Objectives

- Creating and Working with Views
- Creating and Working with Indexes
- Working with Triggers

Views:

Introduction:

When retrieving data from a database, we often use joins or nested queries to obtain the desired results. However, this approach can lead to repetitive operations, especially when the same logic is needed for multiple queries. To address this issue, we can use **database views**. A view is a virtual table that is based on the result of a SQL query. It allows we to encapsulate complex queries, including joins and nested queries, into a reusable object. By creating a view, we can simplify our queries, avoid redundancy, and improve maintainability. Instead of rewriting the same logic multiple times, we can simply query the view, making ourr code cleaner and more efficient.

Creating View:

Here’s how we can create a view in SQL:

1. **Decide on the Query**: First, we need to determine the query we want to turn into a virtual table. This query could include joins, filters, aggregations, or any other SQL operations depending on our needs.
2. **Use the `CREATE VIEW` Statement**: Once we have our query ready, we use the `CREATE VIEW` statement to define the view. We start with `CREATE VIEW`, followed by the name we want to give to our new virtual table.
3. **Add the `AS` Keyword**: After specifying the view name, we use the `AS` keyword to link it to our query.
4. **Write the Query**: Finally, we write the query that will define the data for our view. This query will determine what data the view will represent.

Example :

Create a view that stores customer information and their corresponding orders.

```
CREATE VIEW customer_orders AS
SELECT customers.customer_id, customers.customer_name, orders.order_id,
orders.order_date
FROM customers JOIN orders
ON  customers.customer_id = orders.customer_id;
```

After creating a view, we can treat it like a table and query data from it using `SELECT` statements.

```
SELECT * FROM customer_orders;
```

Temporary View:

The view we created earlier is a **permanent view**, meaning it will persist in our database indefinitely and can be reused at any time. However, there is another type of view called a **temporary view**. Unlike permanent views, a temporary view exists only for the duration of the current session. Once the session ends, the temporary view is automatically dropped. Temporary views are particularly useful when we need a short-lived, session-specific representation of data without permanently storing it in the database.
We create a temporary view in the same way as we do for a permanent view, but we add the **`TEMPORARY`** keyword to explicitly indicate that the view is temporary.

Example:

If we want to declare the previous view as a temporary view, it will look like the following:

```
CREATE TEMPORARY VIEW customer_orders AS
SELECT customers.customer_id, customers.customer_name, orders.order_id,
orders.order_date
FROM customers JOIN orders
ON  customers.customer_id = orders.customer_id;
```

Remark:

We can remove views from the database using the **`DROP VIEW`** command. This command permanently deletes the specified view, freeing up resources and ensuring it is no longer accessible.

```
DROP VIEW view_name;
```

Common Table Expression (CTE):

A **Common Table Expression (CTE)** is a special type of temporary result set that exists only for the duration of a single query. Unlike views, CTEs are not stored in the database schema and are discarded once the query execution is complete. To create a CTE, we use the `WITH` keyword instead of the `CREATE VIEW` statement.
A **Common Table Expression (CTE)** helps simplify complex queries by breaking them into smaller, more logical parts. By using CTEs, we can avoid repeating the same subquery multiple times within a query, making the code cleaner, more readable, and easier to maintain.

Example:

Using a Common Table Expression (CTE) for the previous example would look like this:

```
WITH customer_orders AS(
SELECT customers.customer_id, customers.customer_name, orders.order_id,
orders.order_date
FROM customers JOIN orders
ON  customers.customer_id = orders.customer_id)
SELECT * FROM customer_orders;
```

Indexes:

Introduction:

Querying and searching data from a database is typically a **linear process**. This means the database scans through the table row by row to find the required data. For small databases, this approach is acceptable. However, as the database grows larger and starts storing millions of rows, searching for data can become time-consuming, which may slow down your applications.
One way to address this issue is by upgrading to more powerful hardware, such as faster processors and larger storage. However, this solution can be **costly**. Fortunately, SQL provides a more efficient and cost-effective solution: **indexes**.
Indexes in a database are similar to indexes in a book. For example, if you want to find a specific chapter in a book, you can refer to the index, which will direct you to the exact page number where the chapter begins. Similarly, in a database, you can create an index on a specific column. The database management system (DBMS) will then use this index to quickly locate the data, significantly reducing search time.

Creating Indexes:

In SQL, we create an index using the `CREATE INDEX` statement. The syntax is as follows:

1. Start with `CREATE INDEX`, followed by the name you want to assign to the index.
2. Use the `ON` keyword, followed by the name of the table where the index will be created.
3. Specify the column(s) you want to index inside parentheses `()`.

```
CREATE INDEX index_name ON table_name (column_name);
```

we no longer need an index, we can remove it using the `DROP INDEX` statement

```
DROP INDEX index_name;
```

Partial Index:

Partial indexes are special types of indexes that operate only on a subset of rows in a table, rather than the entire table. To create a partial index, we use the same syntax as for a regular index, but we add a `WHERE` clause at the end to specify the condition that defines the subset of rows.

```
CREATE INDEX index_name
ON table_name (column_name)
WHERE condition;
```

Indexes Trade-off:

While indexes provide significant benefits, such as faster query performance, they do not come for free. They introduce several trade-offs that need to be carefully considered.

Space Trade-off:

Just like a book index takes up extra pages, indexes in a database consume additional storage space. Each index creates a separate data structure (e.g., B-tree, hash table) that stores references to the indexed columns. For large tables with multiple indexes, this can result in a significant increase in storage requirements.

Inseting Trade-off:

Inserting data into a table with indexes takes more time compared to a table without indexes. This is because, during an insert operation, the database must update both the table and all associated indexes.

Update Trade-off

Updating data in a table with indexes also incurs additional overhead. When a row is updated, the database must update both the table and any indexes that include the modified columns.

Deletion Trade-off:

Deleting data from a table with indexes requires the database to remove the corresponding entries from all associated indexes. This adds extra work to the deletion process.

VACUUM:

When deleting data from a database, the space that the data used to occupy is not immediately freed. Instead, it is marked as **unused space**. This means that future insertions can reuse this space, but it is not returned to the operating system or made available for other purposes.
To address this issue and **free up the unused space**, you can use the `VACUUM` command.

Triggers:

Introduction:

Triggers in SQLite are a powerful way to automate actions in a database based on specific events. These events include **inserting**, **updating**, or **deleting** data in a table. When creating a trigger, you need to specify **when** the trigger should execute: **BEFORE**, **AFTER**, or **INSTEAD OF** the event.

- **BEFORE**: The trigger runs before the event occurs. This is useful for validating or modifying data before it is committed to the database.
- **AFTER**: The trigger runs after the event has occurred. This is commonly used for logging, auditing, or cascading changes.
- **INSTEAD OF**: The trigger replaces the event entirely. This is typically used with views to handle operations that would otherwise not be allowed.

Creating Trigger:

To create a trigger in SQLite, follow these steps:

1. Start with the **`CREATE TRIGGER`** keyword.
2. Specify the **name of the trigger** after `CREATE TRIGGER`.
3. Define **when the trigger should execute** using one of the following:
    - **`BEFORE`**: The trigger runs before the event.
    - **`AFTER`**: The trigger runs after the event.

   - `INSTEAD OF` : The trigger replaces the event (typically used with views).
4. Specify the **event** that activates the trigger:
   - `INSERT` : Triggered when a new row is inserted.
   - `UPDATE` : Triggered when a row is updated.
   - `DELETE` : Triggered when a row is deleted.
5. Specify the **table name** on which the trigger should be applied.
6. Use the `FOR EACH ROW` clause to indicate that the trigger should execute for each affected row.
7. Enclose the trigger's logic within the `BEGIN` **and** `END` keywords. Inside this block, write the statements that should run when the trigger is activated.

```
CREATE TRIGGER trigger_name
[AFTER|BEFORE| INSTEAD OF] UPDATE ON table_name
FOR EACH ROW
BEGIN
    -- Here our statements
END;
```

Before Trigger:

We use a `BEFORE` **trigger** when we want to perform an action in the database **before** the event that activates the trigger is completed.This is useful for tasks like data validation, modification, or enforcing business rules before the actual operation (e.g., `INSERT` , `UPDATE` , or `DELETE`) takes place.

Example:

Suppose we want to ensure that a user's email is always in lowercase before it is inserted or updated in the `users` table. We can use a `BEFORE` trigger to modify the email value before the operation is completed.

```
CREATE TRIGGER validate_user_email
BEFORE INSERT OR UPDATE ON users
FOR EACH ROW
BEGIN
    -- Convert the email to lowercase before insertion or update
    SET NEW.email = LOWER(NEW.email);
END;
```

Remark:

- **`NEW`**: Used to access the updated values of the row **after** the update.

AFTER Triggers:

We use an **`AFTER` trigger** when we want to perform an action in the database **after** the event that activates the trigger is completed. This is useful for tasks like logging, auditing, or cascading changes after the actual operation (e.g., `INSERT`, `UPDATE`, or `DELETE`) takes place.

Example:

Suppose we want to log changes to the `users` table after an update operation. We can use an `AFTER` trigger to record the old and new values of the updated data.

```
CREATE TRIGGER log_user_updates
AFTER UPDATE ON users
FOR EACH ROW
BEGIN
    -- Log the changes to an audit table
    INSERT INTO audit_log (user_id, old_email, new_email, change_date)
    VALUES (OLD.id, OLD.email, NEW.email, datetime('now'));
END;
```

Remark:

- **`OLD`**: Used to access the original values of the row **before** the update.

INSTEAD OF Triggers:

We use an **`INSTEAD OF` trigger** when we want to replace the event that activates the trigger entirely. This is typically used with **views** to handle operations that would otherwise not be allowed (e.g., inserting into a view that joins multiple tables).

Example:

Suppose we have a view called `user_view` that combines data from multiple tables. We can use an `INSTEAD OF` trigger to handle `INSERT` operations on the view.

```
CREATE TRIGGER handle_user_insert
INSTEAD OF INSERT ON user_view
FOR EACH ROW
```

```
BEGIN
    -- Insert the new user into the actual users table
    INSERT INTO users (id, email, created_at)
    VALUES (NEW.id, NEW.email, datetime('now'));

    -- Log the insert activity
    INSERT INTO activity_log (user_id, activity_type, new_email, change_date)
    VALUES (NEW.id, 'ACCOUNT_INSERT', NEW.email, datetime('now'));
END;
```

Deleting Trigger:

To delete a trigger in SQLite, we use the `DROP TRIGGER` statement, similar to how we delete tables, indexes, or views. The syntax is as follows:

```
DROP TRIGGER trigger_name;
```

Task:

Using everything you’ve learned in this course, create a database to solve a real-world problem. Follow these steps:

1. **Design the ERD (Entity-Relationship Diagram)** using **Mermaid** syntax to visualize the database structure, including entities, attributes, and relationships.
2. **Create the tables** with appropriate constraints (e.g., primary keys, foreign keys, unique, not null) and define relationships between them.
3. **Write basic `INSERT` and `SELECT` queries** to populate and retrieve data from the database.
4. **Create views** to simplify complex queries and provide meaningful insights into the data.
5. **Use triggers** where necessary to automate tasks (e.g., logging changes, enforcing business rules).
6. **Create indexes** to optimize queries that require improved performance.