National University of Computer and Emerging Sciences

Laboratory Manuals

for

Database Systems Lab

(CL -2005)

Course Instructor	Ms. Mamoona Majid
Lab Instructor	Durraiz Waseem
Lab TA	Sajeel Haider
Section	BCS-4J
Semester	Spring 2025

Department of Computer Science
FAST-NU, Lahore, Pakistan

Lab Manual 09

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?

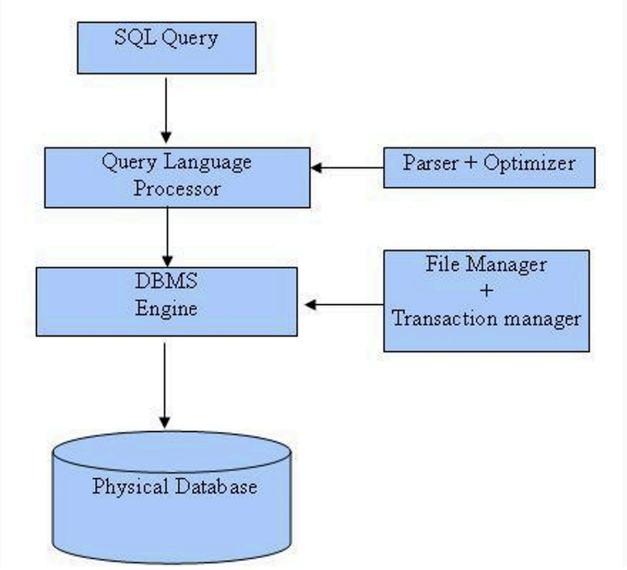
- Allows users to access data in relational database management systems.
- Allows users to describe the data.
- Allows users to define the data in the database and manipulate that data.
- Allows embedding within other languages using SQL modules, libraries & pre-compilers.
- Allows users to create and 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. DateAdd

Adds a specified time interval to a date.

DATEADD(datepart, number, date)

Example: Add 90 days to a course start date to find the exam date.

```
SELECT
    CourseName,
    StartDate,
    DATEADD(DAY, 90, StartDate) AS ExamDate
FROM Courses;
```

2. DateDiff

Returns the difference between two dates in specified units.

DATEDIFF(datepart, startdate, enddate)

Example: Calculate student enrollment duration.

```
SELECT
    StudentName,
    DATEDIFF(DAY, EnrollmentDate, GETDATE()) AS DaysEnrolled
FROM Students;
```

3. EOMONTH

Returns the last day of the month.

EOMONTH(start_date [, month_to_add])

Example: Get month-end fee submission deadline

```
SELECT
    StudentName,
    EOMONTH(GETDATE()) AS FeeDeadline
FROM Students;
```

4. STRING_AGG

Concatenates values with a separator.

STRING_AGG(expression, separator)

Example: List course names a student is enrolled in.

```
SELECT
    StudentID,
    STRING_AGG(CourseID, ', ') AS EnrolledCourses
FROM StudentCourses
GROUP BY StudentID;
```

StudentID	EnrolledCourses
1	101, 102
2	101
3	102

5. FORMAT

Formats values (dates, numbers) as strings

FORMAT(value, format [, culture])

Example: Display formatted enrollment date.

```
SELECT
    StudentName,
    FORMAT(EnrollmentDate, 'dd-MMM-yyyy') AS FormattedDate
FROM Students;
```

StudentName	FormattedDate
Alice	10-Jan-2023
Bob	15-Sep-2022
Charlie	05-Jun-2021

6. TRANSLATE

Replaces characters in a string

TRANSLATE(inputString, characters, replacements)

```
SELECT
    StudentName,
    TRANSLATE(PhoneNumber, '0123456789', '##########') AS ObfuscatedPhone
FROM Students;
```

7. CHARINDEX

Finds the position of a substring.

CHARINDEX(substring, string)

```
SELECT
    Email,
    CHARINDEX('@', Email) AS AtPosition
FROM Students;
```

8. Window Functions

A window function performs a calculation across a set of rows (window) related to the current row, without collapsing rows like aggregate functions do. It's used with OVER() clause. Unlike GROUP BY, window functions retain row-level details while computing aggregate or ranking metrics.

Term	Definition
Window	A set of rows defined by PARTITION BY and ordered by ORDER BY
OVER()	Defines the window context
PARTITION BY	Divides the result set into groups (like GROUP BY)
ORDER BY	Defines the logical order within each partition

1. **ROW_NUMBER**

   Assigns a unique sequential number to rows within a partition, based on an `ORDER BY` clause.

   ROW_NUMBER() OVER (PARTITION BY column ORDER BY column)

   Example: Rank students **within each course** by score.

```sql
SELECT
    StudentID,
    CourseID,
    Score,
    ROW_NUMBER() OVER (PARTITION BY CourseID ORDER BY Score DESC) AS RankInCourse
FROM StudentCourses;
```

2. **RANK()**

   Assigns rank to rows, **with gaps for ties**. Same values get the same rank, but next rank is skipped.

   RANK() OVER (PARTITION BY column ORDER BY column)

   Example: Rank students by **GPA** across the university.

```sql
SELECT
    StudentID,
    GPA,
    RANK() OVER (ORDER BY GPA DESC) AS OverallRank
FROM Students;
```

3. **DENSE_RANK()**

   Same as RANK() but no gaps in ranking sequence.

   DENSE_RANK() OVER (PARTITION BY column ORDER BY column)

   Example: Rank students in each department by GPA.

```sql
SELECT
    StudentName,
    DepartmentID,
    GPA,
    DENSE_RANK() OVER (PARTITION BY DepartmentID ORDER BY GPA DESC) AS DeptRank
FROM Students;
```

4. **NTILE(n)**

Divides rows into *n* equal groups (tiles); assigns a tile number to each row.

NTILE(n) OVER (ORDER BY column)

Example: Divide students into **quartiles** based on GPA

```
SELECT
    StudentID,
    GPA,
    NTILE(4) OVER (ORDER BY GPA DESC) AS GPA_Quartile
FROM Students;
```

5. **LAG()**

Returns the value of a column from **a previous row** in the same window.

LAG(column, offset, default) OVER (PARTITION BY column ORDER BY column)

Example: Compare a student's GPA with the **previous term**

```
SELECT
    StudentID,
    Term,
    GPA,
    LAG(GPA) OVER (PARTITION BY StudentID ORDER BY Term) AS PrevGPA
FROM StudentGPAHistory;
```

StudentID	Term	GPA	PrevGPA
1	2022-Fall	3.50	NULL
1	2023-Spring	3.80	3.50
2	2022-Fall	3.40	NULL
2	2023-Spring	3.60	3.40
3	2021-Fall	3.70	NULL
3	2022-Spring	3.90	3.70

a. Use DEFAULT to return value when not enough previous rows.
b. Use OFFSET to go back further (e.g., LAG(GPA, 2)).

6. **LEAD()**

Returns the value from the **next row** in the window frame

LEAD(column, offset, default) OVER (PARTITION BY column ORDER BY column)

Example: Show what GPA the student achieved **in the next term**.

```
SELECT
    StudentID,
    Term,
    GPA,
    LEAD(GPA) OVER (PARTITION BY StudentID ORDER BY Term) AS NextGPA
FROM StudentGPAHistory;
```

7. **FIRST_VALUE()**

Returns the **first value** in the window, based on ORDER BY.

FIRST_VALUE(column) OVER (PARTITION BY column ORDER BY column)

Example: Retrieve the **first term GPA** for each student.

```
SELECT
    StudentID,
    Term,
    GPA,
    FIRST_VALUE(GPA) OVER (PARTITION BY StudentID ORDER BY Term) AS FirstTermGPA
FROM StudentGPAHistory;
```

8. **LAST_VALUE()**

Returns the last value in the current window frame.

LAST_VALUE(column) OVER (PARTITION BY column ORDER BY column ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)

Example: Find the student's **most recent GPA** in their history.

```
SELECT
    StudentID,
    Term,
    GPA,
    LAST_VALUE(GPA) OVER (
        PARTITION BY StudentID ORDER BY Term
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) AS LatestGPA
FROM StudentGPAHistory;
```

Explanation:

1. **`UNBOUNDED PRECEDING`**: This refers to the first row in the partition. "Preceding" means that the window includes all rows before the current row (up to the first row in the partition). "Unbounded" means there is no limit to the number of rows, so it includes all preceding rows from the beginning of the partition.

2. **`UNBOUNDED FOLLOWING`**: This means the window includes all rows after the current row (up to the last row in the partition). "Following" refers to rows after the current row, and "Unbounded" means no limit, so it includes all rows from the current row to the end of the partition.

9. CTEs

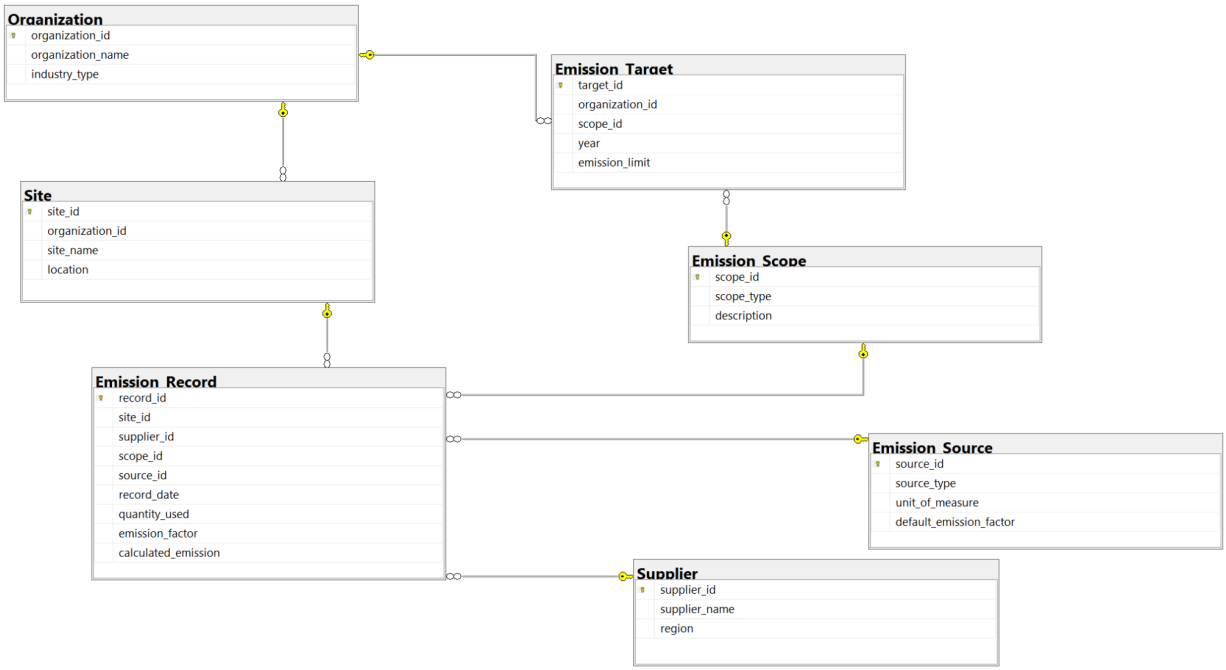
A CTE (Common Table Expression) is a temporary named result set in SQL that you can reference within a SELECT, INSERT, UPDATE, or DELETE statement.

Think of it as a temporary view that's only available during the execution of a single query.

```
WITH cte_name AS (
    SELECT column1, column2
    FROM some_table
    WHERE condition
)
SELECT *
FROM cte_name;
```

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 these queries to getting started with in-lab work

1. Super Dog wants to identify the top polluting sites across all organizations, regardless of emission scope. He needs a report that shows the **site name**, **organization**, and **total emissions produced** — ranked by the highest total emissions. (only nested queries)

2. To monitor trends over time, Super Dog wants to analyze how emissions are changing **year over year** within each **organization and emission scope**. He also wants to see the **cumulative (running) total** emissions across years. Write a SQL query that:

    - Groups data by **organization**, **scope type**, and **year** of emission record.

    - Shows both **yearly total emissions** and a **running total** (cumulative sum) per organization and scope using **window functions**.

3. Organizations have set annual emission reduction targets. Super Dog wants to know who is **failing to meet their targets** by comparing actual emissions to the predefined limits for a given year and scope.

    Write a SQL query that:

    - Compares actual emissions from emission records to targets from the Emission_Target table.
    - Filters only those organizations whose emissions exceed the defined limit for that year and scope.
    - Outputs: organization_name, year, scope_type, emission_limit, actual_emission, and status ("Exceeded" or "Within Limit").

4. Show All Emission Sources Used (concatenated column) by Each Site. Expected output is as in below

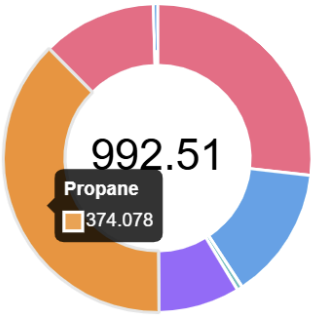
site_name	source_list
BuildBright Site A	Food Waste
CleanFoods Factory	Freight Shipping
EcoCorp HQ	Electricity
EcoCorp Plant A	Diesel Fuel
GreenWare Hub	Employee Commute
SkyHigh Terminal	Natural Gas
SmartHome Factory	Office Equipment, Refrigerant Leak, Food Waste
TransportMax Warehouse	Air Travel
TreeLine Studio	Refrigerant Leak
UrbanPower Plant 1	Electricity

5. Find Emission Increase/Decrease from Previous Record (use lead/lag)

6. Find First Recorded Emission per Site. (use First_Value)

Also help designing a few react components supporting following functional requirements

1. An organization should be able to sign up on localhost through a signup form
2. An organization should be able to sign in on localhost through a signin form
3. A query should be written to support the following pie chart on home page



Details of Pie Chart:

An organization is tracking emissions data for various sources across different sites. The organization needs to analyze and visualize the distribution of total emissions generated by each source for the previous month. The goal is to calculate the sum of emissions per source for the last month and generate a pie chart to visually represent the contribution of each source to the overall emissions for that period. The chart will allow stakeholders to easily identify the most significant sources of emissions and assist in decision-making for emission reduction strategies.

Submission Guidelines

---

1. submit following files strictly following the naming convention: l231234.sql
2. zip of src folders from frontend and backend

Best of Luck! Happy Querying