Stored Procedures

DSC 301: Lecture 16

April 9, 2021

Abstract

Stored programs are extensions to standard SQL, specifically they include: *stored procedures, stored functions, triggers*, and *events*. Stored programs can control the flow of execution, can be called from other applications or within a SQL statement. *Triggers* are executed automatically when an `INSERT`, `UPDATE`, or `DELETE` statement is run on a table. *Events* are scheduled and execute at that time.

Lecture Objectives

In this lesson you will learn about *stored procedures*. Specifically, we cover:

- Why use stored procedures?
- Syntax to create and call a stored procedure
  - Flow control keywords
  - Use `INTO` clause to store results in variables

Why use stored procedures?

There are four basic reasons to use stored procedures:

- Simplicity - complicated workflows with multiple queries can be bundled into one procedure.
  - For example, processing a customer order involves computing tax, total, and shipping as well as updating inventory.
- Consistency - Writing a query one time vs. 100 times reduces the probability of errors. All users execute the same procedure (SQL statements), the results will be the same each time.
- Security - stored procedures can be executed without user having access to underlying data to prevent data corruption (either by accident or intention)

- Increase performance - compiled and stored

NOTE: There are a few drawbacks to stored procedures. First, different DBMS use different syntax making portability difficult. Second, stored procedures add a level of complexity that require greater degree of skill and knowledge than simple SQL statements.

Syntax to create and call a stored procedure

Syntax **template** for creating a stored procedure without input parameters.

```
USE whichDatabase;

# Change the delimiter because semi-colon used inside
DELIMITER $$

CREATE PROCEDURE name_of_proc()

BEGIN
    # Declare local variables if needed
    DECLARE  localVar Type;

    # Write SQL statement(s)

END$$

# Change delimiter back to semicolon
DELIMITER ;
```

How to call a stored procedure

Call the stored procedure using the `CALL` keyword.

```
CALL name_of_proc();
```

Example 1. *Create a stored procedure that displays a message (i.e., no database).*

```
DELIMITER $$

CREATE PROCEDURE msg()

BEGIN
    SELECT 'This is a message that will be displayed' as Message;
END$$

DELIMITER ;
```

Example 2. *Create and call a stored procedure* `test()` *that prints the message, "This is a stored procedure."*

Using our template

```
USE dbsoln_Store;

# Change the delimiter because semi-colon used inside
DELIMITER $$

CREATE PROCEDURE test()

BEGIN
    SELECT "This is a stored procedure."
END$$

# Change delimiter back to semicolon
DELIMITER ;
```

Then, `CALL test();`

Template for creating a stored procedure WITH input parameters

```
USE whichDatabase;

# Change the delimiter because semi-colon used inside
DELIMITER $$

CREATE PROCEDURE name_of_proc(
    IN  x           type,
    IN  y           type
)

BEGIN
    # Declare local variables if needed
    DECLARE  localVar Type;

    # Write SQL statement(s)

END$$

# Change delimiter back to semicolon
DELIMITER ;
```

Template WITH input and OUTPUT parameters

```
USE whichDatabase;
```

```
# Change the delimiter because semi-colon used inside
DELIMITER $$

CREATE PROCEDURE name_of_proc(
    IN      x           type,
    IN      y           type,
    OUT     z           type
)

BEGIN
    # Declare local variables if needed
    DECLARE  localVar Type;

    # Write SQL statement(s)

END$$

# Change delimiter back to semicolon
DELIMITER ;
```

Declare and set variables

A *variable* stores a value that can be used and changed during execution of a stored procedure. First, a (local) variable must be declared before it can be used. To declare a variable, use the `DECLARE` keyword after the `BEGIN` statement of the procedure body. A data type that matches the column type must be specified for each declared variable. For example, a variable for average price should be `DECIMAL(9,2)`. Data types can be any data type used in column definition (e.g., `INT`, `VARCHAR()`, `DECIMAL()`, etc.). A default value can be assigned when declaring variables by using the `DEFAULT` keyword. For example, `DECLARE minMSRP DECIMAL(9,2) DEFAULT 1.00`, could be used to declare a minimum MSRP for the variable `minMSRP` to one dollar.

Example 3. *Create a stored procedure that displays a message (i.e., no database).*

```
DELIMITER $$

CREATE PROCEDURE msg()

BEGIN
    SELECT 'This is a message that will be displayed' as Message;
END$$

DELIMITER ;
```

Flow Control

Flow control statements include branching and looping statements. The following

```
IF...ELSEIF...ELSE          # Conditional
CASE...WHEN...ELSE          # Conditional
WHILE...DO...LOOP           # Repetition
REPEAT...UNTIL...END REPEAT # Repetition
DECLARE CURSOR FOR          # Result set for looping
DECLARE...HANDLER           # Error handler
```

IF Statements

```
DELIMITER $$

CREATE PROCEDURE procIF()

BEGIN
    DECLARE flight_date DATE;

    SELECT date INTO flight_date FROM Flights WHERE fid = 1;

    IF flight_date > NOW() THEN
        SELECT 'Missed your flight.  Sorry';
    END IF
END$$

DELIMITER ;
```

CASE Statements

```
DELIMITER $$

CREATE PROCEDURE procLOOP()

BEGIN
    DECLARE airline CHAR(2);

    SELECT carrier INTO airline FROM Flights WHERE fid = 1;

    CASE airline
        WHEN 'AA' THEN
            SELECT 'You fly American Airlines';
        WHEN 'UA' THEN
            SELECT 'You fly United Airlines';
        ELSE
            SELECT 'You fly some other airlines';
        END CASE;
```

```
END$$

DELIMITER ;
```

Repetition Structures (Looping)

There are three looping structures: `WHILE`, `REPEAT`, and just `LOOP`.

WHILE Loop

```
DELIMITER $$

CREATE PROCEDURE procWHILE()

BEGIN
    DECLARE i INT DEFAULT 1;
    DECLARE s VARCHAR(45) DEFAULT ' ';

    WHILE i < 10 DO
        SET s = CONCAT_WS(',', s, i);
        SET i = i +  1;
    END WHILE;

    SELECT s AS Result;
END$$

DELIMITER ;
```

REPEAT Loop

```
DELIMITER $$

CREATE PROCEDURE procREPEAT()

BEGIN
    DECLARE i INT DEFAULT 1;
    DECLARE s VARCHAR(45) DEFAULT ' ';

    REPEAT
        SET s = CONCAT_WS(',', s, i);
        SET i = i +  1;
    UNTIL i = 10
    END REPEAT;

    SELECT s AS Result;
END$$

DELIMITER ;
```

CURSOR

A cursor is a row-based operation to obtain subset of data by looping through each row one at a time. The steps to work with cursors are:

1. DECLARE cursor - `DECLARE cursorname CURSOR FOR`, then specify the `SELECT` statement.
2. OPEN cursor - Opens cursor
3. FETCH cursor - fetch row from the cursor into a program variable
4. CLOSE cursor - when operations are complete

```
DELIMITER $$
USE `dbsoln_Store`$$
CREATE PROCEDURE `UpdateInventory`
(
       IN oid   int
)
   BEGIN
    declare xProduct INT;
    declare xInventory INT;
    declare row_not_found tinyint default false;
    declare update_count INT default 0;

    Declare curInventory CURSOR FOR
        select D.product, D.qty from OrderDetails D where `order`=
            oid;

    Declare continue handler for not found
        SET row_not_found = TRUE;

    OPEN curInventory;

    while row_not_found = FALSE DO
        fetch curInventory into xProduct, xInventory;
        update Products set inventory = inventory - xInventory
        where product_id = xProduct;
        set update_count = update_count + 1;
    end while;

    close curInventory;

    select concat(update_count, ' row(s) updated.');

    END$$

DELIMITER ;
```

```
# Test
# call UpdateInventory(2)
```