SQL Transaction Control Language (TCL)

Prompt

Create a detailed, beginner-friendly documentation for SQL Transaction Control Language (TCL) commands, including COMMIT, ROLLBACK, and SAVEPOINT. Structure it so each command has its own section under the same document and it should also include the following sections:

1. Title – Include the name of the SQL command.
2. Purpose – A brief explanation of what this command does.
3. Syntax – Provide the correct syntax with placeholders.
4. Parameters – Explain any parameters or keywords used in the command.
5. Examples – Include at least 2–3 examples with explanations.
6. Use Cases – Describe real-world scenarios where this command is used.
7. Best Practices – Mention common mistakes to avoid and tips.
8. Related Commands – Reference other SQL commands that are related.

Transaction Control Language (TCL) commands are used to manage transactions in a database. A **transaction** is a sequence of one or more SQL statements executed as a single, logical unit of work. The key principle of transactions is that they are all-or-nothing: either all statements within the transaction succeed and are made permanent, or none of them are.

The COMMIT Command

Purpose

The **COMMIT** command is used to permanently save all the changes made by the statements in the current transaction. Once you issue a COMMIT, the changes become visible to other database users and cannot be undone with a ROLLBACK.

Syntax

The syntax is simple and straightforward.

```
COMMIT;
```

Parameters

The COMMIT command does not take any parameters. It's a single keyword that finalizes a transaction.

Examples

Let's assume we have a table called accounts with columns id, name, and balance. We want to transfer $100 from account 1 to account 2.

Example 1: A successful transaction

```
BEGIN;

UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;

COMMIT;
```

In this example, both UPDATE statements are successful. The COMMIT command ensures that the balance changes are permanently saved to the database.

Use Cases

- **Financial Transfers:** A transfer from one bank account to another is a classic example. You debit one account and credit the other. If both operations are successful, you COMMIT the transaction.
- **Order Processing:** When a customer places an order, you might need to update inventory, create an order record, and log the transaction. If all these steps succeed, you COMMIT the changes to make the order final.

Best Practices

- Use COMMIT only after you are certain that all statements within the transaction have executed successfully and the data is in a valid state.
- Always start a new transaction with BEGIN or START TRANSACTION before issuing

a series of DML statements (INSERT, UPDATE, DELETE).

Related Commands

- **BEGIN**: Starts a new transaction.
- **ROLLBACK**: Undoes all changes in the current transaction.

The ROLLBACK Command

Purpose

The **ROLLBACK** command is used to undo all the changes made by statements within the current transaction. It reverts the database to the state it was in before the transaction began.

Syntax

The basic syntax is as follows. You can also specify a SAVEPOINT to roll back to a specific point within the transaction.

```
ROLLBACK;
ROLLBACK TO SAVEPOINT savepoint_name;
```

Parameters

- **SAVEPOINT savepoint_name**: An optional clause used to revert the transaction's changes only up to a specific, named SAVEPOINT rather than the entire transaction.

Examples

Using the same accounts table, let's look at a scenario where something goes wrong.

Example 1: Rolling back an entire transaction due to an error

```
BEGIN;

UPDATE accounts SET balance = balance - 100 WHERE id = 1;
-- The next statement fails, maybe because of an invalid account ID
UPDATE accounts SET balance = balance + 100 WHERE id = 99;

-- Because of the error, we revert all changes
ROLLBACK;
```

In this case, the ROLLBACK command ensures that the first UPDATE statement is also undone, and account 1's balance is reverted to its original state.

Use Cases

- **Error Handling:** If any part of a multi-step transaction fails (e.g., a credit card charge is declined), you can ROLLBACK the entire transaction to ensure no partial changes are saved.
- **Data Validation:** If you insert data and a validation check reveals it's incorrect, you can ROLLBACK the insertion.

Best Practices

- Use ROLLBACK as your primary way to handle errors within a transaction.
- Be cautious when using ROLLBACK without a SAVEPOINT, as it will undo all changes since the transaction began.

Related Commands

- **BEGIN**: Starts a new transaction.
- **COMMIT**: Permanently saves the changes.
- **SAVEPOINT**: Creates a point within a transaction to which you can roll back.

The SAVEPOINT Command

Purpose

A **SAVEPOINT** command creates a named checkpoint within a transaction. This allows you to selectively roll back to a specific point without discarding all the work done since the transaction started.

Syntax

You define a savepoint by giving it a unique name within the transaction.

```
SAVEPOINT savepoint_name;
```

Parameters

- **savepoint_name**: A user-defined identifier for the savepoint. This name must be unique within the current transaction.

Examples

Let's look at a more complex scenario with multiple steps, where a failure in one step shouldn't ruin everything.

Example 1: Rolling back to a savepoint

```
BEGIN;

-- Step 1: Update customer information
UPDATE customers SET last_login = NOW() WHERE id = 5;
SAVEPOINT step1_complete;

-- Step 2: Try to process an order
INSERT INTO orders (customer_id, amount) VALUES (5, 500);

-- Let's say the order fails for some reason, but we still want to keep the customer update.
-- Instead of rolling back the whole transaction, we roll back to the savepoint.
ROLLBACK TO SAVEPOINT step1_complete;

-- We can now commit the customer update, and the failed order insertion is gone.
COMMIT;
```

In this example, the customer's last_login update is kept, while the failed order insertion is discarded. The transaction is still active after the ROLLBACK TO SAVEPOINT, allowing you to continue or COMMIT the remaining changes.

Use Cases

- **Complex Multi-Step Processes:** In a long, complicated transaction, you can set savepoints after each successful sub-task. If a later sub-task fails, you can roll back to the last successful savepoint without re-doing the earlier work.
- **Conditional Operations:** You might perform an operation and then check a condition. If the condition is not met, you can roll back to a savepoint before that operation.

Best Practices

- Use SAVEPOINT for complex transactions where you need more granular control over rollbacks.

- Always use a meaningful name for your savepoints to improve code readability.

Related Commands

- **BEGIN**: Starts a new transaction.
- **ROLLBACK TO SAVEPOINT**: Reverts changes to a specific savepoint.
- **RELEASE SAVEPOINT**: Removes a savepoint without rolling back.