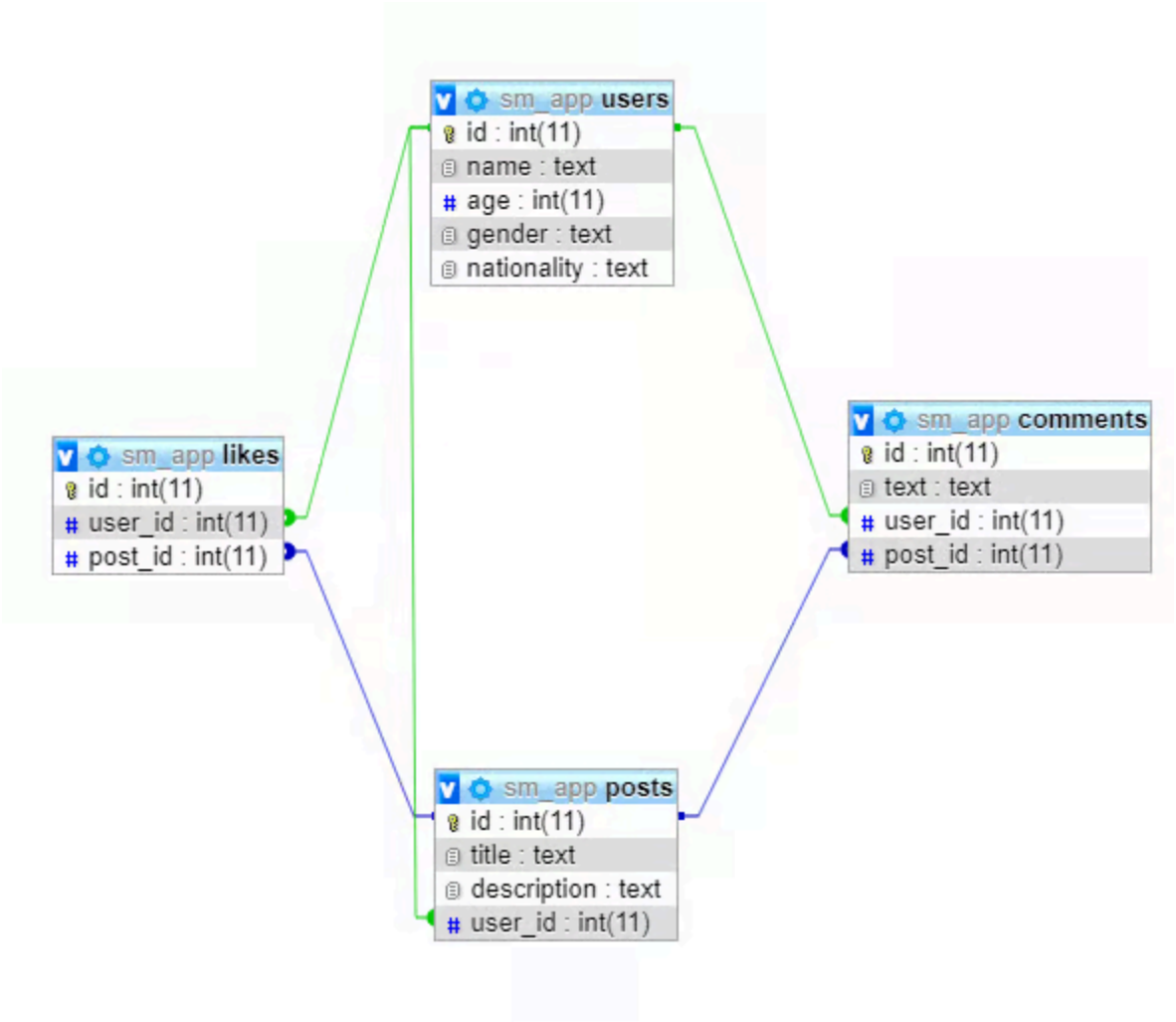
# 15 SQL CRUD Operations

## Understanding the Database Schema

In the example the database will consist of four tables:

1. `users`
2. `posts`
3. `comments`
4. `likes`




Both `users` and `posts` will have a **one-to-many relationship** since one user can like many posts. Similarly, one user can post many comments, and one post can also have multiple comments. So, both `users` and `posts` will also have one-to-many relationships with the `comments` table. This also applies to the `likes` table, so both `users` and `posts` will have a one-to-many relationship with the `likes` table.

## Using Python SQL Library to Connect to a Database

Before you interact with any database through a Python SQL Library, you have to **connect** to that database.

By default, your Python installation contains a Python SQL library named `sqlite3` that you can use to interact with an SQLite database.

SQLite databases are **serverless** and **self-contained**, since they read and write data to a file. This means that, unlike with MySQL and PostgreSQL, you don't even need to install and run an SQLite server to perform database operations.

Here's how you use `sqlite3` to connect to an SQLite database

```
import sqlite3
from sqlite3 import Error

def create_connection(path):
    connection = None
    try:
        connection = sqlite3.connect(path)
        print("Connection to SQLite DB successful")
    except Error as e:
        print(f"The error '{e}' occurred")

    return connection
```

`sqlite3.connect(path)` returns a `connection` object, which is in turn returned by `create_connection()`. This `connection` object can be used to execute queries on an SQLite database.

```
connection = create_connection("./sampledb.sqlite")
```

Once you execute the above script, you'll see that a database file `sm_app.sqlite` is created in the root directory. Note that you can change the location to match your setup.

## Creating Tables

As discussed earlier, you'll create four tables:

1. `users`
2. `posts`
3. `comments`
4. `likes`

   Define a function `execute_query()` that uses cursor.execute() method. Your function will

accept the `connection` object and a query string, which you'll pass to `cursor.execute()`

```
def execute_query(connection, query):
        cursor = connection.cursor()
        try:
                cursor.execute(query)
                connection.commit()
                print("Query executed successfully")
        except Error as e:
                print(f"The error `{e}` occurred")
```

This code tries to execute the given `query` and prints an error message if necessary.

Let's create tables

```
# Query to create users table with id, name, age, gender and nationality
create_users_table = """
CREATE TABLE IF NOT EXISTS users (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT NOT NULL,
        age INTEGER,
        gender TEXT,
        nationality TEXT
);
"""

# create users table by executing the query
execute_query(connection, create_users_table)
```

The following query is used to create the `posts` table

```
create_posts_table = """
CREATE TABLE IF NOT EXISTS posts(
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  title TEXT NOT NULL,
  description TEXT NOT NULL,
  user_id INTEGER NOT NULL,
  FOREIGN KEY (user_id) REFERENCES users (id)
);
"""

execute_query(connection, create_posts_table)
```

Since there's a one-to-many relationship between `users` and `posts`, you can see a foreign key `user_id` in the `posts` table that references the `id` column in the `users` table.

Create the `comments` and `likes` tables

```
create_comments_table = """
CREATE TABLE IF NOT EXISTS comments (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  text TEXT NOT NULL,
  user_id INTEGER NOT NULL,
  post_id INTEGER NOT NULL,
  FOREIGN KEY (user_id) REFERENCES users (id) FOREIGN KEY (post_id)
REFERENCES posts (id)
);
"""

create_likes_table = """
CREATE TABLE IF NOT EXISTS likes (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  user_id INTEGER NOT NULL,
  post_id integer NOT NULL,
  FOREIGN KEY (user_id) REFERENCES users (id) FOREIGN KEY (post_id)
REFERENCES posts (id)
);
"""

execute_query(connection, create_comments_table)
execute_query(connection, create_likes_table)
```

## Inserting Records

To insert records into your SQLite database, you can use the same `execute_query()` function that you used to create tables. First, you have to store your `INSERT INTO` query in a string. Then, you can pass the `connection` object and `query` string to `execute_query()`.

```
create_users = """
INSERT INTO
  users (name, age, gender, nationality)
VALUES
  ('James', 25, 'male', 'USA'),
  ('Leila', 32, 'female', 'France'),
  ('Brigitte', 35, 'female', 'England'),
  ('Mike', 40, 'male', 'Denmark'),
  ('Elizabeth', 21, 'female', 'Canada');
```

```
"""

execute_query(connection, create_users)
```

Since you set the `id` column to auto-increment, you don't need to specify the value of the `id` column for these `users`. The `users` table will auto-populate these five records with `id` values from `1` to `5`.

Now insert six records into the `posts` table

```
create_posts = """
INSERT INTO
  posts (title, description, user_id)
VALUES
  ("Happy", "I am feeling very happy today", 1),
  ("Hot Weather", "The weather is very hot today", 2),
  ("Help", "I need some help with my work", 2),
  ("Great News", "I am getting married", 1),
  ("Interesting Game", "It was a fantastic game of tennis", 5),
  ("Party", "Anyone up for a late-night party today?", 3);
"""

execute_query(connection, create_posts)
```

It's important to mention that the `user_id` column of the `posts` table is a **foreign key** that references the `id` column of the `users` table. This means that the `user_id` column must contain a value that **already exists** in the `id` column of the `users` table. If it doesn't exist, then you'll see an error.

Inserts records into the `comments` and `likes` tables

```
create_comments = """
INSERT INTO
  comments (text, user_id, post_id)
VALUES
  ('Count me in', 1, 6),
  ('What sort of help?', 5, 3),
  ('Congrats buddy', 2, 4),
  ('I was rooting for Nadal though', 4, 5),
  ('Help with your thesis?', 2, 3),
  ('Many congratulations', 5, 4);
"""

create_likes = """
```

```
INSERT INTO
  likes (user_id, post_id)
VALUES
  (1, 6),
  (2, 3),
  (1, 5),
  (5, 4),
  (2, 4),
  (4, 2),
  (3, 6);
"""

execute_query(connection, create_comments)
execute_query(connection, create_likes)
```

# Selecting Records

## SELECT

To select records using SQLite, you can again use `cursor.execute()`. However, after you've done this, you'll need to call `.fetchall()`. This method returns a list of tuples where each tuple is mapped to the corresponding row in the retrieved records.

To simplify the process, you can create a function execute_read_query()

```
def execute_read_query(connection, query):
    cursor = connection.cursor()
    result = None
    try:
        cursor.execute(query)
        result = cursor.fetchall()
        return result
    except Error as e:
        print(f"The error '{e}' occurred")
```

This function accepts the `connection` object and the `SELECT` query and returns the selected record.

Let's now select all the records from the `users` table

```
select_users = "SELECT * from users"
users = execute_read_query(connection, select_users)
```

```
for user in users:
    print(user)
```

**Note:** It's not recommended to use `SELECT *` on large tables since it can result in a large number of I/O operations that increase the network traffic.

In the same way, you can retrieve all the records from the `posts` table

```
select_posts = "SELECT * FROM posts"
posts = execute_read_query(connection, select_posts)

for post in posts:
    print(post)
```

## JOIN

You can also execute complex queries involving `JOIN` **operations** to retrieve data from two related tables. For instance, the following script returns the user ids and names, along with the description of the posts that these users posted.

```
select_users_posts = """
SELECT
  users.id,
  users.name,
  posts.description
FROM
  posts
  INNER JOIN users ON users.id = posts.user_id
"""

users_posts = execute_read_query(connection, select_users_posts)

for users_post in users_posts:
    print(users_post)
```

You can also select data from three related tables by implementing **multiple** `JOIN` **operators**. The following script returns all posts, along with the comments on the posts and the names of the users who posted the comments

```
select_posts_comments_users = """
SELECT
  posts.description as post,
  text as comment,
```

```
  name
FROM
  posts
  INNER JOIN comments ON posts.id = comments.post_id
  INNER JOIN users ON users.id = comments.user_id
"""

posts_comments_users = execute_read_query(
    connection, select_posts_comments_users
)

for posts_comments_user in posts_comments_users:
    print(posts_comments_user)
```

You can see from the output that the column names are not being returned by `.fetchall()`. To return column names, you can use the `.description` attribute of the `cursor` object.

```
cursor = connection.cursor()
cursor.execute(select_posts_comments_users)
cursor.fetchall()

column_names = [description[0] for description in cursor.description]
print(column_names)
```

## WHERE

Now you'll execute a `SELECT` query that returns the post, along with the total number of likes that the post received

```
select_post_likes = """
SELECT
  description as Post,
  COUNT(likes.id) as Likes
FROM
  likes,
  posts
WHERE
  posts.id = likes.post_id
GROUP BY
  likes.post_id
"""

post_likes = execute_read_query(connection, select_post_likes)
```

```
for post_like in post_likes:
    print(post_like)
```

## Updating Table Records

Updating records in SQLite is pretty straightforward. You can again make use of `execute_query()`. As an example, you can update the description of the post with an `id` of `2`. First, `SELECT` the description of this post

```
select_post_description = "SELECT description FROM posts WHERE id = 2"

post_description = execute_read_query(connection, select_post_description)

for description in post_description:
    print(description)
```

The following script updates the description

```
update_post_description = """
UPDATE
  posts
SET
  description = "The weather has become pleasant now"
WHERE
  id = 2
"""

execute_query(connection, update_post_description)
```

## Deleting Table Records

You can again use `execute_query()` to delete records from YOUR SQLite database. All you have to do is pass the `connection` object and the string query for the record you want to delete to `execute_query()`. Then, `execute_query()` will create a `cursor` object using the `connection` and pass the string query to `cursor.execute()`, which will delete the records

As an example, try to delete the comment with an `id` of `5`

```
delete_comment = "DELETE FROM comments WHERE id = 5"
execute_query(connection, delete_comment)
```