National University of Computer and Emerging Sciences

Laboratory Manuals

for

Database Systems Lab

(CL -2005)

Course Instructor	Ms. Aleena Ahmad
Lab Instructor	Muhammad Kamran
Lab TA	Abdullah Naeem
Section	BCS-4F
Semester	Spring 2025

Department of Computer Science
FAST-NU, Lahore, Pakistan

Lab Manual 13

SQL

SQL tutorial gives unique learning on Structured Query Language and it helps to make practice on SQL commands which provides immediate results. SQL is a language of database, it includes database creation, deletion, fetching rows and modifying rows etc. SQL is an ANSI (American National Standards Institute) standard, but there are many different versions of the SQL language.

Why SQL?

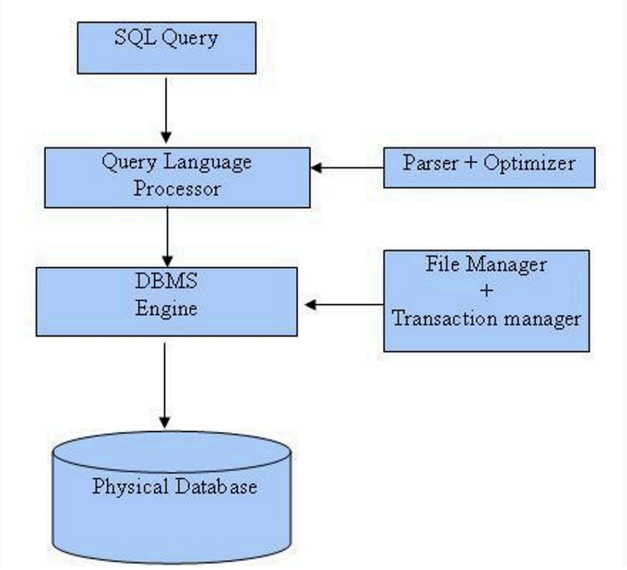
- Allows users to access data in relational database management systems.
- Allows users to describe the data.
- Allows users to define the data in the database and manipulate that data.
- Allows embedding within other languages using SQL modules, libraries & pre-compilers.
- Allows users to create and drop databases and tables.
- Allows users to create views, stored procedure, functions in a database.
- Allows users to set permissions on tables, procedures and views

SQL Process

When you are executing an SQL command for any RDBMS, the system determines the best way to carry out your request and SQL engine figures out how to interpret the task.

There are various components included in the process. These components are Query Dispatcher, Optimization Engines, Classic Query Engine and SQL Query Engine, etc. Classic query engine handles all non-SQL queries, but SQL query engine won't handle logical files.

Following is a simple diagram showing SQL Architecture:



1. Introduction to SQL Transactions

A **transaction** in a database is a sequence of one or more operations treated as a single logical unit. In SQL Server, transactions ensure that either all the operations succeed or none take effect, preserving data integrity. In practice, if any statement in a transaction fails, the database automatically rolls back all previous changes from that transaction, preventing partial updates. Transactions follow the ACID properties – **Atomicity, Consistency, Isolation, Durability** – to guarantee reliability. *Atomicity* ensures all parts of a transaction succeed or the whole is rolled back; *Consistency* ensures the database moves from one valid state to another; *Isolation* means concurrent transactions do not interfere with each other; and *Durability* means once a transaction is committed, its changes persist even after a failure. Together, these properties allow complex operations (such as transfers, batch updates, or coordinated inserts across tables) to be performed safely, knowing that errors will not leave the database in an inconsistent state.

Why use transactions?

- They group multiple SQL operations into one unit so that **all succeed or all fail** together.
- If an error occurs during the transaction, SQL Server can **ROLLBACK** all changes automatically, avoiding partial updates that would corrupt data consistency.
- When everything in the transaction executes without error, **COMMIT** makes all changes permanent atomically.
- In summary, transactions preserve data integrity by enforcing the ACID guarantees across multiple operations

Key Transaction Commands and Constructs

In T-SQL, the main commands for managing transactions are:

- **BEGIN TRANSACTION:** Marks the start of an explicit transaction. All subsequent operations become part of this transaction until a COMMIT or ROLLBACK is issued. For example:

```
BEGIN TRANSACTION;
-- (Perform one or more SQL statements here)
COMMIT;
```

This BEGIN TRANSACTION statement “marks the starting point of an explicit, local transaction

- **COMMIT:** Saves (makes permanent) all changes made in the current transaction. The `COMMIT` command applies all pending modifications to the database. As the Tutorialspoint guide explains, *"The COMMIT command is the transactional command used to save changes invoked by a transaction"*.
- **ROLLBACK:** Undoes all changes made in the current transaction since the last BEGIN TRANSACTION or SAVEPOINT. Executing `ROLLBACK TRANSACTION;` erases all data modifications done since the transaction began (or since a specified savepoint). For example:

```
BEGIN TRANSACTION;
UPDATE Emission_Record SET Quantity = Quantity + 10 WHERE SiteID = 101;
-- Oops, something went wrong:
ROLLBACK TRANSACTION;
```

This will revert the `UPDATE` and return the data to its original state

- **SAVEPOINT (T-SQL: SAVE TRANSACTION):** Sets a named point within a transaction to which you can roll back without canceling the entire transaction. In T-SQL you use `SAVE TRANSACTION SavePointName;`. This creates a marker inside the transaction. Later, you can issue `ROLLBACK TRANSACTION SavePointName;` to undo all changes back to that point, while preserving earlier work. For example:

```
BEGIN TRANSACTION;
  INSERT INTO Emission_Record (…) VALUES (…);
  SAVE TRANSACTION S1;    -- create a savepoint
  INSERT INTO Emission_Record (…) VALUES (…);
  -- If the second insert fails, we can roll back to S1:
  ROLLBACK TRANSACTION S1;
COMMIT;
```

This ensures that if the second insert fails, only it is undone (up to savepoint **S1**), while the first insert remains in the transaction

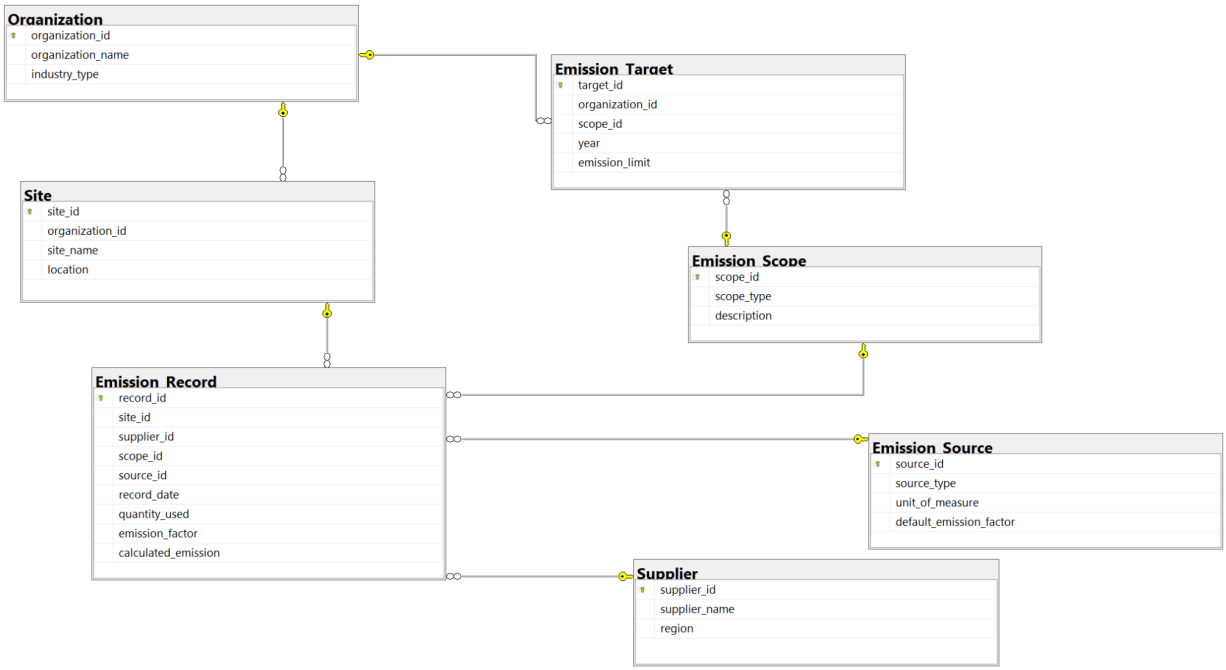
- **TRY…CATCH blocks:** In T-SQL, error handling is done with `TRY…CATCH`. You can wrap transactional code inside a `BEGIN TRY ... END TRY` block and handle errors in an associated `BEGIN CATCH ... END CATCH` block. If any statement in the TRY block causes an error, control immediately jumps to the CATCH block. A common pattern is:

```sql
BEGIN TRY
    BEGIN TRANSACTION;
    -- (Perform one or more SQL statements here)
    COMMIT;
END TRY
BEGIN CATCH
    ROLLBACK;
    PRINT 'Error occurred, transaction rolled back.';
END CATCH;
```

This ensures that if any error occurs during the transaction, the CATCH block will execute a ROLLBACK, preventing partial commits

Happy learning and querying!

Download sql file provided with this manual. Run that file and a schema with following details will be created:



In Lab Exercises

Super Dog and the Carbon Code Crackdown

In a futuristic city called EcoMetra, pollution has been on the rise. But there's hope! The legendary hero, Super Dog, has sniffed out a new mission — to analyze and reduce carbon emissions across the city. With his sharp instincts, high-speed tail-powered jetpack, and a team of brilliant animal sidekicks, he sets out to understand where all the carbon is coming from.

Characters:

- Super Dog – The hero! Can fly, analyze data at super-speed, and communicate with AI systems.
- EcoCat – Expert in Scope 1, 2, and 3 emissions, has infrared vision to detect direct emissions.
- PandaBytes – The coder panda, good with spreadsheets, databases, and modeling.
- Chameleon Camo – Disguises herself to infiltrate polluting facilities and collect raw data.

Mission Objective:

Help Super Dog and his team **gather, calculate, and analyze carbon emissions data** from different **suppliers, sites, and emission scopes** to build a full emissions profile for EcoMetra's organizations.

Write the T-SQL statements or scripts to accomplish the following tasks. Use the SuperDogCarbonDB schema (tables such as `Emission_Record`, `Sites`, `Site_Emissions`, `OrganizationCredits`, etc.) as needed. Make sure to include transactions, COMMIT, ROLLBACK, SAVEPOINT, and TRY/CATCH where appropriate.

1. Super Dog and EcoCat have just collected two new emissions readings: one for Site 101 and one for Site 102. In one transaction, insert both new rows into the Emission_Record table. For example:

```
BEGIN TRANSACTION;
INSERT INTO Emission_Record (...) VALUES (... for Site 101 ...);
INSERT INTO Emission_Record (...) VALUES (... for Site 102 ...);
COMMIT;
```

   Ensure that if either insert fails (for example, due to a constraint), neither row remains. In other words, both inserts should be rolled back if any single insert fails.

2. GadgetTail discovered a discrepancy that requires updating two tables for Site 101 simultaneously. In one transaction, update the Site_Emissions summary table by adding 50 tons to SiteID=101, and update the Sites table by increasing the TotalEmissions for SiteID=101 by 50 as well. For example:

```
BEGIN TRANSACTION;
UPDATE Site_Emissions SET EmissionTotal = EmissionTotal + 50
WHERE SiteID = 101;
UPDATE Sites SET TotalEmissions = TotalEmissions + 50
WHERE SiteID = 101;
COMMIT;
```

   If either update fails for any reason, roll back the entire transaction so that neither table is changed.

3. PandaBytes is inserting batch data into `Emission_Record`. After inserting the first new record for Site 101, create a savepoint (e.g. `SP1`). Then attempt to insert a second record that is intentionally invalid (for example, violate a NOT NULL or duplicate-key constraint). Use a `SAVE TRANSACTION SP1;` before the second insert. If the second insert fails, issue `ROLLBACK TRANSACTION SP1;` so that only the first insert is kept. Finally, `COMMIT` the transaction. Verify that the valid first

record remains in the table, while the failed insert was undone by rolling back to the savepoint.

4. Super Dog writes a script to handle critical data updates. In a `BEGIN TRY…END TRY` block, begin a transaction and perform two operations in `Emission_Record`. For example:

```
BEGIN TRY
    BEGIN TRANSACTION;
    INSERT INTO Emission_Record (...) VALUES (...);  -- first insert
    INSERT INTO Emission_Record (...) VALUES (...);  -- second insert
    COMMIT;
END TRY
BEGIN CATCH
    ROLLBACK;
    PRINT 'Error: Transaction rolled back.';
END CATCH;
```

   Cause the second insert to fail (e.g. insert a NULL into a non-nullable column). Confirm that control goes to the CATCH block and the transaction is rolled back, so neither insert remains in the table.

5. To see rollback in action, perform these steps manually:
   - Begin a transaction.
   - Update an existing emission record's value (for example, change an Emissions quantity). Do not commit yet.
   - Within the same transaction, run a SELECT on that record and observe the changed value (inside your session, it shows the update).
   - Now execute ROLLBACK TRANSACTION;.
   - Finally, run SELECT on that record again. Observe that the original value is restored (the update did not persist), demonstrating that the rollback undid the change.

6. EcoCat needs to transfer 100 carbon credits from Organization 1 to Organization 2 in a table OrganizationCredits(OrgID, Credits). Write a transaction that debits 100 from OrgID=1 and adds 100 to OrgID=2. Before committing, check a condition: if OrgID=1 does not have at least 100 credits, roll back instead of committing. For example:

```
BEGIN TRANSACTION;
UPDATE OrganizationCredits SET Credits = Credits - 100 WHERE OrgID = 1;
UPDATE OrganizationCredits SET Credits = Credits + 100 WHERE OrgID = 2;
IF (SELECT Credits FROM OrganizationCredits WHERE OrgID = 1) < 0
    ROLLBACK;
ELSE
    COMMIT;
```

This ensures the transfer only succeeds if Org 1 has enough credits; otherwise, it is aborted.

7. When registering a new site, two tables must be updated together: Sites and SiteManagers. In one transaction, insert a new row into Sites (e.g., SiteID=201, Name, Location, etc.) and a corresponding row into SiteManagers (e.g., the manager for that site). For example:

```
BEGIN TRANSACTION;
INSERT INTO Sites (SiteID, Name, Location, …) VALUES (201, 'New Site', 'East Zone', …);
INSERT INTO SiteManagers (ManagerID, SiteID, Name, …) VALUES (501, 201, 'Alex Eco', …);
COMMIT;
```

If inserting into either table fails (for instance, a foreign key or duplicate key violation), roll back the entire transaction so that neither table gets a new row.

8. PandaBytes wants to insert three new records into Emission_Record and use savepoints before each. Write a single transaction where you do the following:
    - Insert the first record for Site 202, then issue SAVE TRANSACTION S1;.
    - Insert the second record for Site 203, then issue SAVE TRANSACTION S2;.
    - Insert the third record for Site 204. Suppose the third insert fails (e.g. bad data). In that case, do ROLLBACK TRANSACTION S2;.
    - Finally, COMMIT the transaction.
    - This way, if the third insert fails, rolling back to S2 undoes only the third insert, while the first two inserts (before S2) remain. Verify which records end up saved in the table.
9. In SQL Server, nested BEGIN TRANSACTION calls increment a transaction count but do not create truly independent sub-transactions. Write a script to explore this:

```
BEGIN TRANSACTION;                -- Outer transaction starts
    UPDATE Site_Emissions ...;    -- perform some update
    BEGIN TRANSACTION;            -- Inner transaction (transaction count = 2)
        UPDATE Sites ...;         -- perform another update
    COMMIT;                       -- COMMIT inner (transaction count = 1)
    -- Note: data is still uncommitted at this point
ROLLBACK;                         -- ROLLBACK outer transaction (transaction count = 0)
```

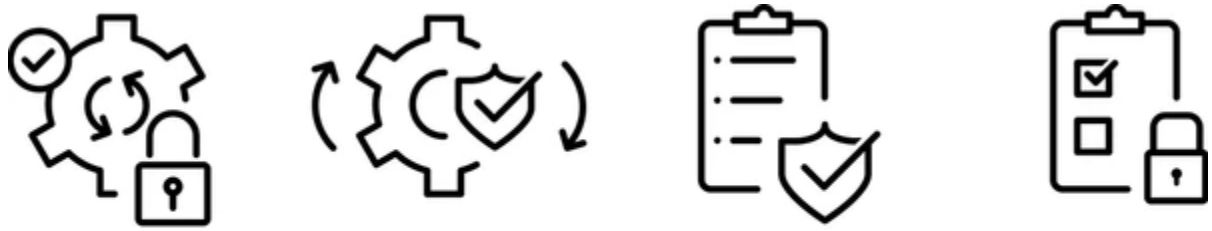
Observe that both updates are rolled back. This demonstrates that in T-SQL, a `COMMIT` of an inner transaction only decrements the count; only the final outer `COMMIT` actually makes changes permanent, and a rollback undoes all. (Super Dog notes that T-SQL does not support fully independent nested transactions.)

10. Write a stored procedure ProcessCarbonData that performs multiple operations (for example, inserting or updating emissions data) within a transaction and uses error handling. Include BEGIN TRANSACTION at the start and COMMIT at the end inside a TRY block, with a CATCH block to ROLLBACK and report errors. For instance:

```
CREATE PROCEDURE ProcessCarbonData
    @SiteID INT, @Credits INT
AS
BEGIN
    BEGIN TRY
        BEGIN TRANSACTION;
        -- (e.g., insert into Emission_Record, update OrganizationCredits, etc.)
        UPDATE OrganizationCredits SET Credits = Credits - @Credits WHERE OrgID = 1;
        INSERT INTO Emission_Record (SiteID, …) VALUES (@SiteID, …);
        COMMIT;
    END TRY
    BEGIN CATCH
        ROLLBACK;
        PRINT 'Error in ProcessCarbonData: ' + ERROR_MESSAGE();
    END CATCH;
END;
```

Test this procedure with valid inputs (which should commit successfully) and invalid inputs (which should trigger the CATCH and rollback). Document (with comments or printouts) that the transaction behaves atomically.

Part-II: Eco Mission Deployment Simulation using SQL



Now imagine coordinating a series of missions as a game. Two teams (Team 1 – EcoCat's crew, and Team 2 – PandaBytes' crew) take turns deploying to missions. Each deployment attempt has a random chance of success or failure (like rolling dice). Use transactions and error handling to simulate each mission attempt atomically. The first team to successfully complete a set number of missions (say 10) wins the simulation.

Requirements:

1. **Table Design: Create a table `MissionDeploy` (or similar) to log each deployment attempt. Suggested columns:**
    - **`deploy_id` (INT IDENTITY PRIMARY KEY) – unique attempt ID.**
    - **`team_id` (INT) – 1 for Team EcoCat, 2 for Team PandaBytes.**
    - **`risk_roll` (INT) – a random roll (1–6) to simulate mission risk.**
    - **`mission_count` (INT) – cumulative missions completed by that team after this attempt.**
    - **`success` (BIT) – 1 if the mission succeeded, 0 if it failed.**
2. **Game Flow: Simulate turns in T-SQL (e.g., in a loop or using a sequence of statements):**
    - **Turns: Team 1 (EcoCat) starts first, then Team 2, alternating each attempt.**
    - **Each Turn:**
        - **Begin a TRY…CATCH block and start a transaction (`BEGIN TRANSACTION`).**
        - **Generate a random roll, e.g.: 1-6**
        - **Determine success threshold (for example, success if `@roll > 4`).**

- If success: increment that team's mission count by 1, insert a new row into `MissionDeploy` with `success = 1` and updated `mission_count`, then `COMMIT`.

- If failure: ROLLBACK the transaction to undo any changes (the insert will not happen), and optionally insert a failure row by doing an insert *before* the error or outside the transaction with `success = 0`.

- Loop Condition: Continue alternating turns until a team's `mission_count` reaches 10 (or a chosen target). Use a `WHILE` loop that checks each team's count.

- Logging: Record each attempt in `MissionDeploy`. For example, on success commit, on failure you may still log a row (perhaps after rollback) indicating failure (depending on design).

- End of Game: After exiting the loop, print or select a message announcing which team won (the one who first reached the target count). For instance:

```
PRINT 'Team ' + CAST(@winningTeam AS VARCHAR) + ' wins with ' + CAST(@target AS VARCHAR) + ' missions completed!';
```

Submission Guidelines

---

1. submit following files strictly following the naming convention: l231234.sql

---

Best of Luck! Happy Querying
