Transaction Control Language (TCL) in SQL

Transaction Control Language (TCL) manages transactions in SQL and allows the user to control and handle multiple statements as a single unit of work. TCL commands are essential to ensure data consistency and allow rollback or commit operations.

Key concepts in TCL:

- **Transaction**: A transaction is a unit of work that contains multiple SQL queries and ensures that the database remains consistent. Either all the queries are executed, or none are.
- **TCL Commands**:
  1. **START TRANSACTION**: Begins a new transaction.
  2. **SAVEPOINT**: Marks a point within a transaction to which you can later roll back.
  3. **ROLLBACK**: Reverts the database to the last savepoint or the start of the transaction, undoing changes made after that point.
  4. **COMMIT**: Finalizes the transaction, saving all changes made during the transaction to the database permanently.

Detailed Breakdown of Each TCL Command

1. START TRANSACTION

- This command initiates a new transaction. Once you start a transaction, all subsequent queries run under that transaction until you either commit or rollback.

```
START TRANSACTION;
```

- After `START TRANSACTION`, any SQL operations performed remain pending (uncommitted) until explicitly committed or rolled back.

2. SAVEPOINT

- The `SAVEPOINT` command is used to define a specific point within a transaction that you can rollback to. It allows partial rollback without affecting the entire transaction.

```
SAVEPOINT savepoint_name;
```

- Example:

```
SAVEPOINT a;
```

- Here, the transaction can later be rolled back to this specific point.

3. ROLLBACK

- The `ROLLBACK` command undoes changes made in the current transaction. If a savepoint is specified, it rolls back the transaction to that particular savepoint, undoing only changes made after the savepoint.

```
ROLLBACK [TO SAVEPOINT savepoint_name];
```

- Example:

```
ROLLBACK TO SAVEPOINT a;
```

- If no savepoint is mentioned, `ROLLBACK` will revert all the changes made since the start of the transaction.

4. COMMIT

- The `COMMIT` command saves all changes made during the transaction permanently to the database.

```
COMMIT;
```

- After a `COMMIT`, changes are permanent, and you cannot undo them using rollback.

Practical Example with Step-by-Step TCL Operations

Let's walk through the example you provided:

1. **Creating the Database and Table**

   - We start by creating a new database and table: ```sql CREATE DATABASE demo_tcl; USE demo_tcl;

   CREATE TABLE emp2 (

   ```
   emp_id INT,
   name VARCHAR(20),
   salary FLOAT
   ); ```
   ```

2. **Inserting Data**

   - Inserting multiple rows of employee data into the `emp2` table:

   ```
   INSERT INTO emp2 VALUES
   (1, 'aman', 65000),
   (2, 'deepak', 78000),
   (3, 'akash', 45000),
   (4, 'pankaj', 28000),
   (5, 'saroj', 48000),
   (6, 'bipin', 35000),
   (7, 'raj', 29000);
   ```

3. **SAVEPOINT A**

   - The `SAVEPOINT a` command marks the first savepoint:

   ```
   SAVEPOINT a;
   ```

4. **Delete Operation**

   - We delete employees with a salary greater than 50,000:

   ```
   DELETE FROM emp2 WHERE salary > 50000;
   ```

   - This removes employees `aman` and `deepak` from the table.

5. **SAVEPOINT B**

   - After deleting, we create another savepoint:

   ```
   SAVEPOINT b;
   ```

6. **Updating Salaries**
   - Update the salaries of remaining employees by increasing them by 3000:
     ```sql
     UPDATE emp2 SET salary = salary + 3000;
     ```
7. **SAVEPOINT C**
   - After updating, create the third savepoint:
     ```sql
     SAVEPOINT c;
     ```
8. **Deleting More Rows**
   - Delete employees whose salary is now greater than 50,000 (which applies to `saroj` after the salary increase):
     ```sql
     DELETE FROM emp2 WHERE salary > 50000;
     ```
9. **ROLLBACK TO SAVEPOINT C**
   - We now roll back to `SAVEPOINT c`, undoing the most recent delete operation:
     ```sql
     ROLLBACK TO SAVEPOINT c;
     ```
   - This restores `saroj` back into the table with their updated salary.
10. **COMMIT**
    - Finally, we commit the transaction, saving all the changes permanently:
      ```sql
      COMMIT;
      ```

Final State of the `emp2` Table

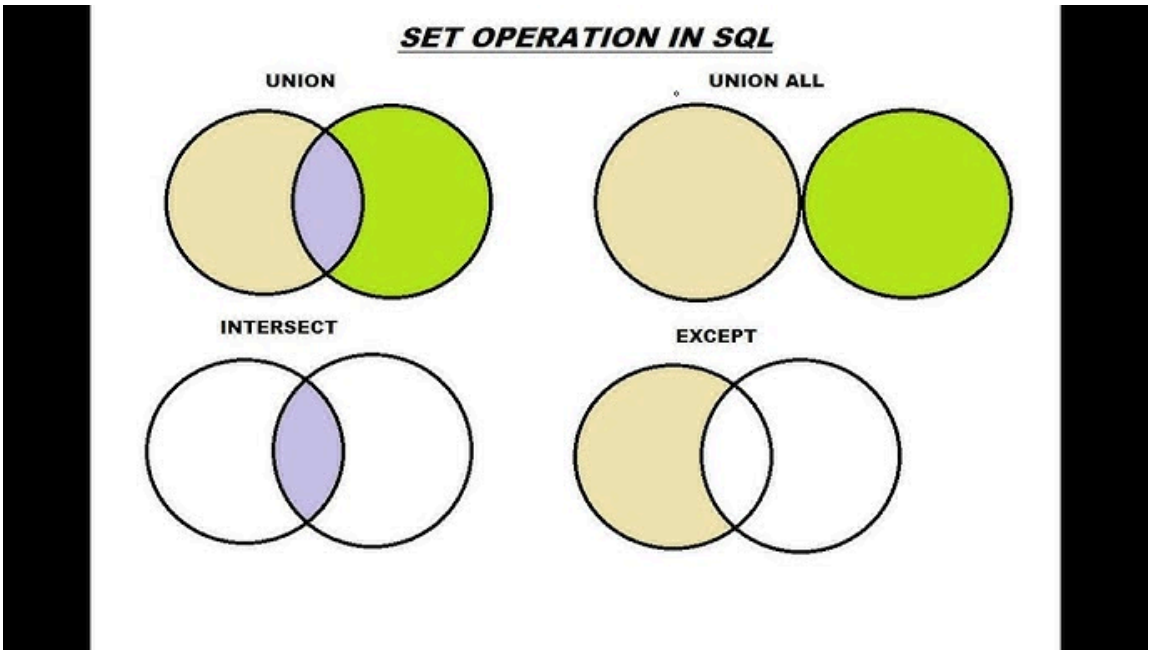
After rolling back to `SAVEPOINT c` and committing the transaction, the final state of the `emp2` table will be:

```sql
SELECT * FROM emp2;
+--------+--------+--------+
| emp_id | name   | salary |
+--------+--------+--------+
|      3 | akash  |  48000 |
|      4 | pankaj |  31000 |
|      5 | saroj  |  51000 |
|      6 | bipin  |  38000 |
|      7 | raj    |  32000 |
+--------+--------+--------+
```

Explanation of Key Points:

- **SAVEPOINT** allows you to mark specific points in your transaction and roll back to these points if necessary. This is useful for long transactions where you may need to undo only specific operations.
- **ROLLBACK** reverts the database to a previous state. You can either roll back the entire transaction or just part of it using `SAVEPOINT`.
- **COMMIT** makes the changes permanent, ensuring that the data stays modified even after the transaction is complete.

Set Operators



1. **UNION**:

   - Combines the result of two queries and returns **only distinct** values.
   - Columns and data types must match between both tables.
   - **Syntax**:

     ```
     SELECT column1, column2 FROM table1
     UNION
     SELECT column1, column2 FROM table2;
     ```

   - **Example** (Combining Books and Movies with distinct values):

     ```
     SELECT * FROM Books
     UNION
     SELECT * FROM Movies;
     ```

     Output:

     ```
     +------+---------------------+
     | ID   | TITLE               |
     +------+---------------------+
     | 1    | The Witcher         |
     | 2    | Harry Potter        |
     | 3    | Nineteen Eighty Four |
     | 4    | The Great Gatsbby   |
     | 1    | Iron Man            |
     | 3    | Matrix              |
     | 4    | Dr Strange          |
     +------+---------------------+
     ```

2. **UNION ALL**:

   - Combines the result of two queries and returns **all** values, including duplicates.
   - **Syntax**:

     ```
     SELECT column1, column2 FROM table1
     UNION ALL
     SELECT column1, column2 FROM table2;
     ```

- **Example** (Combining Books and Movies including duplicates):

```sql
SELECT * FROM Books
UNION ALL
SELECT * FROM Movies;
```

Output:

```
+------+---------------------+
| ID   | TITLE               |
+------+---------------------+
| 1    | The Witcher         |
| 2    | Harry Potter        |
| 3    | Nineteen Eighty Four |
| 4    | The Great Gatsbby   |
| 1    | Iron Man            |
| 2    | Harry Potter        |
| 3    | Matrix              |
| 4    | Dr Strange          |
+------+---------------------+
```

3. **INTERSECT**:

- Returns the **common** rows that are in both queries.
- Not natively supported in MySQL, but you can simulate it using `INNER JOIN` or `EXISTS`.
- **Example** (Finding common titles between Books and Movies):

```sql
SELECT * FROM Books
WHERE TITLE IN (SELECT TITLE FROM Movies);
```

Output:

```
+------+--------------+
| ID   | TITLE        |
+------+--------------+
| 2    | Harry Potter |
+------+--------------+
```

4. **EXCEPT (or MINUS)**:

- Returns the rows from the first query that **are not** in the second query.
- MySQL does not have `EXCEPT` or `MINUS`, but you can simulate it using `LEFT JOIN` or `NOT IN`.
- **Example** (Books that are not Movies):

```sql
SELECT * FROM Books
WHERE TITLE NOT IN (SELECT TITLE FROM Movies);
```

Output:

```
+------+---------------------+
| ID   | TITLE               |
+------+---------------------+
| 1    | The Witcher         |
| 3    | Nineteen Eighty Four |
| 4    | The Great Gatsbby   |
+------+---------------------+
```

Unique Values with `DISTINCT`

- **DISTINCT** is used to select only distinct (different) values. It does not delete duplicates but just filters them in the result.

  Example (Get distinct names from `contacts`):

```
SELECT DISTINCT(name) FROM contacts;
```

  Output:

```
+--------+
| name   |
+--------+
| deepak |
| aman   |
| pankaj |
| aakash |
+--------+
```

Removing Duplicates Using `DELETE`

- To permanently remove duplicates based on some criteria, use the `DELETE` statement with an `INNER JOIN` to find and delete duplicate entries.

  Example (Remove duplicates from `contacts` based on `email`):

```
DELETE t1 FROM contacts t1
INNER JOIN contacts t2
WHERE t1.id < t2.id AND t1.email = t2.email;
```

  Output:

```
Query OK, 3 rows affected
```

  Result (After deletion):

```
SELECT * FROM contacts;
```

  Output:

```
+----+--------+-----------------+
| id | name   | email           |
+----+--------+-----------------+
| 1  | deepak | deep@gmail.com  |
| 4  | aakash | aakash@gmail.com |
| 5  | pankaj | pankaj@gmail.com |
| 7  | deepak | deepak@gmail.com |
| 8  | aman   | aman@gmail.com  |
+----+--------+-----------------+
```

These examples illustrate how set operators work in SQL and how you can manage unique and duplicate data effectively.

In [ ]: