The purpose of a foreign key is to ensure referential integrity between the data in the two tables.

```
sqlite> CREATE TABLE authors (
    author_id INTEGER PRIMARY KEY,
    author_name TEXT
);

CREATE TABLE books (
    book_id INTEGER PRIMARY KEY,
    book_title TEXT,
    author_id INTEGER,
    FOREIGN KEY (author_id) REFERENCES authors(author_id)
);
sqlite> INSERT INTO authors (author_name) VALUES ('J.K. Rowling');
INSERT INTO authors (author_name) VALUES ('George R.R. Martin');
sqlite> INSERT INTO books (book_title, author_id) VALUES ('A Game of Thrones', 2);
sqlite> INSERT INTO authors (author_name) VALUES ('Jane Austen');
sqlite> INSERT INTO books (book_title, author_id) VALUES ('A stone', 1);
sqlite> .tables
authors  books
sqlite> SELECT * FROM authors;
1|J.K. Rowling
2|George R.R. Martin
3|Jane Austen
sqlite> SELECT * FROM books;
1|A Game of Thrones|2
2|A stone|1
sqlite> SELECT * FROM authors;
SELECT * FROM books;
1|J.K. Rowling
2|George R.R. Martin
3|Jane Austen
1|A Game of Thrones|2
2|A stone|1
sqlite> .quit
```

b) Cascading Deletes:

Cascading Deletes is a feature in relational databases that automatically deletes related records in a child table when a record in the parent table is deleted. This feature helps maintain referential integrity between tables, ensuring that there are no orphaned records.

```
sqlite> PRAGMA foreign_keys = ON;
sqlite> PRAGMA foreign_keys;
1
sqlite> DROP TABLE IF EXISTS products;
DROP TABLE IF EXISTS categories;
sqlite> CREATE TABLE categories (
    category_id INTEGER PRIMARY KEY,
    category_name TEXT
);

CREATE TABLE products (
    product_id INTEGER PRIMARY KEY,
    product_name TEXT,
    category_id INTEGER,
    FOREIGN KEY (category_id) REFERENCES categories(category_id) ON DELETE CASCADE
);
sqlite> INSERT INTO categories (category_name) VALUES ('Electronics');
INSERT INTO categories (category_name) VALUES ('Books');
sqlite> INSERT INTO products (product_name, category_id) VALUES ('Laptop', 1);
INSERT INTO products (product_name, category_id) VALUES ('Smartphone', 1);
INSERT INTO products (product_name, category_id) VALUES ('Harry Potter', 2);
sqlite> DELETE FROM categories WHERE category_id = 1;
sqlite> SELECT * FROM products;
3|Harry Potter|2
sqlite>
```

c) Violating Foreign Key Constraint:

Violating a Foreign Key Constraint occurs when an operation in a relational database attempts to create a relationship that does not adhere to the rules defined by the foreign key. This ensures data integrity and maintains valid relationships between tables.

```
sqlite> PRAGMA foreign_keys = ON;
sqlite> PRAGMA foreign_keys;
1
sqlite> sqlite3 my_database.db
   ...> DROP TABLE IF EXISTS orders;
Parse error: near "sqlite3": syntax error
  sqlite3 my_database.db  DROP TABLE IF EXISTS orders;
  ^--- error here
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...> order_date TEXT,
(x1...> FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);
sqlite> INSERT INTO customers (customer_name) VALUES ('Harshitha');
INSERT INTO customers (customer_name) VALUES ('Naidu');
sqlite> INSERT INTO orders (customer_id, order_date) VALUES (999, '2024-10-05');
Runtime error: FOREIGN KEY constraint failed (19)
sqlite>
```

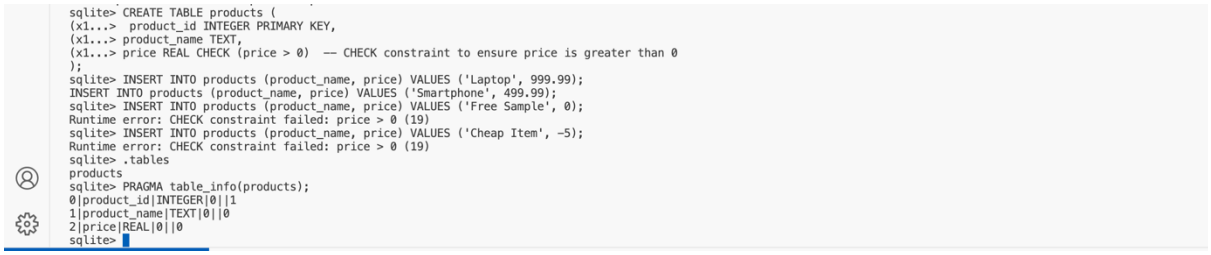
3. Consistency Constraints:

a) Unique Constraint:

A Unique Constraint is a rule applied to a column or a group of columns in a relational database table that ensures all values in that column are distinct across the table.

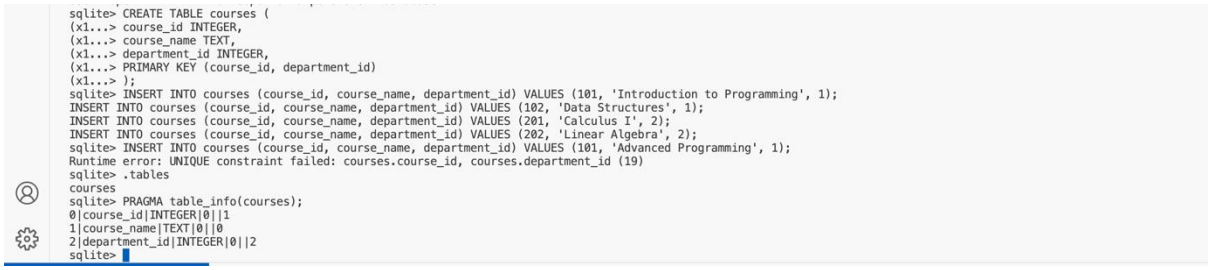
b) Check Constraint:

A Check Constraint is a rule applied to a column in a relational database table that enforces a specific condition on the values that can be inserted or updated in that column. This helps ensure data integrity by limiting the range or format of values allowed in a particular column.



c) Primary Key and Consistency:

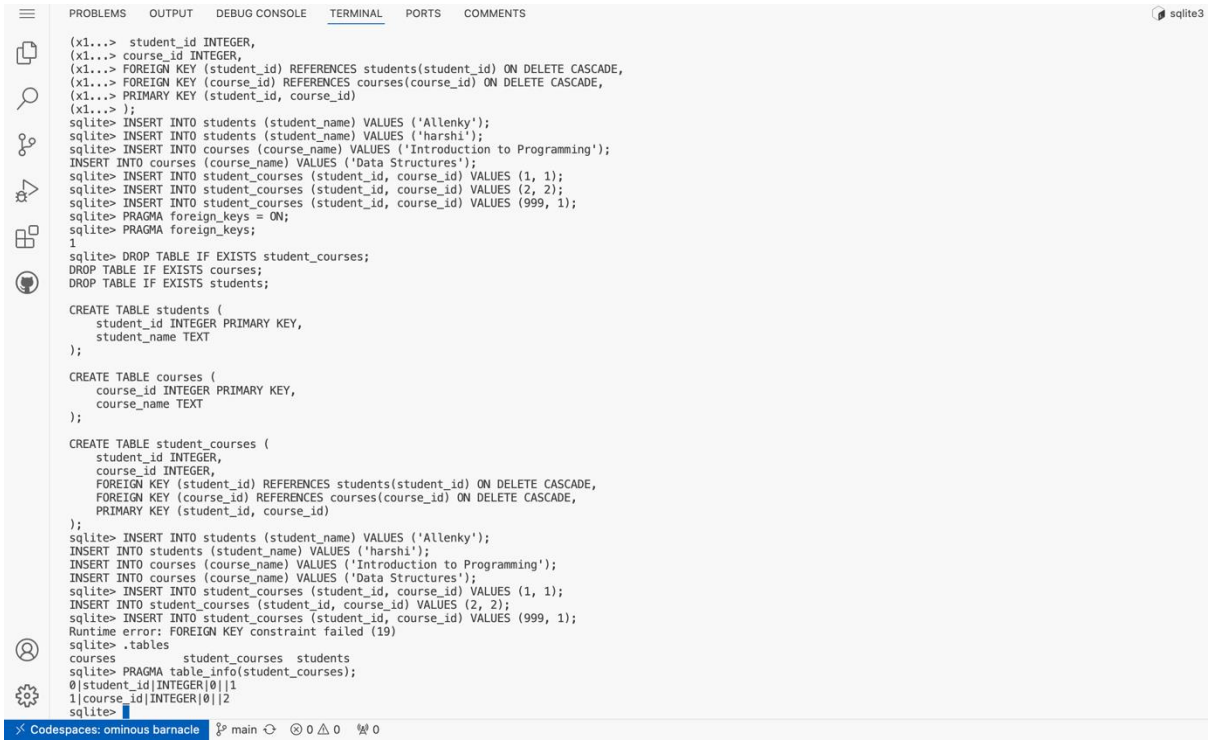
A Primary Key is a unique identifier for each record in a database table. It ensures that each entry in the table can be uniquely identified, enforcing data integrity and consistency within the database.



d) Foreign Key and Consistency:

A Foreign Key is a field (or a collection of fields) in one table that uniquely identifies a row of another table or the same table. It is a key used to establish a

relationship between two tables. Foreign keys play a crucial role in maintaining referential integrity and consistency in relational databases.

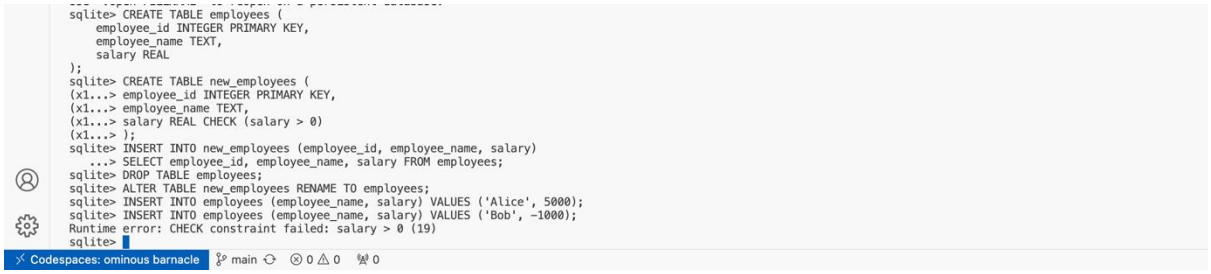


e) Not Null Constraint:

A Not Null Constraint is a rule applied to a column in a relational database table that ensures that the column cannot have NULL values. This constraint is used to enforce the requirement that a column must always contain a value, which is essential for maintaining data integrity.

f) Adding a Check Constraint to an Existing Table:

A Check Constraint is used to enforce specific conditions on the values in a column or a set of columns in a database table. If you want to add a check constraint to an existing table, you typically need to create a new table with the desired constraints and then migrate the data from the old table to the new one, as most database systems do not support adding constraints directly to existing columns.



g) Composite Key Constraint:

A Composite Key Constraint is a type of primary key constraint that consists of two or more columns in a table. The combination of these columns uniquely identifies each record in that table. Composite keys are particularly useful when a single column is not sufficient to uniquely identify a record, and multiple columns are needed.

Github link:

https://github.com/allenkyharshithanaidu/assignment4?tab=readme-ov-file#assignment4