Advanced SQL Programming

TRƯỜNG ĐẠI HỌC QUỐC TẾ MIỀN ĐÔNG
EASTERN INTERNATIONAL UNIVERSITY

Week 4, Topic 3, Lesson 7 and Lesson 8

- Embedded SQL
- Dynamic SQL
- Stored Procedures and Functions

Agenda (what we will build)

TRƯỜNG ĐẠI HỌC QUỐC TẾ MIỀN ĐÔNG
EASTERN INTERNATIONAL UNIVERSITY

- Embedded SQL: host variables, cursors, transactions
- Dynamic SQL: safe patterns (parameters, quoting, whitelists)
- Embed SQL into app code using host variables and cursors
- Generate dynamic SQL safely (avoid SQL injection)

Sample schema used in examples

You can adapt these patterns to your own tables.

```
-- Minimal schema (compact)
CREATE TABLE departments(dept_id int primary key, dept_name text not null);

CREATE TABLE employees(
  emp_id int primary key,
  dept_id int references departments(dept_id),
  full_name text not null,
  email text unique,
  salary numeric(12,2) not null default 0,
  updated_at timestamptz not null default now()
);

CREATE TABLE products(product_id int primary key, name text not null, price numeric(12,2) not null);
CREATE TABLE inventory(product_id int primary key references products(product_id), qty_on_hand int not null);

CREATE TABLE orders(order_id bigint generated always as identity primary key,
  customer_id int not null, order_date date not null default current_date);

CREATE TABLE order_items(order_id bigint references orders(order_id),
  product_id int references products(product_id), quantity int not null,
  unit_price numeric(12,2) not null, primary key(order_id, product_id));
```

Embedded SQL: what it is

- You write SQL in your program; a precompiler/driver sends it to the DB
- Host variables carry values between your code and SQL
- Best practice: use parameter markers (don’t string-concatenate values)
- Common patterns: SELECT INTO, INSERT/UPDATE, cursors, transactions
- Embedded SQL means writing SQL statements inside a host programming language such as C, C++, Java, COBOL, or Python.

Why use embedded SQL?

You keep SQL close to the app logic while still letting the database do set-based work.

```
-- Key idea: values come from host variables (not string concat)
SELECT full_name, salary
FROM employees
WHERE dept_id = ? AND salary > ?
ORDER BY salary DESC;
```

Embedded SQL...

Structure -

```
EXEC SQL
  SQL statement
END-EXEC;
```

- Variables from the program are shared with SQL using host variables (preceded by :).

Embedded SQL Example using Java – JDBC style

```
Connection con = DriverManager.getConnection(
  "jdbc:mysql://localhost:3306/company", "user", "pass");

PreparedStatement ps =
  con.prepareStatement("SELECT name FROM employee WHERE id=?");

ps.setInt(1, 101);

ResultSet rs = ps.executeQuery();

while (rs.next()) {
  System.out.println(rs.getString("name"));
}
```

Embedded SQL: host variables

"Host variables" are placeholders bound from your programming language.

Pattern

1) Declare host variables 2) Bind values 3) Run SQL 4) Read output
2) Variables declared in the host language (Java)
3) Used to pass data to SQL queries and retrieve results
4) Replaced at runtime with actual values
5) Host variables in Java are represented using **?** Placeholders
6) Values are set using PreparedStatement

Why Host Variables?

- Improves security (prevents SQL Injection)
- Enhances performance (precompiled queries)
- Makes code dynamic and reusable

SQL in MySQL Workbench (NOT Embedded SQL)-
-SELECT name FROM student WHERE roll_no = 1;

- This is normal SQL
- NO Embedded SQL (no programming language)

```
/* JAVA (idea is similar in many languages) */

int rollNo = 1;  // Host variable

String query = "SELECT name FROM student WHERE roll_no = ?";
PreparedStatement ps = con.prepareStatement(query);
ps.setInt(1, rollNo);  // Using host variable

ResultSet rs = ps.executeQuery();
```

Embedded SQL: Cursor loop

Use a cursor when you must process rows one-by-one in the host language.

- A cursor is a temporary pointer used to retrieve one row at a time from a result set.
- Used when a query returns multiple rows.
- A loop that repeatedly fetches rows from a cursor
- Continues until no more rows are left
- Declare cursor for a SELECT query
- OPEN → FETCH in a loop → CLOSE
- Prefer set-based SQL when possible; cursors can be slower

Why Cursor Loop is Needed

- Process records row by row
- Useful for calculations, validations, or updates

❑ Declare: A cursor is declared within a stored procedure or function using a CURSOR statement. This binds the cursor to a specified SQL query.

```
DECLARE cursor_name CURSOR FOR select_statement;
```

- cursor_name: This will be the name given to your cursor.
- select_statement: SQL statement to define a result set for the cursor.

Open: open the cursor before you fetch rows from it. You do this with the OPEN statement.

```
OPEN cursor_name;
```

Fetch the Data from the Cursor

The FETCH statement retrieves the data from the cursor and moves the cursor to the next line in the result set; it loads the data into variables.

Embedded SQL: Cursor loop

```
FETCH cursor_name INTO variable1,
variable2, ...;
```

- cursor_name : Name of the cursor.
- variable1, variable2, ... : Variables in which the fetched data has to be stored.

Close Cursor

Finally, you would close the cursor after you have processed all the data, so that the resources that are allocated for it will be released.

Use case: Display all student names one by one using a cursor.

```
DELIMITER $$
CREATE PROCEDURE show_students()
BEGIN
  DECLARE done INT DEFAULT 0;
  DECLARE s_name VARCHAR(50);

  -- Declare cursor
  DECLARE student_cursor CURSOR FOR
    SELECT name FROM student;

-- Handler to stop loop
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET
done = 1;

  -- Open cursor
  OPEN student_cursor;

 -- Cursor loop
  read_loop: LOOP
    FETCH student_cursor INTO s_name;

    IF done = 1 THEN
      LEAVE read_loop;
    END IF;

    -- Process each row
    SELECT s_name;
  END LOOP;

  -- Close cursor
  CLOSE student_cursor;
END$$

DELIMITER ;
```

Transaction control with Cursor example -

- Transfer a bonus of 500.000 to all employees in the Sales department
- If any error occurs, rollback the entire transaction

```
DELIMITER $$

CREATE PROCEDURE give_bonus()
BEGIN
   DECLARE done INT DEFAULT 0;
   DECLARE emp_id INT;

   -- Cursor declaration
   DECLARE emp_cursor CURSOR FOR
      SELECT id FROM employee WHERE department = 'Sales';

   -- Handler for cursor end
   DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

 -- Start transaction
   START TRANSACTION;

   OPEN emp_cursor;

bonus_loop: LOOP
      FETCH emp_cursor INTO emp_id;

      IF done = 1 THEN
         LEAVE bonus_loop;
      END IF;

      -- Update salary
      UPDATE employee
      SET salary = salary + 500
      WHERE id = emp_id;
   END LOOP;

   CLOSE emp_cursor;

   -- Commit transaction
   COMMIT;
END$$

DELIMITER ;
```

Execute -

```
CALL give_bonus();
```

Dynamic SQL: what it is

Building a SQL statement at runtime (string) and executing it.

- Dynamic SQL is a powerful SQL programming technique that allows us to construct and execute SQL statements at runtime.
- These statements are often used when the exact SQL query cannot be determined during the development phase, such as when working with user inputs or dynamic database objects.

Syntax –
EXEC sp_executesql N'SELECT statement';
*(*N prefix indicates that the SQL statement is treated as a Unicode string.*)*

Steps to use Dynamic SQL
1. Declare Variables: Declare two variables, @var1 for holding the name of the table and @var 2 for holding the dynamic SQL :
DECLARE
@var1 NVARCHAR(MAX),
@var2 NVARCHAR(MAX);

2. Assign Values to Variables: Set the value of the @var1 variable to table_name :
SET @var1 = N'table_name';

Dynamic SQL...

- Construct the SQL Statement: Create the dynamic SQL by adding the SELECT statement to the table name parameter :

```
SET @var2= N'SELECT *
FROM ' + @var1;
```

- Execute the SQL Statement: Run the sp_executesql stored procedure by using the @var2 parameter:

```
EXEC sp_executesql @var2;
```

Example –

```
SET @table = 'employee';
SET @sql = CONCAT('SELECT * FROM ', @table);

PREPARE stmt FROM @sql;
EXECUTE stmt;
DEALLOCATE PREPARE stmt;
```

Pros and Cons of Embedded SQL

Advantages -

- Small footprint database: As embedded SQL uses an UltraLite database engine compiled specifically for each application, the footprint is generally smaller than when using an UltraLite component, especially for a small number of tables. For a large number of tables, this benefit is lost.
- High performance: Combining the high performance of C, C++, JAVA, Python applications with the optimization of the generated code, including data access plans, makes embedded SQL a good choice for high-performance application development.
- Extensive SQL support: With embedded SQL you can use a wide range of SQL in your applications.

Disadvantages -

- Knowledge of host programming language required: If you are not familiar with host programming, you may wish to use one of the other UltraLite interfaces. UltraLite components provide interfaces from several popular propgramming languages and tools.
- Complex development model: The use of a reference database to hold the UltraLite database schema, together with the need to preprocess your source code files, makes the embedded SQL development process complex. The UltraLite components provide a much simpler development process.
- SQL must be specified at design time: Only SQL statements defined at compile time can be included in your application. The UltraLite components allow dynamic use of SQL statements.

Local procedures

- ❑ A Local Stored Procedure is a temporary stored procedure.
- ❑ It is created inside a session and exists only for that session.
- ❑ It is automatically deleted when the session ends.
- ❑ Name starts with # (example: #MyProcedure).
- ❑ Used for temporary tasks, testing, or intermediate calculations.

```
DELIMITER $$

CREATE PROCEDURE AddNumbers()
BEGIN
   DECLARE a INT DEFAULT 10;
   DECLARE b INT DEFAULT 20;
   DECLARE sum INT;

   SET sum = a + b;
   SELECT sum AS Result;
END $$

DELIMITER ;
```

Note - a, b, and sum are local variables and cannot be used outside the procedure.

Advanced SQL Programming : Stored procedures

- ❑ A Stored Procedure is a precompiled collection of SQL statements stored in the database.
- ❑ It is executed as a single unit to perform specific tasks (CRUD operations, business logic).
- ❑ Stored procedures accept input parameters and can return output values.
- ❑ Commonly used in databases like MySQL, SQL Server, Oracle, PostgreSQL.
- ❑ Contain control flow statements (IF, WHILE, etc.)
- ❑ Modify database state
- ❑ It accept Reusability (Write once, use multiple times.)
- ❑ It is used for such as Transaction control, Executing sequence of statements, Data modification
- ❑ Each parameter is an IN parameter by default. To specify otherwise for a parameter, use the keyword OUT or INOUT before the parameter name.

Advanced SQL Programming : When to Use Stored Procedures

- ❑ Complex Operations: When you need to perform a series of SQL operations as a single unit.
- ❑ Data Integrity: For operations that require multiple steps to maintain data consistency.
- ❑ Security: To restrict direct access to tables and provide a controlled interface to the data.
- ❑ Performance: To reduce network traffic by sending only the call to the procedure instead of multiple SQL statements.
- ❑ Maintenance: When you want to centralize business logic for easier updates and maintenance.

Example -

```
DELIMITER $$

CREATE PROCEDURE get_students()
BEGIN
   SELECT * FROM student;
END $$

DELIMITER ;
```

Execute

```
call get_students()
```

Stored Procedures: IN, OUT, INOUT

Example –

```
delimiter $$
CREATE PROCEDURE birthday_count( IN bday date, OUT count int)
BEGIN
SET count = (SELECT count(*) FROM birthdays WHERE birthday = bday);
END
Delimiter;
```

EXECUTE –

```
SET @count = 0;
call birthday_count('1985-01-10', @count);
SELECT @count;
```

Output: 2

Advanced SQL Programming : Functions

- A Function returns a single value
- Must return a value using RETURN
- Commonly used in: SELECT statements , Calculations and data formatting
- Cannot modify database data (in most DBs)
- Example. - Calculating tax, age, or total price

```
DELIMITER $$
CREATE FUNCTION emp_name_max_salary() RETURNS VARCHAR(50)
DETERMINISTIC NO SQL READS SOL DATA
BEGIN
        DECLARE V_max INT;
        DECLARE V_emp_name VARCHAR (50) ;
        SELECT MAX(salary) into v_max FROM employees;
        SELECT fame into v_emp_name FROM employees WHERE salary=v_max;
        return v_emp_name;
END$$
DELIMITER;
```

Output –
SELECT emp_name_max_salary();

Name of the employee who has highest salary

Functions - When to use ?

- ❑ Calculations: For complex calculations that you want to reuse across multiple queries.
- ❑ Data Transformation: To standardize data formatting or transformation logic.
- ❑ Custom Aggregations: When you need custom aggregation logic that isn’t available in built-in MySQL functions.
- ❑ Encapsulation: To encapsulate complex logic that returns a single value and can be used in SELECT statements.