AI/ML Workshop Series

Sessions 1: Introduction to SQL

by

Prof. Muhammad Al-Zafar

22 October 2025

Learning Outcomes

1. Learn the basic terminology encountered.
2. Identify why we should even bother learning `SQL` in the age of GenAI.
3. Write basic queries.

Table of Contents

11 Joining tables
- When to use the different types of joins
- Two tables
- Three tables and more
- Left outer join example
- Right outer join example

12 Functions
- Built-in functions
- Calculations with text

13 GROUP BY and HAVING
- Grand totals

14 The golden order of writing queries

15 Subqueries
- Correlated subqueries

Background

Why Learn SQL in the Age of AI?

1. Most organizational data resides in relational databases, which are accessed and managed using SQL. AI models require clean, structured data for training and analysis, and SQL is the primary tool for preparing and extracting this data.
2. SQL proficiency allows users to understand the underlying data structures, ensuring that AI models receive relevant and accurate information.
3. A significant portion of AI and data science work ($\approx 85\%$) involves data cleaning and preparation. SQL provides efficient tools for filtering, transforming, and joining data, which is essential for creating suitable datasets for AI models; this is commonly referred to as **E**xtract, **T**ransform, and **L**oad (ETL).
4. Understanding SQL enables users to perform complex data manipulations directly within the database, often more efficiently than loading and processing data in other environments.

5. While AI can generate SQL queries, human oversight and understanding are necessary to verify the accuracy and relevance of these queries. SQL knowledge enables users to debug and validate AI-generated code.
6. Understanding SQL allows users to interpret and analyze the data retrieved by AI-generated queries, providing a deeper understanding of AI model behavior and results.
7. Learning SQL fosters structured and logical thinking, which is a valuable skill in any data-related field, including AI. The discipline of writing precise SQL queries translates into a systematic approach to problem-solving.
8. SQL serves as a common language across data teams, facilitating collaboration between data engineers, analysts, and scientists. This shared understanding is vital for effective data management and AI development.
9. Many data-centric roles, such as Data Analyst, Business Intelligence Developer, and Data Engineer, still require strong SQL skills, even with the increasing use of AI tools. SQL remains a fundamental requirement for working with data.

Terminology

1. **Database:** A structured collection of data that is organized and stored, typically in a digital format. It serves as a single source of truth for an application or system.
2. **Relational Database:** A database that stores and provides access to data points that are related to one another. Data is organized into one or more tables (relations) of rows and columns, and relationships between tables are defined using keys.
3. **SQL: S**tructured **Q**uery **L**anguage is a standardized programming language used for managing relational databases and performing various operations on the data within them. It's used to create, retrieve, update, and delete data.

4. **Table:** A fundamental structure in a relational database used to hold data. It is composed of rows (records) and columns (fields) and represents a single type of entity (e.g., a "Customers" table).
5. **Query:** A request for data or information from a database. In SQL, this is typically a `SELECT` statement used to retrieve data, but it can also refer to statements that modify the database (e.g., `INSERT`, `UPDATE`, `DELETE`).
6. **Schema:** The logical structure or blueprint of the entire database. It defines how the data is organized, including the tables, columns, data types, relationships, indexes, and constraints.
7. **Field:** Another name for a variable or column name.

8. **Primary Key:** A column or set of columns that uniquely identifies each row in a table. It ensures that no two rows have the same value for that key and there are no NULL values.
9. **Index:** A special lookup table used by the database search engine to speed up data retrieval operations. Think of it like the index in the back of a book.
10. **Normalization:** A systematic process of structuring a relational database to minimize data redundancy and improve data integrity.

Writing Basic Queries

Selecting All Elements from a Table

```
SELECT
    *
FROM
    Students
```

Selecting Specific Fields From a Table

```
-- 1 field
SELECT
    name
FROM
    Students
```

```
-- more than 1 field
SELECT
    name,
    major
FROM
    Students
```

Specifying a Condition

```
-- picking all the students whose age is greater
    than 20 years old
SELECT
    *
FROM
    Students
WHERE
    age > 20
```

```
-- specifying multiple conditions
SELECT
    *
FROM
    Students
WHERE
    age > 20
        AND
    major = 'mathematics'
```

Specifying a Range Within a Condition

```
-- seeking all grades between 90 and 100
SELECT
    *
FROM
    Students
WHERE
    grade BETWEEN 90 AND 100
```

```
-- specifying exact values
SELECT
    *
FROM
    Students
WHERE
    grade IN (92, 97, 100)
```

Specifying Exact Values Within a Condition

```
-- numerical values
SELECT
    *
FROM
    Students
WHERE
    age = 20
```

```
-- string values
SELECT
    *
FROM
    Students
WHERE
    last_name = 'Alnuaimi'
```

Wild Card Characters

Suppose from a movie database, you want to select all the movies that have Avengers in them.

```
SELECT
    *
FROM
    Movies
WHERE
    Title LIKE 'Avengers'
```

Suppose that we want to select all the movies that start with Pokemon and end with something else.

```
SELECT
    *
FROM
    Movies
WHERE
    Title LIKE 'Pokemon%'
```

Suppose that we want Pokemon anywhere in the title.

```
SELECT
    *
FROM
    Movies
WHERE
    Title = '%Pokemon%'
```

Suppose that in our student table, we want to find all students born on 3 May 2007.

SQL uses the format: YYYY-MM-DD, i.e., year-month-day for dates.

```
SELECT
    *
FROM
    Stuents
WHERE
    date_of_birth = '2007-05-03'
```

Suppose that we want to find all the students born in July.

```
SELECT
    *
FROM
    Students
WHERE
    MONTH(date_of_birth) = 5
```

Suppose that we want to find all the students born in the year 2008.

```
SELECT
    *
FROM
    Students
WHERE
    YEAR(date_of_birth) = 2008
```

Suppose that we want to find all the students born on the 23^{rd}.

```
SELECT
    *
FROM
    Students
WHERE
    DAY(date_of_birth) = 23
```

It's important to note that in a `WHERE` clause, you cannot put an alias as a column. This means if you want to filter by some condition, you would would have to put the whole condition in. We will see an example of this when dealing with `CASE` statements.

Ordering a Table

```sql
-- ascending (lowest to highest) order
SELECT
    name, major
FROM
    Students
WHERE
    age > 20
ORDER BY
    name ASC;
```

```sql
-- descending (highest to lowest) order
SELECT
    name, major
FROM
    Students
WHERE
    age > 20
ORDER BY
    grade DESC;
```

Renaming a Field to Something You Want

```
SELECT
    name AS [Name],
    age,
    date_of_birth AS [DOB]
FROM
    Students
```

Selecting n Number of Records From a Table

```
-- top 10 records
SELECT TOP 10
    name,
    age,
    date_of_birth
FROM
    Students
```

```
-- top 50 records
SELECT TOP 50
    name,
    age,
    date_of_birth
FROM
    Students
```

Calculated Columns in a Table

Suppose that you wanted to work out the profit/loss for a movie based upon the difference between its box office takings and its budget, i.e.,

$$\boxed{\text{profit/loss} = \text{box office takings} - \text{budget}}$$

```
SELECT
    movie_name,
    movie_box_office_takings,
    movie_budget,
    movie_box_office_takings - movie_budget  AS [
   Profit]
FROM
    Movies
ORDER BY
    Profit DESC
```

Suppose that you wanted to calculate the **V**alue **A**dded **T**ax (VAT) from some transactions in our database from our cost of sales

```
-- assuming VAT is 5% in the UAE
SELECT
    sales,
    cost_of_sales,
    cost_of_sales * 0.05 AS [VAT]
FROM
    sales_table
```

CASE Statements

In SQL, `CASE` statements are like `if` statements in any programming language: It is used to test whether criteria are met, and if they are met, it returns a specific result.

Suppose that we want to categorize student's grades as follows:

- A: 90-100%
- B: 80-89%
- C: 70-79%
- D: 60-69%
- F: 0-59%

```
SELECT
    Name,
    Student_Number,
    Grade,
    CASE
        WHEN Grade >= 90 AND <= 100 THEN 'A'
        WHEN Grade >= 80 AND <= 89 THEN 'B'
        WHEN Grade >= 70 AND <= 79 THEN 'C'
        WHEN Grade >= 60 AND <= 69 THEN 'D'
        ELSE 'F'
    END AS [Symbol], -- adding an alias
    Date_of_Birth
FROM
    Students
```

Suppose that in the student grade and symbol example, we wanted to know all the students that had the **F** symbol, i.e., those students that failed.

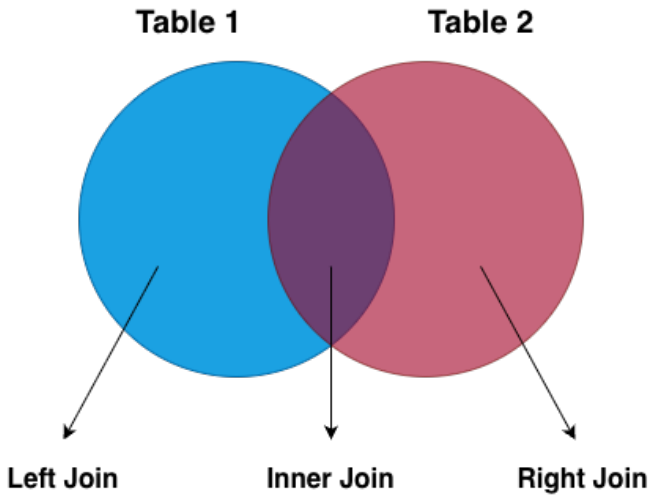
```
SELECT
    Name,
    Student_Number,
    Grade,
    CASE
        WHEN Grade >= 90 AND <= 100 THEN 'A'
        WHEN Grade >= 80 AND <= 89 THEN 'B'
        WHEN Grade >= 70 AND <= 79 THEN 'C'
        WHEN Grade >= 60 AND <= 69 THEN 'D'
        ELSE 'F'
    END AS [Symbol], -- adding an alias
    Date_of_Birth
FROM
    Students
WHERE
        CASE
        WHEN Grade >= 90 AND <= 100 THEN 'A'
        WHEN Grade >= 80 AND <= 89 THEN 'B'
        WHEN Grade >= 70 AND <= 79 THEN 'C'
        WHEN Grade >= 60 AND <= 69 THEN 'D'
        ELSE 'F'
    END = 'F'
```

Suppose that we wanted to identify all the students who have the name **Mohamed** in their full names, and print out **Amazing** for that student.

```
SELECT
    *,
    CASE
        WHEN Name LIKE '%Mohamed%' THEN 'Amazing'
    END
FROM
    Students
```

Joining tables together are important for connecting two, or more, tables that are related by some common features, so that a "super table" can be formed that is holistic and contains more information than the singular tables on their own.

- Essentially, there are three types of joins: Right, left, and inner.
- It is best to understand these joins using a Venn diagram



When to Use the Different Joins?

- **Inner joins** are perfect when you want to see corresponding information in two different tables.
- **Left joins** are used when you want to retrieve all records from the left table and any matching records from the right table. If there is no match, a `NULL` value will be returned.
- **Right joins** are used when you want to retrieve all records from the right table and any matching records from the left table. If there is no match, a `NULL` value will be returned.

Suppose that you wanted to join a Movies table with a Directors table on the common key **FilmDirectorID** from the **first table**, and **DirectorID** from the **second table**.

```
SELECT
    *
FROM
    Movies_Table AS t1
        INNER JOIN -- you could also just use the
    word JOIN as well, but it is good practice to be
     explicit
    Directors_Table AS t2
        ON
    t1.FilmDirectorID = t2.DirectorID
```

We can easily join more than two tables by following the similar syntax and methodology as above.

```
SELECT
    *
FROM
    tbl1 AS t1
        INNER JOIN
    tbl2 AS t2
        ON
    t1.key = t2.key
        INNER JOIN
    tbl3 AS t3
        ON
    t1.key = t3.key
```

Suppose that you wanted to join the **Customers** table to the **Orders** table and both tables have the common key **CustomerID**, and you wanted to keep all the information in the **Customers** table.

```
SELECT
    *
FROM
    Customers AS t1
        LEFT OUTER JOIN
    Orders AS t2
        ON
    t1.CustomerID = t2.CustomerID
```

Suppose that you wanted to join the **Orders** table to the **Employees** table and both tables have the **EmployeeID** as a common key, and you wanted to keep all the information in the **Employees** table.

```
SELECT
    *
FROM
    Orders AS t1
        RIGHT OUTER JOIN
    Employees AS t2
        ON
    t1.EmployeeID = t2.EmployeeID
```

Built-In Functions

Suppose that you wanted to convert all the student names to upper case.

```
SELECT
    UPPER(Names),
    Date_of_Birth
FROM
    Students
```

- There are many more built-in functions. A simple Google search or asking ChatGPT can tell you everything you need to know.

Suppose that you wanted to write a student's name as **Mr. Ali Almansoori**, respecting the spaces between the title, name, and surname.

```
SELECT
    Title,
    Name,
    Surname,
    Title + ' ' + Name + ' ' + Surname
FROM
    Students
```

The + symbol is used to join text.

GROUP BY and HAVING Statements

The GROUP BY and HAVING statements are used for aggregating queries.

Aggregate functions change a column of numbers into a single datapoint. The five most common aggregate functions are: SUM, AVG, MAX, MIN, and COUNT.

Suppose that you wanted to find the sum of all the ages of students.

```
SELECT
    SUM(Age)
FROM
    Students
```

When you are using the aggregate functions, any other fields that are not aggregated must be put into the `GROUP BY` clause.

Suppose that you want to calculate the average amount of money that consumers spend in a month.

```
SELECT
    Customer_Name,
    Account_Number,
    AVG(Amount_Spent)
FROM
    Account_Table
GROUP BY
    Customer_Name,
    Account_Number
ORDER BY
    Customer_Name ASC
```

We can also add **grand totals** to numerical columns.

Suppose you wanted the average age of consumers and the total amount of money they spent. Additionally, you wanted a grand total of each of these columns.

```
SELECT
    Customer_Name,
    AVG(Age),
    SUM(Spending)
FROM
    Customers_Table
GROUP BY
    Customer_Name WITH ROLLUP
ORDER BY
    Customer_Name ASC
```

The HAVING clause is used when:

- You need to filter data based on the result of an aggregate function.
- When you need to apply conditions to groups of rows created by GROUP BY.

Suppose that you want to find departments in a large multinational corporation that have more than 50 employees.

```
SELECT
    Department,
    COUNT(*)
FROM
    Employees_Table
GROUP BY
    Department
HAVING
    COUNT(*) > 5
```

The Golden Order of Writing Queries

- The golden order of writing queries is the optimal sequence of writing queries such that they run without failing.
- This order is

```
SELECT
FROM
WHERE
GROUP BY
HAVING
ORDER BY
```

A cool mnemonic I made up that can help you remember this is: **S**tan Lee (the ultimate creator) must specify exactly what to select for the final credits. Next, **F**ury identifies the primary faction that the heroes are from: The force of Earth's Mightiest Heroes. He needs to **W**eed out any weak, defeated villains; he only includes heroes where their power level is high. Then, he needs to **G**roup the heroes **B**y their teams (Avengers, Guardians, X-Men, etc.) to analyze them. He is **H**aving a meeting with the only heaps of groups that have more than five members. Finally, he puts the heroes in **O**rder **B**y their box office gross, from highest to lowest.

(Uncorrelated) Subqueries

A **subquery** is simply one query nested inside another query. Usually, subqueries are nested within the `SELECT`, `FROM`, or `WHERE` clause.

- Essentially, to write subqueries, one should follow a three-step process:
  1. Wrap your subquery up inside a set of brackets, forming part of the `WHERE` clause in your outer query.
  2. Write the inner query as normal.
  3. Combine the outer and inner queries.

Suppose that you wanted the student's name, date of birth, major, and high school name, who had the highest grade in Physics.

```
SELECT
    Name,
    Date_of_Birth,
    Major,
    High_School
FROM
    Students
WHERE
    (SELECT MAX(Grade) WHERE subject = 'Physics') --
     MAX finds the highest/largest value
```

Notice how we wrote the outer query in one line (linearly). In SQL we can write queries in any way, in lower or upper case. However, for understanding and simplification, previously we wrote queries in a spread-out manner.

Suppose we wanted to know the average grade of all students across all their subjects.

```
SELECT
    Name
FROM
    Students
WHERE
    (SELECT AVG(Grade) FROM Students) -- AVG finds
    the (arithmetic) average of a numerical column
```

Correlated Subqueries

A **correlated subquery** is a `SQL` subquery that references columns from an outer query. Unlike a regular subquery that runs once, a correlated subquery is executed for each row processed by the outer query, making it a "row-by-row" operation. This makes them useful for row-by-row comparisons, but they can be a performance bottleneck on large datasets.

- At the beginner-level, we shall avoid these unnecessary complications.