Databases
Lab 05

Andrés Calderón, Ph.D.

March 5, 2025

1 Introduction

In this lab, we will explore essential SQL concepts through interactive exercises and hands-on implementation. The primary focus will be on understanding and utilizing SQL Views, a powerful feature that enhances query abstraction, security, and maintainability. By working through structured exercises, students will gain practical experience in creating, modifying, and managing views, as well as understanding their role in simplifying database interactions.

The lab is divided into three key components:

1. Interactive SQL Practice: Utilizing the W3Schools SQL tutorial, students will reinforce fundamental SQL operations, including database and table creation, constraints, indexing, and security mechanisms.
2. SQL Views and Their Applications: A detailed examination of SQL Views, covering their creation, modification, and deletion. This section will demonstrate how views can encapsulate complex queries, restrict access to sensitive data, and optimize query performance.
3. Hands-on Exercises: Students will apply their knowledge by creating various types of SQL views, including basic, join-based, aggregated, updatable, and security-oriented views. These exercises will solidify their understanding of real-world database design and query structuring.

By the end of this lab, students will have a comprehensive grasp of how SQL Views work, their advantages in database management, and how they can be leveraged to simplify and optimize database operations. Through practical implementation, they will also develop proficiency in SQL scripting and database manipulation, essential skills for database management and development.

2 W3Schools Interactive SQL Tutorial

We will continue with the interactive SQL tutorial on the W3Schools website. This time we will cover the lessons in the section of SQL Database (look for it the the left panel). Remember that the tutorial can be found here. We will cover the following lessons:

1. Create DB
2. Drop DB
3. Create Table
4. Drop Table
5. Alter Table
6. Constrains

7. Not Null
8. Unique
9. Primary Key
10. Foreign Key
11. Check
12. Default
13. Index
14. Auto Increment
15. Dates
16. Views
17. Injection

In order to support your learning, you will have to not just create (or update) an account but also set a nice profile picture. For these lessons you will follow the content and execute the examples by you own. In this occasion, you must complete the set of exercise of each of the lessons. Once you have completed all the exercises, you will capture a screenshot showing the complete task. Be aware to capture your profile picture in the screenshot and attach them in your report. Watch this simple video to understand better what you have to do.

3 SQL Views: A Comprehensive Guide

A View in SQL is a virtual table that represents the result of a SQL query. Views do not store data themselves but provide an abstraction over the underlying tables, allowing users to simplify complex queries, enhance security, and improve maintainability.

Why Use Views?

- **Simplicity**: Encapsulate complex queries into a single reference.
- **Security**: Restrict access to sensitive data by exposing only necessary columns.
- **Reusability**: Avoid repeating SQL logic in multiple queries.
- **Data Integrity**: Ensure consistency across applications by centralizing query definitions.

3.1 Creating a Simple View

A basic view is created using the `CREATE VIEW` statement:

```
CREATE VIEW EmployeeInfo AS
  SELECT
    EmployeeID, FirstName, LastName, Department
  FROM
    Employees;
```

Now, you can retrieve data using:

```sql
SELECT * FROM EmployeeInfo;
```

3.1.1 Modifying an Existing View

If you need to update the definition of a view, use `CREATE OR REPLACE VIEW`:

```sql
CREATE OR REPLACE VIEW EmployeeInfo AS
  SELECT
    EmployeeID, FirstName, LastName, Department, Salary
  FROM
    Employees;
```

3.1.2 Dropping a View

To remove a view:

```sql
DROP VIEW EmployeeInfo;
```

3.2 Advanced View Concepts

3.2.1 Views with Joins

Views can combine data from multiple tables:

```sql
CREATE VIEW EmployeeDepartment AS
  SELECT
    e.EmployeeID, e.FirstName, e.LastName, d.DepartmentName
  FROM
    Employees e
  JOIN
    Departments d
  ON
    e.DepartmentID = d.DepartmentID;
```

3.2.2 Views with Aggregations

Views can aggregate data too:

```sql
CREATE VIEW DepartmentSalary AS
  SELECT
    DepartmentID, AVG(Salary) AS AvgSalary
```

```
  FROM
    Employees
  GROUP BY
    DepartmentID;
```

3.3 Updatable Views

Some views can be updated if they meet these conditions:

- Selects from a single table.
- Does not use `DISTINCT`, `GROUP BY`, or `HAVING`.
- Does not include aggregate functions.

Example:

```
CREATE VIEW EmployeeSalary AS
  SELECT
    EmployeeID, Salary
  FROM
    Employees;
```

Updating the view:

```
UPDATE
  EmployeeSalary
SET
  Salary = 60000
WHERE
  EmployeeID = 101;
```

3.4 Using Views for Security

You can restrict access by creating a view that excludes sensitive columns:

```
CREATE VIEW PublicEmployeeInfo AS
  SELECT
    EmployeeID, FirstName, LastName, Department
  FROM
    Employees;
```

Grant access only to this view:

```
GRANT SELECT ON PublicEmployeeInfo TO public_user;
```

3.5 Performance Considerations

- **Indexes**: Views do not have indexes, but queries using indexed columns in base tables perform well.
- **Materialized Views**: These store the result of a query and improve performance for complex calculations (in databases like PostgreSQL, Oracle).

Example (PostgreSQL):

```
CREATE MATERIALIZED VIEW SalesSummary AS
  SELECT
    ProductID, SUM(Amount) AS TotalSales
  FROM
    Sales
  GROUP BY
    ProductID;
```

To refresh it:

```
REFRESH MATERIALIZED VIEW SalesSummary;
```

SQL Views provide a powerful way to structure queries, simplify access to data, and enforce security rules. Understanding when and how to use them can significantly improve database design and maintainability.

4 SQL Script: Working with Views

4.1 Step 1: Create Sample Tables and Insert Data

```
-- Drop existing tables if they exist (to reset the example)
DROP TABLE IF EXISTS Employees;
DROP TABLE IF EXISTS Departments;

-- Create Departments Table
CREATE TABLE Departments (
    DepartmentID INT PRIMARY KEY,
    DepartmentName VARCHAR(100)
);

-- Create Employees Table
CREATE TABLE Employees (
```

```
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50),
    DepartmentID INT,
    Salary DECIMAL(10,2),
    HireDate DATE,
    FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID)
);

-- Insert sample data into Departments
INSERT INTO Departments (DepartmentID, DepartmentName) VALUES
(1, 'Engineering'),
(2, 'Human Resources'),
(3, 'Finance');

-- Insert sample data into Employees
INSERT INTO Employees (EmployeeID, FirstName, LastName, DepartmentID, Salary, HireDate) VALUES
(101, 'Alice', 'Johnson', 1, 75000.00, '2019-05-22'),
(102, 'Bob', 'Smith', 2, 65000.00, '2020-08-15'),
(103, 'Charlie', 'Brown', 1, 80000.00, '2018-07-30'),
(104, 'Diana', 'Martinez', 3, 72000.00, '2021-03-10');
```

4.2 Step 2: Create and Use SQL Views

4.2.1 Basic View: Employee Information

```
CREATE VIEW EmployeeInfo AS
SELECT EmployeeID, FirstName, LastName, DepartmentID
FROM Employees;
```

Query the view:

```
SELECT * FROM EmployeeInfo;
```

4.2.2 View with Joins: Employee with Department Name

```
CREATE VIEW EmployeeDepartment AS
SELECT e.EmployeeID, e.FirstName, e.LastName, d.DepartmentName
FROM Employees e
JOIN Departments d ON e.DepartmentID = d.DepartmentID;
```

Query the view:

```
SELECT * FROM EmployeeDepartment;
```

4.2.3 Aggregation View: Average Salary by Department

```
CREATE VIEW AvgDepartmentSalary AS
SELECT d.DepartmentName, AVG(e.Salary) AS AvgSalary
FROM Employees e
JOIN Departments d ON e.DepartmentID = d.DepartmentID
GROUP BY d.DepartmentName;
```

Query the view:

```
SELECT * FROM AvgDepartmentSalary;
```

4.2.4 Updatable View: Employee Salaries

```
CREATE VIEW EmployeeSalary AS
SELECT EmployeeID, Salary FROM Employees;
```

Update an employee's salary through the view:

```
UPDATE EmployeeSalary SET Salary = 78000 WHERE EmployeeID = 101;
```

Verify the update:

```
SELECT * FROM Employees WHERE EmployeeID = 101;
```

4.3 Step 3: Managing Views

4.3.1 Modify an Existing View

```
CREATE OR REPLACE VIEW EmployeeInfo AS
SELECT EmployeeID, FirstName, LastName, DepartmentID, Salary
FROM Employees;
```

Query the updated view:

```sql
SELECT * FROM EmployeeInfo;
```

4.3.2 Delete a View

```sql
DROP VIEW EmployeeInfo;
```

Bonus: Materialized View (PostgreSQL & Oracle)

If you're using **PostgreSQL** or **Oracle**, you can create a **materialized view**:

```sql
CREATE MATERIALIZED VIEW TotalSalaries AS
SELECT DepartmentID, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY DepartmentID;
```

To refresh the materialized view:

```sql
REFRESH MATERIALIZED VIEW TotalSalaries;
```

Summary of What You Learned

- Creating basic `SQL` views.
- Using views with `JOIN`s.
- Aggregating data with `GROUP BY`.
- Updating data through updatable views.
- Managing views (modifying and deleting).
- Using materialized views for better performance.

5 Advanced SQL Views Tutorial

5.1 Filtering Data in Views

5.1.1 Example: View for Recently Hired Employees

```sql
CREATE VIEW RecentHires AS
SELECT EmployeeID, FirstName, LastName, HireDate
FROM Employees
WHERE HireDate >= '2020-01-01';
```

Query the view:

```
SELECT * FROM RecentHires;
```

5.1.2 Example: View for High-Salary Employees

```
CREATE VIEW HighEarners AS
SELECT EmployeeID, FirstName, LastName, Salary
FROM Employees
WHERE Salary > 70000;
```

Query the view:

```
SELECT * FROM HighEarners;
```

5.2 Using Multiple Joins in Views

```
CREATE VIEW EmployeeFullInfo AS
SELECT e.EmployeeID, e.FirstName, e.LastName, d.DepartmentName, e.Salary
FROM Employees e
JOIN Departments d ON e.DepartmentID = d.DepartmentID;
```

Query the view:

```
SELECT * FROM EmployeeFullInfo;
```

5.3 Views with Aggregations and GROUP BY

5.3.1 Example: Total Salary per Department

```
CREATE VIEW DepartmentSalarySummary AS
SELECT d.DepartmentName, SUM(e.Salary) AS TotalSalary
FROM Employees e
JOIN Departments d ON e.DepartmentID = d.DepartmentID
GROUP BY d.DepartmentName;
```

Query the view:

```sql
SELECT * FROM DepartmentSalarySummary;
```

5.3.2 Example: Count of Employees per Department

```sql
CREATE VIEW EmployeeCountByDepartment AS
SELECT d.DepartmentName, COUNT(e.EmployeeID) AS TotalEmployees
FROM Employees e
JOIN Departments d ON e.DepartmentID = d.DepartmentID
GROUP BY d.DepartmentName;
```

Query the view:

```sql
SELECT * FROM EmployeeCountByDepartment;
```

5.4 Security with Views

5.4.1 Example: Hide Salary Information

```sql
CREATE VIEW PublicEmployeeInfo AS
SELECT EmployeeID, FirstName, LastName, DepartmentID
FROM Employees;
```

Granting limited access:

```sql
GRANT SELECT ON PublicEmployeeInfo TO public_user;
```

5.5 Using Views in Subqueries

5.5.1 Example: Employees Earning Above the Department Average

```sql
CREATE VIEW AboveAverageEarners AS
SELECT e.EmployeeID, e.FirstName, e.LastName, e.Salary, d.DepartmentName
FROM Employees e
JOIN Departments d ON e.DepartmentID = d.DepartmentID
WHERE e.Salary > (
    SELECT AVG(Salary) FROM Employees WHERE DepartmentID = e.DepartmentID
);
```

Query the view:

```
SELECT * FROM AboveAverageEarners;
```

5.6 Materialized Views for Performance Optimization (PostgreSQL & Oracle)

5.6.1 Example: Materialized View for Total Sales

```
CREATE MATERIALIZED VIEW SalesSummary AS
SELECT ProductID, SUM(Amount) AS TotalSales
FROM Sales
GROUP BY ProductID;
```

Refreshing a materialized view:

```
REFRESH MATERIALIZED VIEW SalesSummary;
```

Summary

- Filtering data in views.
- Joining multiple tables.
- Grouping and aggregations.
- Using views for security.
- Employing views in subqueries.
- Optimizing performance with materialized views.

6 Hands-On Exercises

Exercise 1: Create a View for Employees in a Specific Department

Task:

Create a view called 'EngineeringEmployees' that contains only employees from the *Engineering* department.

Expected Output:

When running:

```
SELECT * FROM EngineeringEmployees;
```

You should see only employees from the Engineering department.

Hint:

Use a `JOIN` between Employees and Departments and filter by DepartmentName = 'Engineering'.

Exercise 2: Create a View for Employees Hired After a Certain Date

Task:

Create a view called 'NewHires' that includes employees hired after **January 1, 2021**.

Expected Output:

When running:

```
SELECT * FROM NewHires;
```

You should see only employees hired in 2021 or later.

Hint:

Use a `WHERE` clause with the 'HireDate' column.

Exercise 3: Create a View Showing Employees with Above-Average Salaries

Task:

Create a view called 'AboveAverageSalaries' that includes employees earning more than the **company-wide** average salary.

Expected Output:

When running:

```
SELECT * FROM AboveAverageSalaries;
```

Only employees earning more than the average salary should be listed.

Hint:

Use a subquery with `AVG(Salary)`.

Exercise 4: Create an Aggregation View

Task:

Create a view called 'DepartmentStats' that shows each department's **total salary and the number of employees**.

Expected Output:

When running:

```sql
SELECT * FROM DepartmentStats;
```

You should see something like:

DepartmentName	TotalSalary	EmployeeCount
Engineering	155000	2
HR	65000	1
Finance	72000	1

Hint:

Use `SUM(Salary)` and `COUNT(EmployeeID)`, and group by 'DepartmentName'.

Exercise 5: Create a Security-Based View

Task:

Create a view called 'PublicEmployees' that **hides salary information** and only shows EmployeeID, FirstName, LastName, and DepartmentName. Then, grant access to this view for a user named `readonly_user`.

Hint:

Use a `JOIN` between Employees and Departments and `GRANT SELECT` to give access.

Exercise 6: Create an Updatable View

Task:

Create a view called 'EmployeeSalaries' that includes 'EmployeeID' and 'Salary'. Then, **update** an employee's salary through this view.

Hint:

The view must only include one table (Employees). Use `UPDATE` to modify a salary.

Exercise 7 (Advanced): Create a Materialized View (PostgreSQL/Oracle)

Task:

Create a **materialized view** called 'SalarySummary' that stores each department's **average salary**.

Hint:

Use `CREATE MATERIALIZED VIEW` and `REFRESH MATERIALIZED VIEW` when needed.

Bonus Challenge: Combining Multiple Concepts

Design a view that:

- Includes employees **earning above the department average salary**.
- Hides **salaries** but includes an **"Above Average"** column ('Yes/No').
- Groups results by department.

For each query, you must present the `SQL` code and the query result (either copy-pasted or as a screenshot). We expect you to submit a well-structured report in PDF format, containing the requested information from the previous sections. Additionally, include the `SQL` code with the answers to the queries in a ZIP file. The submission deadline is **March 19, 2025**.

Happy Hacking ☺!