**GH Raisoni College of Engineering and Management, Pune**

**(An Empowered Autonomous Institute Affiliated to Savitribai Phule Pune University)**

**Computer Engineering Department**

**Value Added Certification Course in IoS and Android Development**

**Laboratory Manual**

Subject: SQLite and Firebase

(With effect from Academic Year 2024-25)

## Evaluation Scheme

| | |
|---|---|
| **Class:** SY B.Tech | **Sem :** III |
| **Exam Scheme :** Internal Practical : 25 Marks | |

| Sr.No | Name of the Experiment |
|---|---|
| 1 | To install SQLite , Set up SQLite environment , create simple database and table and insert sample data into the table |
| 2 | To perform basic SQL Operations such as SELECT, INSERT, UPDATE, and DELETE. • Query the database to retrieve specific data. • Update and delete records in the database. |
| 3 | To create and manage new SQLite database , manage tables with various data types and implement constraints such as PRIMARY KEY, FOREIGN KEY, UNIQUE, and NOT NULL. |
| 4 | To Write queries to retrieve data from multiple tables using JOIN operations and different query clauses like WHERE, ORDER BY, and GROUP BY |
| 5 | T o perform SQLite Transactions and Atomic Operations |
| 6 | To Create indexes on tables to improve query performance |
| 7 | To perform Triggers and Views operation on SQLite |
| 8 | To Perform CRUD operations through the programming language |
| 9 | . To Create and configure a Firebase project |
| 10 | To perform Firebase Authentication and Security Rules |

**Subject Teacher** **HOD**

| Experiment No: 1 | | |
|---|---|---|
| **Aim:** Installing and Setting Up SQLite<br>Install SQLite on your system.<br>Set up SQLite environment.<br>Create a simple database and table.<br>Insert sample data into the table. | | |
| | | |

**Objective:** To understand the basic of SQLite database

**Outcome:** Students will able to understand the about SQLite environment and installation of SQLite and perform basic operation like creating database and table .

**Pre-requisite:** Computer with SQLite installed
**Theory:**

**1.To install SQLite database and set up your environment**

- **Download SQLite**:
  - Visit the SQLite download page.
  - Under the "Precompiled Binaries for Windows" section, download the zip file for the SQLite tools (e.g., `sqlite-tools-win32-x86-*.*.*.zip`).
- **Extract the Files**:
  - Extract the downloaded zip file to a folder of your choice.
- **Add to PATH (Optional)**:
  - You can add the directory containing `sqlite3.exe` to your system's PATH for easier access from the command line.
- **Run SQLite**:
  - Open Command Prompt, navigate to the folder, and run `sqlite3` to start the SQLite shell.

**2. Create a simple database and table.**

1. **Open your command prompt or terminal.**
2. **Start SQLite** by typing:

```
sqlite3 sample.db
```

This command creates a new database file named `sample.db` (or opens it if it already exists).

**Step 2: Create a Table**

Once you're in the SQLite shell, you can create a table. Here's an example of creating a simple `users` table:

Open Jupiter Notebook

```
CREATE TABLE users (
 id INTEGER PRIMARY KEY AUTOINCREMENT,
name TEXT NOT NULL,
email TEXT NOT NULL UNIQUE,
age INTEGER
 );
```

**Step 3: Insert Sample Data**

Open Jupiter Notebook

You can insert some sample records into the `users` table:

```
  INSERT INTO users (name, email, age) VALUES ('Alice',
'alice@example.com', 30);
  INSERT INTO users (name, email, age) VALUES ('Bob', 'bob@example.com',
25);
  INSERT INTO users (name, email, age) VALUES ('Charlie',
'charlie@example.com', 35);
```

Output :

```
[1]: import sqlite3

# Step 1: Connect to the database (creates it if it doesn't exist)
conn = sqlite3.connect('sample.db')

# Step 2: Create a cursor object using the connection
cursor = conn.cursor()

# Step 3: Create a new table
cursor.execute('''
CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    email TEXT NOT NULL UNIQUE,
    age INTEGER
)
''')

# Step 4: Insert sample data
cursor.execute("INSERT INTO users (name, email, age) VALUES (?, ?, ?)", ('Alice', 'alice@example.com', 30))
cursor.execute("INSERT INTO users (name, email, age) VALUES (?, ?, ?)", ('Bob', 'bob@example.com', 25))
cursor.execute("INSERT INTO users (name, email, age) VALUES (?, ?, ?)", ('Charlie', 'charlie@example.com', 35))

# Step 5: Commit the changes
conn.commit()

# Optional: Fetch and print all rows from the users table
cursor.execute("SELECT * FROM users")
rows = cursor.fetchall()
for row in rows:
    print(row)

# Close the connection
conn.close()
```

```
(1, 'Alice', 'alice@example.com', 30)
(2, 'Bob', 'bob@example.com', 25)
(3, 'Charlie', 'charlie@example.com', 35)
```

**Conclusion** : In this way we have studied creation of database and table

| **Experiment No: 2** | | |
|---|---|---|
| **Aim:** Basic SQL Operations<br>• Perform basic SQL operations such as SELECT, INSERT, UPDATE, and DELETE.<br>• Query the database to retrieve specific data.<br>• Update and delete records in the database. | | |
| | | |

**Objective:** To learn basic SQL operation

**Outcome:** Student will able to understand basic SQL operations such as SELECT, INSERT, UPDATE, and DELETE.

**Pre-requisite:** Computer with anaconda installe

**SELECT**

The **SELECT** statement retrieves data from a database.

**Syntax:**

```
sql
Copy code
SELECT column1, column2, ... FROM table_name WHERE condition;
```

**Examples:**

- Select all columns:

  ```
  sql
  Copy code
  SELECT * FROM users;
  ```

- Select specific columns:

  ```
  sql
  Copy code
  SELECT name, age FROM users;
  ```

- Select with a condition:

  ```
  sql
  Copy code
  SELECT * FROM users WHERE age > 25;
  ```

**2. INSERT**

The **INSERT** statement adds new records to a table.

**Syntax:**

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

**Examples:**

- Insert a new user:

  ```
  sql
  Copy code
  INSERT INTO users (name, email, age) VALUES ('Alice',
  'alice@example.com', 30);
  ```

- Insert multiple records:

  ```
  sql
  Copy code
  INSERT INTO users (name, email, age) VALUES
  ('Bob', 'bob@example.com', 25),
  ('Charlie', 'charlie@example.com', 35);
  ```

**3. UPDATE**

The **UPDATE** statement modifies existing records in a table.

**Syntax:**

```
sql
Copy code
UPDATE table_name SET column1 = value1, column2 = value2, ... WHERE
condition;
```

**Examples:**

- Update a user's age:

  ```
  sql
  Copy code
  UPDATE users SET age = 31 WHERE name = 'Alice';
  ```

- Update multiple fields:

  ```
  sql
  Copy code
  UPDATE users SET email = 'bob_new@example.com', age = 26 WHERE name =
  'Bob';
  ```

**4. DELETE**

The **DELETE** statement removes records from a table.

**Syntax:**

```
sql
Copy code
DELETE FROM table_name WHERE condition;
```

**Examples:**

- Delete a specific user:

  ```
  sql
  Copy code
  DELETE FROM users WHERE name = 'Charlie';
  ```

- Delete all users above a certain age:

```
sql
Copy code
DELETE FROM users WHERE age > 40;  -- Use with caution!
```

In [1]:

```python
import sqlite3

# Connect to the SQLite database (or create it if it doesn't exist)
conn = sqlite3.connect('basic_operations.db')
cursor = conn.cursor()

# Create a table
cursor.execute('''
CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    email TEXT NOT NULL UNIQUE,
    age INTEGER
)
''')

# Function to insert a user
def insert_user(name, email, age):
    cursor.execute("INSERT INTO users (name, email, age) VALUES (?, ?, ?)", (name, email, age))
    conn.commit()
    print(f"Inserted: {name}, {email}, {age}")

# Function to select all users
def select_users():
    cursor.execute("SELECT * FROM users")
    rows = cursor.fetchall()
    for row in rows:
        print(row)

# Function to update a user
def update_user(user_id, name=None, email=None, age=None):
    updates = []
    if name:
        updates.append(f"name = '{name}'")
    if email:
        updates.append(f"email = '{email}'")
    if age is not None:
        updates.append(f"age = {age}")

    if updates:
        update_query = f"UPDATE users SET {', '.join(updates)} WHERE id = {user_id}"
        cursor.execute(update_query)
        conn.commit()
        print(f"Updated user with ID {user_id}")
```

```
    print(f"Deleted user with ID {user_id}")

# Sample Operations
if __name__ == "__main__":
    # Insert users
    insert_user('Alice', 'alice@example.com', 30)
    insert_user('Bob', 'bob@example.com', 25)

    # Select all users
    print("\nAll users:")
    select_users()

    # Update a user
    update_user(1, age=31)

    # Select all users after update
    print("\nUsers after update:")
    select_users()

    # Delete a user
    delete_user(2)

    # Select all users after deletion
    print("\nUsers after deletion:")
    select_users()

# Close the connection
conn.close()
```

```
Inserted: Alice, alice@example.com, 30
Inserted: Bob, bob@example.com, 25

All users:
(1, 'Alice', 'alice@example.com', 30)
(2, 'Bob', 'bob@example.com', 25)
Updated user with ID 1

Users after update:
(1, 'Alice', 'alice@example.com', 31)
(2, 'Bob', 'bob@example.com', 25)
Deleted user with ID 2

Users after deletion:
(1, 'Alice', 'alice@example.com', 31)
```

## Conclusion

In this way we have studied for crating SQLite databse basic operations .

| Experiment No:<br>3 | | |
|---|---|---|
| **Aim: Creating and Managing Databases**<br>**• Create a new SQLite database.**<br>**• Define and manage tables with various data types.**<br>**• Implement constraints such as PRIMARY KEY, FOREIGN KEY, UNIQUE, and NOT NULL.** | | |
| | | |

**Objective:** To learn **PRIMARY KEY, FOREIGN KEY, UNIQUE, and NOT NULL**

**Outcome:** Student will able to **PRIMARY KEY, FOREIGN KEY, UNIQUE, and NOT NULL**

**Pre-requisite:** Computer with anaconda installed

**Theory:**

### 1. PRIMARY KEY

A **PRIMARY KEY** uniquely identifies each record in a table. Each table can have only one primary key, and it must contain unique values. A primary key cannot contain `NULL` values.

**Example:**

```
sql
Copy code
CREATE TABLE users (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    email TEXT NOT NULL
);
```

In this example, `id` is the primary key for the `users` table.

### 2. FOREIGN KEY

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. The foreign key establishes a relationship between the two tables.

**Example:**

```
sql
Copy code
```

```
CREATE TABLE orders (
    order_id INTEGER PRIMARY KEY,
    user_id INTEGER,
    product TEXT NOT NULL,
    FOREIGN KEY (user_id) REFERENCES users(id)
);
```

In this example, `user_id` in the `orders` table is a foreign key that references the `id` field in the `users` table.

### 3. UNIQUE

The **UNIQUE** constraint ensures that all values in a column are different. Unlike the primary key, a table can have multiple unique constraints, and unique columns can contain `NULL` values (though they cannot contain duplicate non-null values).

**Example:**

```
sql
Copy code
CREATE TABLE users (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    email TEXT NOT NULL UNIQUE
);
```

In this example, the `email` column must contain unique values across the `users` table.

### 4. NOT NULL

The **NOT NULL** constraint ensures that a column cannot have a `NULL` value. This constraint enforces that a field must contain a value when a new record is inserted.

**Example:**

```
CREATE TABLE users (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    email TEXT NOT NULL
);
```

```
]: import sqlite3

# Connect to the SQLite database (creates it if it doesn't exist)
conn = sqlite3.connect('constraints_example.db')
cursor = conn.cursor()

# Create the 'users' table with constraints
cursor.execute('''
CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY,          -- PRIMARY KEY
    name TEXT NOT NULL,              -- NOT NULL
    email TEXT NOT NULL UNIQUE       -- UNIQUE and NOT NULL
)
''')

# Create the 'orders' table with constraints
cursor.execute('''
CREATE TABLE IF NOT EXISTS orders (
    order_id INTEGER PRIMARY KEY,    -- PRIMARY KEY
    user_id INTEGER,                 -- Foreign Key
    product TEXT NOT NULL,           -- NOT NULL
    FOREIGN KEY (user_id) REFERENCES users(id) -- FOREIGN KEY
)
''')

# Sample data insertion
try:
    # Inserting users
    cursor.execute("INSERT INTO users (name, email) VALUES (?, ?)", ('Alice', 'alice@example.com'))
    cursor.execute("INSERT INTO users (name, email) VALUES (?, ?)", ('Bob', 'bob@example.com'))

    # Inserting an order
    cursor.execute("INSERT INTO orders (user_id, product) VALUES (?, ?)", (1, 'Laptop'))
    cursor.execute("INSERT INTO orders (user_id, product) VALUES (?, ?)", (1, 'Phone'))

    # Attempting to insert a duplicate email (should raise an error)
    cursor.execute("INSERT INTO users (name, email) VALUES (?, ?)", ('Charlie', 'alice@example.com'))
except sqlite3.IntegrityError as e:
    print(f"Integrity Error: {e}")

# Commit the changes
conn.commit()
```

```
# Sample data insertion
try:
    # Inserting users
    cursor.execute("INSERT INTO users (name, email) VALUES (?, ?)", ('Alice', 'alice@example.com'))
    cursor.execute("INSERT INTO users (name, email) VALUES (?, ?)", ('Bob', 'bob@example.com'))

    # Inserting an order
    cursor.execute("INSERT INTO orders (user_id, product) VALUES (?, ?)", (1, 'Laptop'))
    cursor.execute("INSERT INTO orders (user_id, product) VALUES (?, ?)", (1, 'Phone'))

    # Attempting to insert a duplicate email (should raise an error)
    cursor.execute("INSERT INTO users (name, email) VALUES (?, ?)", ('Charlie', 'alice@example.com'))
except sqlite3.IntegrityError as e:
    print(f"Integrity Error: {e}")

# Commit the changes
conn.commit()

# Fetch and display all users
print("Users:")
cursor.execute("SELECT * FROM users")
for row in cursor.fetchall():
    print(row)

# Fetch and display all orders
print("\nOrders:")
cursor.execute("SELECT * FROM orders")
for row in cursor.fetchall():
    print(row)

# Close the connection
conn.close()
```

```
Integrity Error: UNIQUE constraint failed: users.email
Users:
(1, 'Alice', 'alice@example.com')
(2, 'Bob', 'bob@example.com')

Orders:
(1, 1, 'Laptop')
(2, 1, 'Phone')
```

**Conclusion**

In this way we have studied different keys for SQLite database .

| Experiment No: 4 | | |
|---|---|---|
| **Aim:** Querying SQLite Databases<br>• Write queries to retrieve data from multiple tables using JOIN operations.<br>• Use subqueries to fetch data.<br>• Experiment with different query clauses like WHERE, ORDER BY, and GROUP BY. | | |
| | | |

**Objective:** To learn how to implement WHERE , ORDER BY and GROUP BY clauses
**Outcome:** Student will able to understand and write sql command with WHERE , ORDER BY and GROUP BY clauses

**Pre-requisite:** Computer with anaconda installed

**Theory:**

## 1. WHERE Clause

The `WHERE` clause is used to filter records based on specified conditions. It allows you to retrieve only those rows that meet certain criteria.

**Syntax:**

```
sql
Copy code
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

**Examples:**

- Selecting records with a specific condition:

  ```
  sql
  Copy code
  SELECT * FROM users WHERE age > 30;
  ```

- Using multiple conditions:

  ```
  sql
  ```

```
Copy code
SELECT * FROM users WHERE age > 25 AND email LIKE '%example.com';
```

## 2. GROUP BY Clause

The `GROUP BY` clause is used to group rows that have the same values in specified columns into summary rows. It is often used in conjunction with aggregate functions like `COUNT()`, `SUM()`, `AVG()`, etc.

**Syntax:**

```
sql
Copy code
SELECT column1, aggregate_function(column2)
FROM table_name
WHERE condition
GROUP BY column1;
```

**Examples:**

- Counting the number of users in each age group:

  ```
  sql
  Copy code
  SELECT age, COUNT(*) FROM users GROUP BY age;
  ```

- Getting the total number of orders per user:

  ```
  sql
  Copy code
  SELECT user_id, COUNT(*) FROM orders GROUP BY user_id;
  ```

## 3. ORDER BY Clause

The `ORDER BY` clause is used to sort the result set of a query by one or more columns. You can specify the sorting order as ascending (`ASC`, default) or descending (`DESC`).

**Syntax:**

```
sql
Copy code
SELECT column1, column2, ...
FROM table_name
ORDER BY column1 [ASC|DESC], column2 [ASC|DESC], ...;
```

```
]: import sqlite3

# Connect to the SQLite database (creates it if it doesn't exist)
conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Create the 'users' table with sample data
cursor.execute('''
CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    age INTEGER NOT NULL,
    email TEXT NOT NULL UNIQUE
)
''')

# Sample data insertion (clear table first)
cursor.execute("DELETE FROM users")  # Clear previous data
sample_users = [
    (1, 'Alice', 30, 'alice@example.com'),
    (2, 'Bob', 25, 'bob@example.com'),
    (3, 'Charlie', 35, 'charlie@example.com'),
    (4, 'David', 30, 'david@example.com'),
    (5, 'Eve', 40, 'eve@example.com')
]
cursor.executemany("INSERT INTO users (id, name, age, email) VALUES (?, ?, ?, ?)", sample_users)
conn.commit()

# Query using WHERE, GROUP BY, and ORDER BY
query = '''
SELECT age, COUNT(*) AS user_count
FROM users
WHERE age > 25
GROUP BY age
ORDER BY user_count DESC;
'''

# Execute the query and fetch results
cursor.execute(query)
results = cursor.fetchall()

# Display the results
print("Age Group and User Count (Age > 25):")
for row in results:
    print(f"Age: {row[0]}, User Count: {row[1]}")

# Close the connection
conn.close()
```

```
Age Group and User Count (Age > 25):
Age: 30, User Count: 2
Age: 40, User Count: 1
Age: 35, User Count: 1
```

## Conclusion

In this way we have studied different clauses like where , order by and group by.

| **Experiment No: 5** | | |
|---|---|---|
| **Aim:** SQLite Transactions and Atomic Operations<br>• Implement transactions to ensure data integrity.<br>• Perform commit and rollback operations.<br>• Demonstrate atomic operations in SQLite. | | |
| | | |

**Objective:** To Perform the SQlite Transaction and Atomic operation

**Outcome:** Student will able to understand the SQlite Transaction and Atomic operation

**Pre-requisite:** Computer with  anaconda installed

**Theory:**

SQLite transactions and atomic operations are crucial for ensuring data integrity, especially in multi-user environments or when multiple operations must be completed successfully. Here's an overview of how transactions work in SQLite and how to implement them using Python.

## What is a Transaction?

A transaction is a sequence of one or more SQL operations that are executed as a single unit. Transactions ensure that all operations within the transaction are completed successfully before committing any changes to the database. If any operation fails, the entire transaction can be rolled back, leaving the database in a consistent state.

## ACID Properties

Transactions in SQLite adhere to the ACID properties:

- **Atomicity**: Ensures that all operations in a transaction are completed successfully. If any operation fails, the transaction is aborted, and the database state is unchanged.
- **Consistency**: Ensures that the database transitions from one valid state to another valid state.
- **Isolation**: Ensures that transactions do not interfere with each other. Each transaction operates independently until it's completed.
- **Durability**: Once a transaction is committed, its changes are permanent, even in the event of a system failure.

## Using Transactions in SQLite

In SQLite, transactions can be managed using the following commands:

- `BEGIN TRANSACTION`: Starts a new transaction.
- `COMMIT`: Saves all changes made in the transaction.
- `ROLLBACK`: Undoes all changes made in the transaction if an error occurs.

.

```python
import sqlite3

# Connect to the SQLite database (creates it if it doesn't exist)
conn = sqlite3.connect('banking_example.db')
cursor = conn.cursor()

# Create the 'accounts' table
cursor.execute('''
CREATE TABLE IF NOT EXISTS accounts (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    balance REAL NOT NULL CHECK (balance >= 0)
)
''')

# Function to create sample accounts
def create_accounts():
    cursor.execute("INSERT INTO accounts (name, balance) VALUES (?, ?)", ('Alice', 1000.0))
    cursor.execute("INSERT INTO accounts (name, balance) VALUES (?, ?)", ('Bob', 500.0))
    conn.commit()

# Function to transfer money between accounts
def transfer_money(sender_id, receiver_id, amount):
    try:
        # Begin a transaction
        conn.execute('BEGIN TRANSACTION')

        # Check balances
        cursor.execute("SELECT balance FROM accounts WHERE id = ?", (sender_id,))
        sender_balance = cursor.fetchone()[0]

        if sender_balance < amount:
            raise ValueError("Insufficient funds")

        # Deduct amount from sender
        cursor.execute("UPDATE accounts SET balance = balance - ? WHERE id = ?", (amount, sender_id))

        # Add amount to receiver
        cursor.execute("UPDATE accounts SET balance = balance + ? WHERE id = ?", (amount, receiver_id))

        # Commit the transaction
        conn.commit()
        print(f"Transferred ${amount} from account {sender_id} to account {receiver_id}.")
```

```python
    except ValueError as ve:
        # Rollback in case of insufficient funds
        conn.rollback()
        print(f"Transaction failed: {ve}. Transaction rolled back.")

    except Exception as e:
        # Rollback for any other error
        conn.rollback()
        print(f"An error occurred: {e}. Transaction rolled back.")

# Create accounts only if the database is empty
if cursor.execute("SELECT COUNT(*) FROM accounts").fetchone()[0] == 0:
    create_accounts()

# Sample transfer operations
transfer_money(1, 2, 200)  # Transfer $200 from Alice to Bob
transfer_money(1, 2, 900)  # Attempt to transfer $900 (should fail)

# Display account balances
print("\nAccount Balances:")
cursor.execute("SELECT * FROM accounts")
for row in cursor.fetchall():
    print(f"Account ID: {row[0]}, Name: {row[1]}, Balance: ${row[2]:.2f}")

# Close the connection
conn.close()
```

```
Transferred $200 from account 1 to account 2.
Transaction failed: Insufficient funds. Transaction rolled back.

Account Balances:
Account ID: 1, Name: Alice, Balance: $800.00
Account ID: 2, Name: Bob, Balance: $700.00
```

**Conclusion**

In this way we have studied different transactional operation

| **Experiment No: 6** | | |
|---|---|---|
| **Aim:** Indexing and Performance Optimization<br>• Create indexes on tables to improve query performance.<br>• Analyze the performance of indexed vs. non-indexed queries.<br>• Use EXPLAIN QUERY PLAN to understand query execution plans. | | |
| | | |

**Objective:** The objective of creating indexes on tables to improve performance

**Outcome:** Student will able to understand indexes on query performance

**Pre-requisite:** Computer with anaconda installed

Theory:

Creating indexes on tables is a common technique used to improve query performance in SQL databases, including SQLite. Indexes allow the database to find rows more quickly than scanning the entire table.

Here's a brief overview of how to create indexes and a Python program that demonstrates how to create indexes on a SQLite database.

**What is an Index?**

An **index** is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional space and potential overhead for data modification operations (inserts, updates, deletes).

**Benefits of Indexes**

- **Faster Query Performance**: Indexes help the database engine find rows much faster, especially for large tables.
- **Efficient Sorting**: Indexes can speed up sorting operations for `ORDER BY` clauses.
- **Improved Search**: Indexes enhance the performance of searches involving `WHERE` clauses.

**Types of Indexes**

- **Unique Index**: Ensures that all values in the indexed column are unique.
- **Composite Index**: An index on multiple columns.
- **Full-text Index**: Used for searching text in large datasets (specific to certain databases).

1]:
```python
import sqlite3
import time

# Connect to the SQLite database (creates it if it doesn't exist)
conn = sqlite3.connect('index_example.db')
cursor = conn.cursor()

# Create the 'users' table
cursor.execute('''
CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    email TEXT NOT NULL UNIQUE,
    age INTEGER NOT NULL
)
''')

# Insert sample data
def insert_users(num_users):
    for i in range(num_users):
        cursor.execute("INSERT INTO users (name, email, age) VALUES (?, ?, ?)",
                       (f'User{i}', f'user{i}@example.com', 20 + (i % 10)))
    conn.commit()

# Create indexes
def create_indexes():
    # Create an index on the 'email' column
    cursor.execute("CREATE INDEX IF NOT EXISTS idx_email ON users (email)")

    # Create an index on the 'age' column
    cursor.execute("CREATE INDEX IF NOT EXISTS idx_age ON users (age)")

    conn.commit()
    print("Indexes created successfully.")

# Measure query performance
def query_performance():
    # Query without index
    start_time = time.time()
    cursor.execute("SELECT * FROM users WHERE email = 'user100@example.com'")
    print(f"Query without index took: {time.time() - start_time:.6f} seconds")

    # Query with index
    start_time = time.time()
    cursor.execute("SELECT * FROM users WHERE age = 25")
    print(f"Query with index took: {time.time() - start_time:.6f} seconds")
```

```python
# Measure query performance
def query_performance():
    # Query without index
    start_time = time.time()
    cursor.execute("SELECT * FROM users WHERE email = 'user100@example.com'")
    print(f"Query without index took: {time.time() - start_time:.6f} seconds")

    # Query with index
    start_time = time.time()
    cursor.execute("SELECT * FROM users WHERE age = 25")
    print(f"Query with index took: {time.time() - start_time:.6f} seconds")

# Main execution
insert_users(1000)  # Insert 1000 users
create_indexes()     # Create indexes
query_performance()  # Measure query performance

# Display all users
print("\nAll users:")
cursor.execute("SELECT * FROM users")
for row in cursor.fetchall():
    print(row)

# Close the connection
conn.close()
```

```
Indexes created successfully.
Query without index took: 0.000000 seconds
Query with index took: 0.001002 seconds

All users:
(1, 'User0', 'user0@example.com', 20)
(2, 'User1', 'user1@example.com', 21)
(3, 'User2', 'user2@example.com', 22)
(4, 'User3', 'user3@example.com', 23)
(5, 'User4', 'user4@example.com', 24)
(6, 'User5', 'user5@example.com', 25)
(7, 'User6', 'user6@example.com', 26)
(8, 'User7', 'user7@example.com', 27)
(9, 'User8', 'user8@example.com', 28)
(10, 'User9', 'user9@example.com', 29)
(11, 'User10', 'user10@example.com', 20)
(12, 'User11', 'user11@example.com', 21)
(13, 'User12', 'user12@example.com', 22)
```

**Conclusion**

In this way we have studied different indexing operation

| Experiment No: 7 | | |
|---|---|---|
| **Aim**: Working with Triggers and Views • Create and use triggers for automated database operations. • Define and query views to simplify complex queries. • Explore the use cases for triggers and views in database applications. | | |
| | | |

**Objective:** The objective of this assignment is to learn the concepts of user defined packages in Java.

**Outcome:** Student will able to understand how to create trigger in database

**Pre-requisite:** Computer with anaconda installed

**Theory:**

## . What is a Trigger?

A **trigger** in SQL is a special type of stored procedure that automatically executes or fires when specific events occur in a database table. Triggers are useful for enforcing business rules, maintaining audit trails, and automating certain actions without the need for application logic.

## Types of Triggers

Triggers can be classified based on when they are executed:

1. **BEFORE Triggers**: Executed before an insert, update, or delete operation.
2. **AFTER Triggers**: Executed after an insert, update, or delete operation.
3. **INSTEAD OF Triggers**: Used primarily with views to perform an action instead of the triggering operation.

## Common Use Cases

- **Data Validation**: Ensuring that data being inserted or updated meets certain criteria.
- **Cascading Changes**: Automatically updating or deleting related records in other tables.
- **Audit Trails**: Keeping track of changes by logging old and new values whenever a record is modified.
- **Enforcing Business Rules**: Automatically enforcing rules like restrictions on the types of values that can be inserted.

## Syntax for Creating Triggers

The general syntax for creating a trigger in SQLite is as follows:

```
sql
Copy code
CREATE TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF} {INSERT | UPDATE | DELETE}
ON table_name
FOR EACH ROW
BEGIN
    -- Trigger action (SQL statements)
END;
```

## Example of a Trigger

Let's look at an example of a trigger that maintains an audit trail in a SQLite database.

In [2]:
```
import sqlite3

# Connect to the SQLite database
conn = sqlite3.connect('trigger_example.db')
cursor = conn.cursor()

# Create the 'users' table
cursor.execute('''
CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    email TEXT NOT NULL UNIQUE
)
''')

# Create the 'user_audit' table to log changes
cursor.execute('''
CREATE TABLE IF NOT EXISTS user_audit (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER,
    old_email TEXT,
    new_email TEXT,
    change_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
)
''')

# Create a trigger to log email changes
cursor.execute('''
CREATE TRIGGER log_email_change
AFTER UPDATE OF email ON users
FOR EACH ROW
BEGIN
    INSERT INTO user_audit (user_id, old_email, new_email)
    VALUES (OLD.id, OLD.email, NEW.email);
END;
''')

# Function to update a user's email
def update_user_email(user_id, new_email):
    cursor.execute("UPDATE users SET email = ? WHERE id = ?", (new_email, user_id))
    conn.commit()
    print(f"Updated user {user_id} email to {new_email}.")

# Insert a sample user
cursor.execute("INSERT INTO users (name, email) VALUES (?, ?)", ('Alice', 'alice@example.com'))
conn.commit()
```

```python
# Function to update a user's email
def update_user_email(user_id, new_email):
    cursor.execute("UPDATE users SET email = ? WHERE id = ?", (new_email, user_id))
    conn.commit()
    print(f"Updated user {user_id} email to {new_email}.")

# Insert a sample user
cursor.execute("INSERT INTO users (name, email) VALUES (?, ?)", ('Alice', 'alice@example.com'))
conn.commit()

# Update the user's email, which will trigger the logging
update_user_email(1, 'alice_new@example.com')

# Fetch and display the audit log
print("\nAudit Log:")
cursor.execute("SELECT * FROM user_audit")
for row in cursor.fetchall():
    print(row)

# Close the connection
conn.close()
```

```
Updated user 1 email to alice_new@example.com.

Audit Log:
(1, 1, 'alice@example.com', 'alice_new@example.com', '2024-10-04 11:15:02')
```

## Conclusion

In this way we have studied different triggered operation

| **Experiment No: 8** | | |
|---|---|---|
| **Aim**: Integrating SQLite with Programming Languages 10 • Integrate SQLite with a programming language like Python or Java. • Perform CRUD operations through the programming language. • Develop a simple application that uses SQLite for data storage | | |
| | | |

**Objective: To** Perform CRUD operations through the programming language

**Outcome:** Student will able to understand about CRUD operation

**Pre-requisite:** Computer with anaconda installed Theory

## Examples of CRUD Operations on SQLite Database

CRUD operations are fundamental to interacting with databases in software development. Below are practical examples of CRUD operations in a relational database context, using SQL (Structured Query Language), which is commonly used for managing relational databases. These examples provide a glimpse into how each operation is implemented in real-world scenarios.

### 1. Create

The "Create" operation involves adding new records to a database. In SQL, this is typically done using the `INSERT` statement. For example, if you have a table named `users` that stores user information, you can add a new user like this:

*INSERT INTO users (username, email, password) VALUES ('gfg', 'gfg@example.com', 'Geeks124124124');*

This SQL command inserts a new row into the `users` table with the specified username, email, and password.

### 2. Read

The "Read" operation retrieves data from the database. This can be achieved using the `SELECT` statement in SQL. For example, to retrieve the information of all users from the `users` table:

```
SELECT * FROM users;
```

### 3. Update

The "Update" operation modifies existing data within the database. In SQL, this is accomplished with the `UPDATE` statement. For instance, if you want to update the email address of a user in the `users` table, you can do so by:

```
UPDATE users SET email = 'abc@example.com' WHERE username = 'gfg';
```

### 4. Delete

The "Delete" operation removes records from the database. In SQL, this is done using the `DELETE` statement. For example, to delete a user from the `users` table:

```
DELETE FROM users WHERE username = 'gfg';
```

The concept of CRUD operations is a pivotal one in the field of software development, underpinning the interaction between applications and databases. By mastering these four basic operations, developers can ensure efficient data management, facilitating the creation, retrieval, modification, and deletion of data. Whether you are developing a simple application or a complex enterprise system, understanding CRUD operations is essential for effective database interaction and application functionality.

## Output:

In [*]:

```
import sqlite3

# Function to initialize the database and create the contacts table
def initialize_db():
    conn = sqlite3.connect('contacts.db')
    cursor = conn.cursor()

    cursor.execute('''
    CREATE TABLE IF NOT EXISTS contacts (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT NOT NULL,
        phone TEXT NOT NULL
    )
    ''')

    conn.commit()
    conn.close()

# Function to create a new contact
def create_contact(name, phone):
    conn = sqlite3.connect('contacts.db')
    cursor = conn.cursor()

    cursor.execute('INSERT INTO contacts (name, phone) VALUES (?, ?)', (name, phone))

    conn.commit()
    conn.close()
    print(f'Contact added: {name} - {phone}')

# Function to read and display all contacts
def read_contacts():
    conn = sqlite3.connect('contacts.db')
    cursor = conn.cursor()

    cursor.execute('SELECT * FROM contacts')
    contacts = cursor.fetchall()

    conn.close()
    return contacts

# Function to update a contact
def update_contact(contact_id, name, phone):
    conn = sqlite3.connect('contacts.db')
    cursor = conn.cursor()

    cursor.execute('UPDATE contacts SET name = ?, phone = ? WHERE id = ?', (name, phone, contact_id))
```

```
    print(f"Contact with ID {contact_id} deleted.")

# Main function to interact with the user
def main():
    initialize_db()  # Ensure the database and table are created

    while True:
        print("\nContact Manager")
        print("1. Add Contact")
        print("2. View Contacts")
        print("3. Update Contact")
        print("4. Delete Contact")
        print("5. Exit")

        choice = input("Choose an option: ")

        if choice == '1':
            name = input("Enter name: ")
            phone = input("Enter phone: ")
            create_contact(name, phone)

        elif choice == '2':
            contacts = read_contacts()
            if contacts:
                print("\nContacts List:")
                for contact in contacts:
                    print(f"{contact[0]}. {contact[1]} - {contact[2]}")
            else:
                print("No contacts found.")

        elif choice == '3':
            contact_id = int(input("Enter contact ID to update: "))
            name = input("Enter new name: ")
            phone = input("Enter new phone: ")
            update_contact(contact_id, name, phone)

        elif choice == '4':
            contact_id = int(input("Enter contact ID to delete: "))
            delete_contact(contact_id)

        elif choice == '5':
            print("Exiting the Contact Manager.")
            break

        else:
            print("Invalid option! Please choose again.")

if __name__ == "__main__":
    main()
```

```
Contact Manager
1. Add Contact
2. View Contacts
3. Update Contact
4. Delete Contact
5. Exit
```

## Conclusion

In this way we have studied different CRUD Operation

| Experiment No: 9 | | |
|---|---|---|
| **Aim :** : Setting Up a Firebase Project<br>• Create and configure a Firebase project.<br>• Connect a mobile or web application to Firebase.<br>• Explore the Firebase console and its features. | | |
| | | |

**AIM:** To Create and Configure Firebase project and Connect a mobile or web application to Firebase

**OBJECTIVES:** Student will able to understand about joining of web application and firebase project

**APPRATUS: Computer with firebase installed**

**THEORY:**

Firebase is a platform developed by Google for creating mobile and web applications. It provides a variety of tools and services to help developers build high-quality applications quickly and efficiently. Here's an overview of the key components and features of Firebase:

## Key Features

1. **Realtime Database**:
   - A cloud-hosted NoSQL database that allows you to store and sync data in real-time across all clients.
   - Suitable for applications requiring live updates, such as chat applications.
2. **Firestore**:
   - A flexible, scalable database for mobile, web, and server development.
   - Supports more complex querying and better indexing compared to the Realtime Database.
   - Allows hierarchical data structures.
3. **Authentication**:
   - Provides easy-to-use authentication methods, including email/password, phone authentication, and federated identity providers (like Google, Facebook, and Twitter).
   - Simplifies user management and ensures secure access.
4. **Cloud Functions**:
   - Serverless functions that run in response to events triggered by Firebase features and HTTPS requests.
   - Allows you to execute backend code without managing servers.
5. **Hosting**:
   - Fast and secure hosting for web apps and static content.
   - Supports HTTPS and easy deployment via the Firebase CLI.
6. **Cloud Storage**:
   - A powerful, simple, and cost-effective object storage solution for storing user-generated content like photos and videos.
   - Integrated with Firebase Authentication for secure access.
7. **Analytics**:

    - Provides insights into user behavior and app performance.
    - Allows tracking of user engagement and conversion rates with Google Analytics integration.
8. **Crashlytics**:
    - A lightweight, real-time crash reporting tool that helps you track and fix crashes in your app.
    - Provides detailed information about crashes and issues in your application.
9. **Performance Monitoring**:
    - Helps you understand your app's performance from the user's perspective.
    - Offers insights into app startup time, HTTP requests, and more.
10. **Remote Config**:
    - Allows you to change the appearance and behavior of your app without requiring users to download an update.
    - You can create customizable experiences based on user segments.

## Advantages of Using Firebase

- **Quick Development**: Firebase provides out-of-the-box solutions, allowing developers to focus on building features rather than backend infrastructure.
- **Scalability**: It can handle applications of all sizes, from startups to large-scale enterprise applications.
- **Cross-Platform Support**: Supports web, iOS, Android, and Unity applications.
- **Community and Documentation**: Extensive documentation, tutorials, and a strong community support system help developers easily get started.

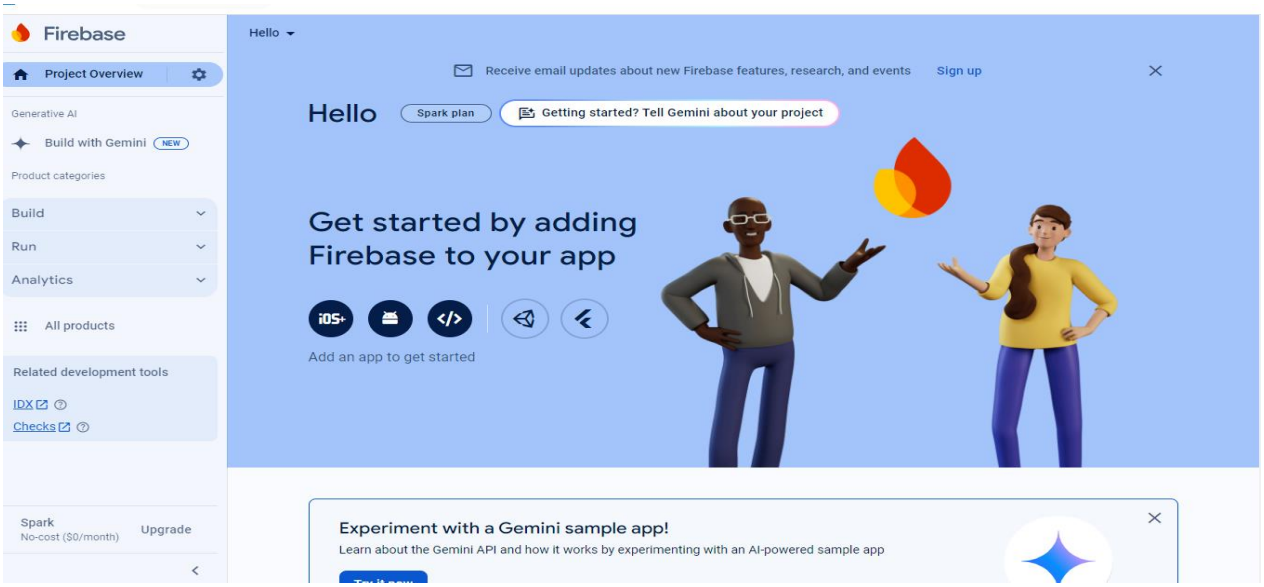
## Use Cases

- **Chat Applications**: Real-time communication features with Firestore or Realtime Database.
- **E-commerce**: Manage user authentication, inventory, and orders seamlessly.
- **Social Media**: Handle user data, posts, likes, and comments in real time.
- **IoT Applications**: Manage and visualize data from connected devices.

## Getting Started

To get started with Firebase:

1. **Create a Firebase Project**: Use the Firebase Console to create a new project.
2. **Add Firebase to Your App**: Include Firebase SDKs in your mobile or web application.
3. **Choose Services**: Decide which Firebase services you want to use (e.g., Firestore, Authentication).
4. **Develop Your Application**: Implement features using Firebase's APIs.




**CONCLUSIONS:**

In This way studied the firebase project

| **Experiment No: 10** | | |
|---|---|---|
| Firebase Authentication and Security Rules<br>• Set up Firebase Authentication for user management.<br>• Implement email/password and social media logins.<br>• Define and apply Firebase security rules to protect data. | | |
| | | |

**AIM: To create rules firebase authentication and user management** .

**OBJECTIVES:** To study concept of Set up Firebase Authentication for user management

**APPRATUS: Computer with firebase installed**

## THEORY:

Firebase provides a flexible set of authentication rules to manage user access and data security. Here's an overview of how you can set up authentication rules and best practices:

### 1. Types of Authentication

Firebase supports several authentication methods, including:

- **Email/Password**
- **Google, Facebook, Twitter, GitHub Sign-In**
- **Anonymous Authentication**
- **Phone Authentication**

### 2. Setting Up Authentication

To set up authentication:

- Go to the Firebase Console.
- Navigate to the "Authentication" section.
- Enable the desired sign-in methods.

### 3. Security Rules

Firebase Security Rules work with Firestore and Realtime Database to define how your data can be accessed based on user authentication.

#### *Key Components:*

- **Authentication State**: Check if a user is authenticated (`request.auth`).
- **User ID**: Access user properties via `request.auth.uid`.
- **Role-Based Access Control**: Implement roles to manage permissions.

### 4. Example Rules

#### *Firestore Rules:*

```
javascript
Copy code
service cloud.firestore {
  match /databases/{database}/documents {
    match /users/{userId} {
      allow read, write: if request.auth != null && request.auth.uid == userId;
    }

    match /publicData/{document} {
      allow read: if true; // Anyone can read public data
      allow write: if request.auth != null && request.auth.token.admin == true; // Only
admins can write
    }
  }
}
```

#### *Realtime Database Rules:*

```
json
Copy code
{
```

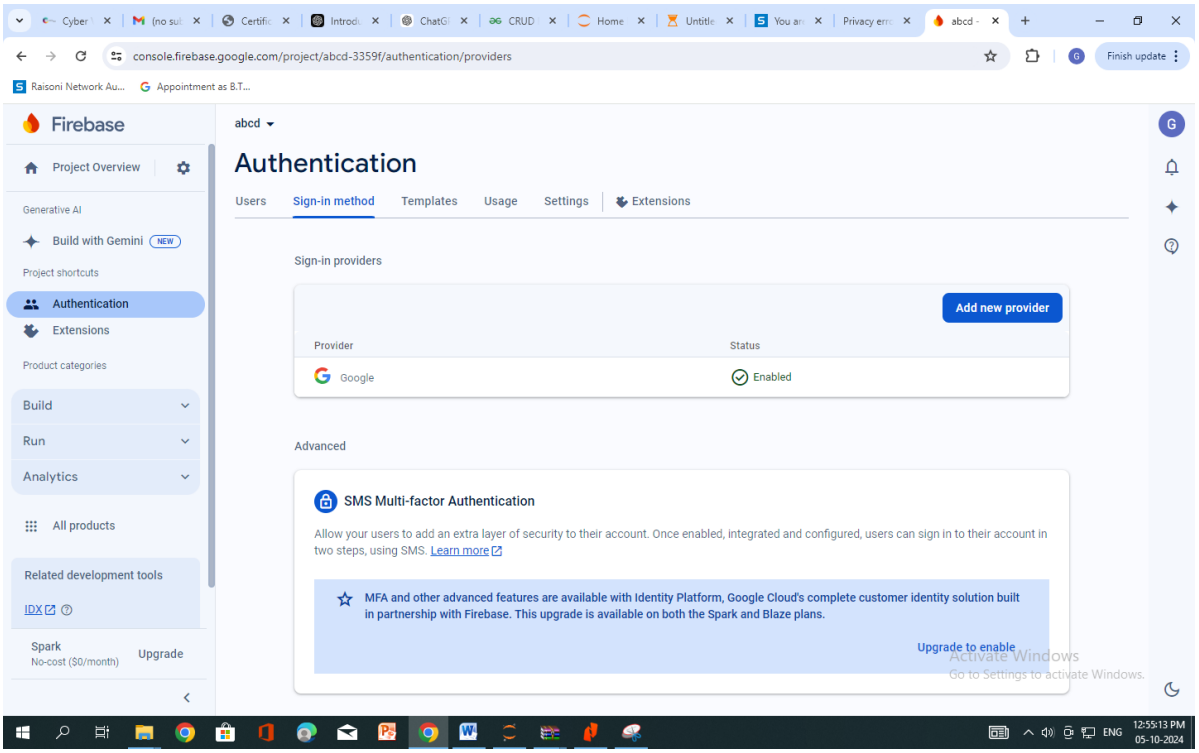
```
  "rules": {
    ".read": "auth != null",
    ".write": "auth != null",

    "users": {
      "$uid": {
        ".read": "$uid === auth.uid",
        ".write": "$uid === auth.uid"
      }
    },
    "publicData": {
      ".read": "true", // Publicly readable
      ".write": "auth != null && auth.token.admin === true" // Admin write access
    }
  }
}
```

## 5. Best Practices

- **Use Custom Claims**: For role management, you can add custom claims to users, like `admin`.
- **Minimize Read/Write Access**: Grant the least privilege needed for users to perform their tasks.
- **Test Your Rules**: Use the Firebase Emulator Suite to test rules locally before deploying.
- **Regularly Review Security Rules**: Update rules as your app evolves to maintain security.

## 6. Monitoring and Logging

Use Firebase's built-in monitoring tools to track authentication events and access patterns. This helps in identifying any potential security issues.




Conclusion : In This way We have studied firebase

\

v