Chapter-7 Transactional SQL with MS SQL Server

In MS SQL Server, transactions are a crucial part of ensuring data integrity and consistency in the database. **Transactional SQL** refers to SQL statements that are executed as part of a transaction, and it guarantees that a series of operations (e.g., INSERT, UPDATE, DELETE) are completed successfully as a unit. If any part of the transaction fails, the entire transaction can be rolled back to maintain consistency and prevent partial changes to the database.

Key Concepts of Transactions

1. **Atomicity**: The concept that a transaction is treated as a single "unit." Either all operations within a transaction are completed, or none of them are.
2. **Consistency**: The database transitions from one valid state to another. A transaction takes the database from a consistent state to another consistent state.
3. **Isolation**: Transactions are isolated from each other, meaning that the operations of one transaction are not visible to others until the transaction is complete. This ensures that transactions do not interfere with each other.
4. **Durability**: Once a transaction is committed, its changes are permanent, even in the case of a system failure.

These principles are collectively known as **ACID** (Atomicity, Consistency, Isolation, Durability).

---

1. BEGIN TRANSACTION

The BEGIN TRANSACTION statement is used to start a new transaction. Any SQL operations that are performed after this statement are part of the transaction.

Syntax:

```
BEGIN TRANSACTION;
```

Example:

```
BEGIN TRANSACTION;
    -- SQL operations here
```

---

2. COMMIT TRANSACTION

The COMMIT statement is used to save all the changes made during the transaction to the database. Once a COMMIT is issued, the changes become permanent and cannot be rolled back.

Syntax:

```
COMMIT TRANSACTION;
```

Example:

```
BEGIN TRANSACTION;
   INSERT INTO Employees (EmployeeID, EmployeeName, DepartmentID, Salary)
   VALUES (101, 'John Doe', 2, 50000);

   COMMIT TRANSACTION;
```

In this example, the insert operation is committed to the Employees table.

---

3. ROLLBACK TRANSACTION

The ROLLBACK statement is used to undo all changes made during the current transaction. This is typically used in the case of an error or if any condition fails during the transaction. When a rollback occurs, the database is returned to its state before the BEGIN TRANSACTION.

Syntax:

```
ROLLBACK TRANSACTION;
```

Example:

```
BEGIN TRANSACTION;
   UPDATE Employees
   SET Salary = 55000
   WHERE EmployeeID = 101;

   -- Simulate an error by checking a condition
   IF (SELECT Salary FROM Employees WHERE EmployeeID = 101) < 50000
   BEGIN
      ROLLBACK TRANSACTION;
      PRINT 'Transaction rolled back due to salary condition';
   END
   ELSE
   BEGIN
      COMMIT TRANSACTION;
```

```
    PRINT 'Transaction committed';
  END
```

In this example, if the salary condition is not met, the transaction is rolled back, and no changes are saved to the database.

---

4. SAVEPOINT

A **SAVEPOINT** allows you to create a point within a transaction to which you can roll back, without rolling back the entire transaction. This is useful when you want to undo some changes but keep others.

Syntax:

```
SAVE TRANSACTION savepoint_name;
```

Example:

```
BEGIN TRANSACTION;

  INSERT INTO Employees (EmployeeID, EmployeeName, Salary) VALUES (102, 'Jane Smith', 60000);
  SAVE TRANSACTION sp1;

  -- Perform another operation
  INSERT INTO Employees (EmployeeID, EmployeeName, Salary) VALUES (103, 'Mike Johnson', 65000);

  -- Rollback to the savepoint if needed
  ROLLBACK TRANSACTION sp1;

  COMMIT TRANSACTION;
```

In this example, after the first insert, a savepoint `sp1` is created. If needed, you can roll back to `sp1`, undoing the second insert while keeping the first insert.

---

5. Transaction Control with Error Handling

In transactional SQL, error handling is essential to ensure that if an error occurs, the transaction can be rolled back and the system maintains integrity. SQL Server provides **TRY...CATCH** blocks to handle errors within transactions.

Syntax:

```
BEGIN TRY
   BEGIN TRANSACTION;

   -- SQL statements
   -- Example: INSERT, UPDATE, DELETE

   COMMIT TRANSACTION;
END TRY
BEGIN CATCH
   -- Error handling
   PRINT 'Error occurred: ' + ERROR_MESSAGE();
   ROLLBACK TRANSACTION;
END CATCH;
```

Example:

```
BEGIN TRY
   BEGIN TRANSACTION;

      -- Inserting records
      INSERT INTO Employees (EmployeeID, EmployeeName, DepartmentID, Salary)
      VALUES (104, 'Sara Brown', 3, 70000);

      -- Simulating an error (division by zero)
      DECLARE @Result INT;
      SET @Result = 10 / 0; -- This will cause an error

   COMMIT TRANSACTION;  -- This will not be reached if an error occurs
END TRY
BEGIN CATCH
   PRINT 'Error occurred: ' + ERROR_MESSAGE();
   ROLLBACK TRANSACTION;  -- Rollback the transaction on error
END CATCH;
```

In this example, the transaction will roll back if an error occurs (such as a division by zero), ensuring that the database is not left in an inconsistent state.

---

6. SET TRANSACTION ISOLATION LEVEL

SQL Server allows you to control the level of isolation for a transaction using the `SET TRANSACTION ISOLATION LEVEL` statement. The isolation level controls how the transaction is isolated from other transactions that are running concurrently.

The different isolation levels are:

- **READ UNCOMMITTED**: Allows dirty reads. Other transactions can see uncommitted changes made by other transactions.

- **READ COMMITTED**: Default isolation level in SQL Server. Does not allow dirty reads, but allows non-repeatable reads.
- **REPEATABLE READ**: Prevents dirty reads and non-repeatable reads by locking the rows that are read during the transaction.
- **SERIALIZABLE**: The highest isolation level, which locks the range of rows being accessed, preventing any other transactions from modifying or inserting rows in that range.
- **SNAPSHOT**: Provides a snapshot of the data at the start of the transaction and does not block other transactions.

Example:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN TRANSACTION;

-- Perform transaction operations here

COMMIT TRANSACTION;
```

In this example, the transaction is set to use the SERIALIZABLE isolation level, which prevents other transactions from modifying the rows being accessed by the current transaction.

---

7. Nested Transactions

In MS SQL Server, transactions can be nested, but only the outermost COMMIT or ROLLBACK will be effective. If a nested transaction is rolled back, all the outer transactions will also be rolled back.

Example:

```
BEGIN TRANSACTION;  -- Outer transaction
   BEGIN TRANSACTION;  -- Inner transaction

   -- Operations for the inner transaction
   INSERT INTO Employees (EmployeeID, EmployeeName, Salary) VALUES (105, 'Tom Green', 60000);

   COMMIT TRANSACTION;  -- Inner commit

   -- Operations for the outer transaction
   UPDATE Employees SET Salary = 70000 WHERE EmployeeID = 104;

COMMIT TRANSACTION;  -- Outer commit
```

In this example, both the inner and outer transactions are committed. If an error occurs in the outer transaction, everything, including the inner transaction, will be rolled back.