AI/ML Workshop Series

Sessions 2: Introduction to SQL

by

Prof. Muhammad Al-Zafar

23 October 2025

Learning Outcomes

1. Learn about `USE` statements.
2. Learn about `WITH` statements.
3. Learn about `RANK` statements.
4. Learn about stored procedures.
5. Apply all we have learned thus far to real-world problems.

Table of Contents

USE Statements

- The USE statement is used to specify exactly which database you are referring to in order to avoid ambiguity.

  Suppose that you wanted to find all the students whose grade is 95% or above from the Students table in the RecordsDB

```
USE RecordsDB
SELECT
    *
FROM
    Students
WHERE
    grade >= 95
```

`WITH` Statements

- The `WITH` clause is used to define a Common Table Expression (CTE).
- A **CTE** is a temporary named result set derived from a simple query, defined within the execution scope of a single `SELECT`, `INSERT`, `UPDATE`, `MERGE`, or `DELETE` statement.
- It enhances query readability and maintainability by breaking down complex queries into smaller, more manageable, and reusable subqueries.

Suppose that you want to pick all the high-value customers from your store.

```sql
WITH HighValueCustomers AS (
    SELECT CustomerID, CustomerName, SUM(OrderTotal)
     AS TotalSpent
    FROM Customers c
    JOIN Orders o ON c.CustomerID = o.CustomerID
    GROUP BY CustomerID, CustomerName
    HAVING SUM(OrderTotal) > 1000
)
SELECT CustomerName, TotalSpent
FROM HighValueCustomers
ORDER BY TotalSpent DESC
```

RANK Statements

- SQL provides several window functions for ranking data within a result set. These functions are used with the `OVER()` clause to define the window (or partition) over which the ranking is performed and the order of ranking.
- `RANK()` assigns a rank to each row within a partition.
- If two or more rows have the same value in the ordering column(s), they receive the same rank.
- Crucially, `RANK()` creates gaps in the ranking sequence after ties. For example, if two rows tie for rank 1, the next rank assigned will be 3 (skipping 2).

Suppose that you wanted to rank students by their grade to assign positions, i.e., 1st, 2nd, 3rd, etc.

```
SELECT
    StudentID,
    Name,
    Grade,
    RANK() OVER (ORDER BY Grade DESC) AS RankByScore
FROM
    Students
```

Stored Procedures

Stored Procedures

A **stored procedure** is simply a bunch of `SQL` queries grouped together under a single heading, similar to methods or subroutines in other programming languages. It can be done, essentially, with any `SELECT` statement.

Why Use Stored Procedures?

Essentially, for speed and efficiency, i.e., avoiding writing the same queries over and over.

Basic Stored Procedures

Suppose that you wanted to write a stored procedure to pick all the information from the students table that is located in the RecordsDB.

```
USE RecordsDB
GO -- Used to define a new batch, which is important
     for the CREATE PROC below

CREATE PROC spStudents -- Could have also written
    CREATE PROCEDURE
AS
BEGIN -- It's good practice to enclose your query in
     the BEGIN and END block
    SELECT
        *
    FROM
        Students
END
```

Location of Stored Procedures in SSMS

- Upon executing this code, you should see a message like **Command(s) completed successfully.**
- To locate your stored procedure, you go to the **Object Explorer** and look in the correct database.
- In the above example, you would check RecordsDB, then the **Programmability** folder, and then navigate to the **Stored Procedures** folder, and here is where you will find the stored procedures that you have written.

Note: Sometimes, the stored procedure you may have just written will not be located at the same time after you execute it. In this case, go back to your database, in this example about it is the **RecordsDB**, and manually refresh the database.

Using a Stored Procedure

- After a stored procedure is created, we usually discard/close the window where we created and executed it.
- To execute/use the stored procedure, the syntax is

```
EXEC spStudents -- or EXECUTE spStudents
```

Modifying a Stored Procedure Once it is Already Created

- You go to the **Object Explorer** on the left-hand side and locate the stored procedure, and right click and **Modify** to open it up. You then change the word CREATE to ALTER.

Suppose that in the previous example, we wanted to order the list of students in descending order.

```
USE RecordsDB
GO -- Used to define a new batch, which is important
     for the CREATE PROC below

ALTER PROC spStudents -- Could have also written
    CREATE PROCEDURE
AS
BEGIN -- It's good practice to enclose your query in
     the BEGIN and END block
    SELECT
        *
    FROM
        Students
END
```

Deleting a Stored Procedure

- To delete a stored procedure, there are effectively two ways:
    - If you are using SSMS, you can simply right-click on it from the **Object Explorer** and then delete it.
    - Alternatively, in a query window, you can just run the command:

```
DROP PROC spStudents
```

Stored Procedures with Parameters

- All parameter names must begin with an ampersand (@) symbol.

Suppose that we wanted to create a stored procedure that returns all the information of the students whose grade was 92% and above.

```
USE RecordsDB
GO

CREATE PROC spMinGrade(@min_grade AS INT)
AS
BEGIN
    SELECT
        *
    FROM
        Students
    WHERE
        Grade >= min_grade
END
```

- To run the query now, we have to specify the parameter as follows:

```
EXEC spMinGrade 92
```

Adding More than One Parameter

Suppose that we wanted to create a stored procedure that returns all the information of the students whose grade was 97% and above, who live in Dubai Hills, and whose surname is Alnuaimi.

```
USE RecordsDB
GO

CREATE PROC spStuInfo
    (
    @min_grade AS INT,
    @stu_address AS VARCHAR(MAX), -- allows you to
   put in strings of maximum length
    @stu_surname AS VARCHAR(MAX)
    )
AS
BEGIN
    SELECT
        *
    FROM
        Students
    WHERE
        Grade >= min_grade AND
        Address LIKE '%' + stu_address + '%' AND
        Surname LIKE '%' stu_surname '%'
END
```

- Now, we execute the stored procedure as follows:

```
EXEC spStuInfo 97, 'Dubai Hills', 'Alnuaimi'
```

- Alternatively, we could explicitly name the parameters as follows:

```
EXEC spStuInfo
    @min_grade = 97,
    @stu_address = 'Dubai Hills',
    @stu_surname = 'Alnuaimi'
```

Making Parameters Optional – Setting Default Values

```
USE RecordsDB
GO

CREATE PROC spStuInfo
    (
    @min_grade AS INT = 0,
    @stu_address AS VARCHAR(MAX) = 'UAE',
    @stu_surname AS VARCHAR(MAX)
    )
AS
BEGIN
    SELECT * FROM Students
    WHERE
        Grade >= min_grade AND
        Address LIKE '%' + stu_address + '%' AND
        Surname LIKE '%' stu_surname '%'
END
```

Real-World Problems: Putting All We Learned Into Practice

Problem 1: e-Commerce

In an e-commerce platform like Amazon or Shopify, you might need to retrieve a list of products that are low in stock but have high recent sales velocity, while accounting for seasonal trends, supplier delays, and customer reviews. This involves joining multiple tables (products, inventory, sales, reviews), using window functions for ranking, and subqueries for filtering.

- **Scenario Details:** Identify the top 10 products in the "Electronics" category that have sold more than 50 units in the last 30 days, have an average review rating above 4.0, and have a current stock of less than 20% of their average monthly sales. Also, flag products with pending supplier orders.

```sql
SELECT p.product_id, p.name,
       SUM(s.quantity) AS total_sold_last_30_days,
       AVG(r.rating) AS avg_rating,
       i.current_stock,
       RANK() OVER (ORDER BY SUM(s.quantity) DESC) AS sales_rank
FROM products p
INNER JOIN sales s ON p.product_id = s.product_id
INNER JOIN inventory i ON p.product_id = i.product_id
LEFT JOIN reviews r ON p.product_id = r.product_id
LEFT JOIN supplier_orders so ON p.product_id = so.product_id AND so.status =
     'Pending'
WHERE p.category = 'Electronics'
  AND s.sale_date >= CURRENT_DATE - INTERVAL '30 days'
  AND i.current_stock < 0.2 * (SUM(s.quantity) / 3)  -- Assuming 3 months avg
     , but subquery needed for precision
GROUP BY p.product_id, p.name, i.current_stock
HAVING SUM(s.quantity) > 50 AND AVG(r.rating) > 4.0
ORDER BY sales_rank
LIMIT 10
```

Problem 2: Tracking Patient Health

In a hospital management system, retrieving patient data for epidemiological studies or quality control, such as identifying patients with chronic conditions who have had multiple readmissions, cross-referenced with treatment efficacy and demographic filters.

- **Scenario Details:** Fetch a report of diabetic patients aged 50+ who were readmitted within 90 days of discharge in the last year, including their average blood sugar levels from lab tests, prescribed medications, and doctor notes. Exclude cases with incomplete records and rank by readmission frequency.

```sql
WITH readmissions AS (
  SELECT patient_id, admission_date, discharge_date,
         LAG(discharge_date) OVER (PARTITION BY patient_id ORDER BY
     admission_date) AS prev_discharge
  FROM admissions
  WHERE diagnosis LIKE '%diabetes%'
)
SELECT p.patient_id, p.age, COUNT(r.patient_id) AS readmission_count,
       AVG(l.blood_sugar) AS avg_blood_sugar,
       STRING_AGG(m.medication_name, ', ') AS medications
FROM patients p
INNER JOIN readmissions r ON p.patient_id = r.patient_id
INNER JOIN lab_tests l ON p.patient_id = l.patient_id AND l.test_date BETWEEN
      r.admission_date AND r.discharge_date
LEFT JOIN prescriptions pr ON p.patient_id = pr.patient_id
LEFT JOIN medications m ON pr.medication_id = m.id
WHERE p.age >= 50
  AND r.admission_date >= CURRENT_DATE - INTERVAL '1 year'
  AND r.admission_date - r.prev_discharge <= 90  -- Readmission within 90
     days
  AND p.records_complete = TRUE
GROUP BY p.patient_id, p.age
HAVING COUNT(r.patient_id) > 1
ORDER BY readmission_count DESC
```