LAB: 3

Aim: To learn Transactions in SQL

Score:

Submission: A report to answer all question and SQL file

Due:

Download the "WideWorldImporters-Full.bak" :

1. **Create a transaction**
   i. Let's run the stored procedure that we created in the previous section.

```
CREATE OR ALTER PROCEDURE Warehouse.uspInsertColor (@Color AS nvarchar(100))
        AS
DECLARE @ColorID INT
SET @ColorID = (SELECT MAX(ColorID) FROM Warehouse.Colors)+1;
INSERT INTO Warehouse.Colors (ColorID, ColorName, LastEditedBy) VALUES (@ColorID, @Color,
1);
SELECT * FROM Warehouse.Colors WHERE ColorID = @ColorID ORDER BY ColorID DESC;
```

   ii. Start the transaction at the beginning and end the transaction at the end.
      a. Begin a transaction

```
BEGIN TRANSACTION FirstTransaction WITH MARK; -- or BEGIN TRAN
```

      Understand why it is important to name a transaction and usage of MARK.
      b. Use the previously developed stored procedure to add a color to the table

```
EXEC Warehouse.uspInsertColor 'Sunset Orange';  EXEC
Warehouse.uspInsertColor 'Tomato Red';
```

      c. Run the following query to view the added information.

```
SELECT * FROM Warehouse.Colors  ORDER
BY ColorID DESC;
```

      d. Open up a new query window and run the above query. What do you observe? Look at the message bar at the very bottom of the screen, what do you see?
      e. Let's leave this query running in this tab and go back to our transaction and roll back the transaction. What do you observe?

```
ROLLBACK TRANSACTION FirstTransaction; -- Undo the data input
```

Go back to your other query window and see what happen to the query that you executed and check the message bar at the bottom of the screen.

f. Repeat the step (a) to (d) and commit the transaction. Write the query for commit the transaction. Explain your observation.

2. Transaction save point

Within a transaction, you can create save points that allow you to roll back just a portion of the transaction while committing other portions of the transaction.

i. Track whether you're currently working inside of a transaction or not. Execute the following query and understand the output.

```
SELECT @@TRANCOUNT AS 'Open Transactions';
```

NOTE: @@TRANCOUNT is a system variable which used to track whether you're currently working inside of a transaction or not.

ii. Begin a transaction (no need to provide a name for this transaction)

iii. Repeat the step (i) and see the changes in the output.

iv. Nested transactions - create transactions within transactions.
plus: helpful to manage complex programming routines.
negative: If a rollback is encountered at any level within a nested transaction, all of the transactions are rolled back from the inside out and @@TRANCOUNT is reset to zero.

v. To alleviate the above limitation, you need to create save points along the way to roll back just a portion of a transaction.

a. Add a color- 'Lemongrass Green'- to the table Color (use Warehouse.uspInsertColor stored procedure)

b. Execute the following query to create a transaction savepoint

```
SAVE TRANSACTION SavePointOne;
```

c. Check the TRANCOUNT again and record the value. Can you state whether are still inside the transaction or not?

d. Add another color- 'Galaxy Purple'- to the table Color.

e. View the added information.

f. Execute the following query to revert to the savepoint – 'SavePointOne'.

```
ROLLBACK TRANSACTION SavePointOne;
```

What do you observe in the above rollback operation? If you omit the save point name, what would happen?

g. Check the TRANCOUNT again and record the value. Can you state whether are still inside the transaction or not?

h. Commit the above transaction and execute the query to view the information of Colors table. Explain your output.

3. Automatically roll back transactions

In the previous exercise, we've manually chosen whether to commit, or roll back a transaction.

i. What happens when a transaction includes a command that generates an error?

ii. Going back again to the colors table and run the SELECT statement to check the most recent color that you added. Note down the color.

iii. Start a new transaction.

iv. Insert a new color called 'burnished bronze'.

v. Repeat the above step. Note down the error message and explain the reason of this error message.

vi. Commit the above transaction.

vii. Run the SELECT statement to check the most recent color that you added.

viii. Go back to the question (i) and check your answer again based on the observation from the above operations.

ix. Use **XACT_ABORT** to tell SQL Server to automatically rollback transactions when run-time errors are encountered. Check status of XACT_ABORT server setting.

```
SELECT CASE WHEN (16384 & @@OPTIONS) = 16384 THEN 'ON'
    ELSE 'OFF'
    END AS XACT_ABORT;
```

x. Let's turn it on XACT_ABORT.

```
SET XACT_ABORT ON; -- or OFF
```

xi. Start a new transaction.

xii. Insert a new color called 'Glittering Gold'.

xiii. Repeat the above step. Now you can notice the reason for the error.

xiv. Commit the above transaction. What did you notice and explain the message?

xv. Run the SELECT statement to check the most recent color that you added. Do you see a record of 'Glittering Gold'?

xvi. Note that XACT_ABORT setting only applies to the current session. To test this, run the query at (viii) again and then open up a new query window and run the query at (viii). What to you get in the result?

4. Assignment

DISCLAIMER: Wherever there is **<YourFirstName>** in the sample code snippets provided in this document, you need to replace it with your actual first name. All the tables and the stored procedure you will create as part of this quiz will have this constraint. For example, a student named Loren Ipsum would be creating the table **<YourFirstName>SampleTable** as **LorenSampleTable**.

Let's consider a scenario where you have a university registration website where students can register for courses. You have three tables in your database: **Students, Courses** and **StudentRegistration**. The **Students** table contains information about each student, such as their student ID, Full Name, Email, and total credits. The **Courses** table contains information about all the available courses provided in the university, such as the CourseID, Course name, Instructor for the course, credits with respect to the course, and the number of seats available for the registered course.
You want to ensure that the **StudentRegistration** table is updated accurately when a student registers for a course, so you need to use a transaction to ensure the integrity of the data.
The required submission is highlighted in **RED** for each of the four questions.

i. Create database <First_Name>ADTLab3. **Submit the screenshots of successful query execution.**

ii. Create table <First_Name>Student table, <First_Name>StudentRegistration table and <First_Name>Course table with the following schema. Insert a few rows in the **Students** and **Course** table only. **Submit the error-free and working SQL code after modifying the table names, and the screenshot of the Students, Courses, and StudentRegistration table after inserting data.**

Students
StudentID (PK): INT
FullName: VARCHAR(100)
Email: VARCHAR(100)
TotalCredits: INT

StudentRegistration
RegistrationID (PK, Identity): INT
StudentID (FK to Students): INT
CourseID(FK to Course): INT

Courses
CourseID (PK): INT
CourseName: VARCHAR(100)
Instructor: VARCHAR(100)
CourseCredits: INT
AvailableSeats: INT

```
-- Insert into Students table
INSERT INTO <First_Name>Students (StudentID, FullName, Email,
TotalCredits)
VALUES (1, 'John Doe', 'john.doe@example.com', 0),
       (2, 'Jane Smith', 'jane.smith@example.com', 0),
       (3, 'Michael Johnson', 'michael.johnson@example.com', 0);

-- Insert into Courses table
INSERT INTO <First_Name>Courses (CourseID, CourseName, Instructor,
CourseCredits, AvailableSeats)
VALUES (1, 'Mathematics', 'Professor Anderson', 3, 1),
       (2, 'English Literature', 'Professor Thompson', 4, 10),
       (3, 'Computer Science', 'Professor Roberts', 5, 20);
```

iii. We need to create a store procedure whenever a student registers for a course. The procedure must check the availability of seats in the course before registering the student for the course. If the student is registered the availability of the seats should be deducted and the credits of the course should be added to the student's total credits.
The structure of the stored procedure is as follows:

- Name of the procedure: <Your_First_Name>_spInsertStudentRegistration which takes StudentID and CourseID as input parameters.
- Check the availability of Seats in the provided course table.
- Decrease the Availability of the Seats in the Courses Table.
- Add Course credits of the Courses table to the Student Total credits in the Students table.
- Insert the record into the StudentRegistration table with RegistrationID, StudentID, and CourseID.
- If the available seats are less than or equal to 0 then the transaction should be rolled back and print the message 'Course is full. Registration failed'. **Submit the error-free working SQL code of the stored procedure.**

iv. Testing the solution by registering below students for the following course.

a John Doe registers for English Literature.

b Michael Johnson registers for Mathematics.

c John Doe registers for Mathematics.

For each order in Question (iv), submit the screenshots of the following

- **Message displayed after executing the stored procedure.**
- **The output of the below queries**

```
SELECT * FROM <YourFirstName>Students;
SELECT * FROM <YourFirstName>Course;
SELECT * FROM <YourFirstName>StudentRegistrations;
```