8.3 Procedural SQL

This section has been set as optional by your instructor.

Overview

Procedural SQL is an extension of the SQL language that includes procedures, functions, conditionals, and loops. Procedural SQL is intended for database applications and does not offer the full capabilities of general-purpose languages such as Java, Python, and C.

Procedural SQL can, in principle, be used for complete programs. More often, however, the language is used to create procedures that interact with the database and accomplish limited tasks. These procedures are compiled by and stored in the database, and therefore are called ***stored procedures***.

Stored procedures can be called from a command line or a host program written in another language. Host program calls to stored procedures can be coded with embedded SQL or built into an API.

SQL/Persistent Stored Modules (***SQL/PSM***) is a standard for procedural SQL that extends the core SQL standard. SQL/PSM is implemented in many relational databases with significant variations and different names. Many names incorporate the acronym 'PL', which stands for 'Procedural Language'.

MySQL conforms closely to the SQL/PSM standard. MySQL procedural SQL is considered part of the SQL language and does not have a special name. This section describes the MySQL implementation.

Table 8.3.1: Procedural SQL.

Database	Programming language
Oracle Database	PL/SQL
SQL Server	Transact-SQL
DB2	SQL PL
PostgreSQL	PL/pgSQL
MySQL	SQL

Feedback?

PARTICIPATION ACTIVITY 8.3.1: Procedural SQL.

1) Procedural SQL is compiled in two steps, with a precompiler followed by a compiler.
   - ○ True
   - ◉ False

Correct

Embedded SQL requires a precompiler for SQL, followed by a compiler for the host language. Procedural SQL is an extension of SQL and compiled in one step.

2) A stored procedure is compiled every time the procedure is called.
   - ○ True
   - ◉ False

Correct

Stored procedures are compiled only once when created. The compiled program is stored in the database for use by any calling program.

3) Procedural SQL languages differ significantly, depending on the database.
   - ◉ True
   - ○ False

Correct

Most Procedural SQL languages are based on the SQL standard and share many basic capabilities. However, most languages do not support all standard features and have specialized extensions that differ from other procedural SQL languages.

4) Most Procedural SQL languages implement the entire SQL/PSM standard.
   - ○ True
   - ◉ False

Correct

The SQL/PSM standard is complex and has evolved after leading Procedural SQL languages were created. Most Procedural SQL languages do not implement the entire standard.

Feedback?

Stored procedures

A stored procedure is declared with a ***CREATE PROCEDURE*** statement, which includes the procedure name, optional parameter declarations, and the procedure body.

A ***parameter declaration*** has an optional `IN`, `OUT`, or `INOUT` keyword, a ParameterName, and a Type. The Type is any data type supported by the database. The `IN`, `OUT`, and `INOUT` keywords limit the parameter to input, output, or both. If no keyword appears, the parameter is limited to input.

The procedure ***body*** consists of either a simple statement, such as `SELECT` and `DELETE`, or a compound statement. Compound statements, described below, are commonly used in stored procedures to code complex database tasks.

Figure 8.3.1: CREATE PROCEDURE statement.

```
CREATE PROCEDURE ProcedureName ( <parameter-declaration>, <parameter-declaration>,
... )
Body;

<parameter-declaration>:
[ IN | OUT | INOUT ] ParameterName Type
```

Feedback?

A stored procedure is executed with a ***CALL*** statement. The `CALL` statement includes the procedure name and a parameter list. The types of the CALL parameters must be compatible with the types of the corresponding `CREATE PROCEDURE` parameters.

Figure 8.3.2: CALL Statement.

```
CALL ProcedureName( ParameterName, ParameterName, ... )
```

Feedback?

Stored procedures can be called from:

- Other stored procedures
- The MySQL Command-Line Client
- C, C++, Java, Python, and other programming languages

When called from another stored procedure, call arguments are literals, like 10 or 'alpha', or stored procedure variables as described below. When called from the command line, call arguments are literals or user-defined variables. A ***user-defined variable*** is an SQL variable that

must begin with an @ character. When called from a different programming language, call parameter syntax varies greatly, depending on the language and API.

PARTICIPATION ACTIVITY 8.3.2: Stored procedure call from the command line.

1 2 3 4 5 2x speed

```
CREATE PROCEDURE FlightCount(IN airline VARCHAR(20), OUT quantity INT)
   SELECT COUNT(*)
   INTO quantity
   FROM Flight
   WHERE AirlineName = airline;
```

```
mysql> CALL FlightCount('British Airways', @result);
1 row in set (0.00 sec)                                  41

mysql> SELECT @result;
+----------+
| @result  |
+----------+
| 41       |
+----------+
1 row in set (0.00 sec)
```

The value of the user-defined variable @result is displayed with a SELECT statement. The Flight table has 41 British Airways flights.

Captions

1. The FlightCount stored procedure has an input parameter airline and an output parameter quantity.
2. The stored procedure body is a single SELECT statement.
3. The INTO clause assigns the COUNT(*) value to the quantity parameter.
4. The stored procedure is called from the command line. airline is assigned 'British Airways'. @result is assigned the resulting quantity value.
5. The value of the user-defined variable @result is displayed with a SELECT statement. The Flight table has 41 British Airways flights.

Feedback?

PARTICIPATION ACTIVITY 8.3.3: Stored procedures.

Match the language element to the description.

@	First character in a user-defined variable. The @ character is used to name a user-defined variable . A user-defined variable may be used as an argument when calling stored procedures from the command line.	Correct
body	Consists of a simple or compound statement. The body of a stored procedure is formally one statement. However, a statement may be simple or compound. Compound statements enable complex behavior in stored procedures.	Correct
CALL	May appear in the body of a stored procedure. When one stored procedure calls another, a CALL statement appears in the stored procedure body.	Correct
INOUT	Precedes a parameter name in a <parameter-declaration>. The IN, OUT, and INOUT keywords precede parameter names in a parameter declaration.	Correct
<parameter-declaration>	Optional in CREATE PROCEDURE statements. A stored procedure may have no parameters, leaving empty parentheses with no parameter declarations .	Correct

Reset

Feedback?

Compound statements

A ***compound statement*** is a series of statements between a `BEGIN` and an `END` keyword. The statements may be variable declarations, assignment statements, if statements, while loops, and other common programming constructs.

The ***DECLARE*** statement creates variables for use inside stored procedures. A variable has a name, data type, and an optional default value. Variable scope extends from the `DECLARE` statement to the next `END` keyword. Variables should not have the same name as a table column.

Figure 8.3.3: DECLARE statement.

```
DECLARE VariableName Type [ DEFAULT Value ];
```

Feedback?

The ***SET*** statement assigns a variable with an expression. The variable must be declared prior to the `SET` statement, and the expression may be any valid SQL expression.

Figure 8.3.4: SET statement.

```
SET VariableName = <expression>;
```

Feedback?

The ***IF*** statement is similar to statements in other programming languages. The statement includes a logical expression and a statement list. If the expression is true, the statements in the list are executed. If the expression is false, the statements are ignored.

`IF` statements have optional `ELSEIF` and `ELSE` clauses. `ELSEIF` and `ELSE` clauses are executed when the expression in the `IF` clause is false, as in other programming languages.

Figure 8.3.5: IF statement.

```
IF expression THEN statement list
[ ELSEIF expression THEN statement list ]
[ ELSE StatementList ]
END IF;
```

Feedback?

The ***WHILE*** statement also includes an expression and a statement list. The statement list repeatedly executes as long as the expression is true. When the expression is false, the `WHILE` statement terminates, and the statement following `END WHILE` executes.

Figure 8.3.6: WHILE statement.

```
WHILE expression DO
   statement list
END WHILE;
```

Feedback?

Other common programming statements are supported within the compound statement, including `CASE`, `REPEAT`, and `RETURN`. Compound statements also support cursors, described below.

PARTICIPATION ACTIVITY 8.3.4: Compound statements in a stored procedure.

1 2 3 4 5 2x speed

```
CREATE PROCEDURE AddFlights(IN startTime TIME, IN endTime TIME)
BEGIN
   DECLARE departTime TIME DEFAULT startTime;
   DECLARE flightNum INT DEFAULT 540;

   WHILE departTime <= endTime DO

      INSERT INTO Flight
      VALUES (flightNum, 'Lufthansa Airlines', departTime, 'FRA');

      SET flightNum = flightNum + 100;

      IF departTime < '12:00' THEN
         SET departTime = ADDTIME(departTime, '00:30');
      ELSE
         SET departTime = ADDTIME(departTime, '1:00');
      END IF;

   END WHILE;
END;
```

The AddFlights() stored procedure inserts new flights into the Flight table. The body is a compound statement.

Captions

1. The AddFlights() stored procedure inserts new flights into the Flight table. The body is a compound statement.
2. The declare statements create and initialize two variables. departTime is assigned the startTime parameter. flightNum is assigned 540.
3. The WHILE loop repeats as long as departTime is ≤ endTime.
4. Each pass through the loop inserts a new flight.
5. flightNum is incremented by 100. If departTime is before noon, add 30 minutes. If departTime is noon or later, add an hour.

Feedback?

PARTICIPATION ACTIVITY 8.3.5: Compound statements.

The following stored procedure awards bonuses based on employee performance ratings:

```
CREATE PROCEDURE AwardBonus()
BEGIN

   DECLARE bonusAmount INT;
   DECLARE ratingCount INT;
   ___A___ rating INT DEFAULT 10;

   ___B___ rating > 0 DO

   SELECT COUNT(*)
   ___C___ ratingCount
   FROM Employee
   WHERE PerformanceRating = rating;

   ___D___ ratingCount > 80
   SET bonusAmount = 1000 * rating;
   ELSE
   SET bonusAmount = 100 * rating;
   END IF;

   UPDATE Employee
   SET EmployeeBonus = bonusAmount
   WHERE PerformanceRating = rating;

   ___E___ rating = rating - 1;

   END WHILE;

END;
```

1) What is keyword A?

declare

Check Show answer

Correct

DECLARE

DECLARE creates variables for a stored procedure.

2) What is keyword B?

while

Check Show answer

Correct

WHILE

WHILE executes a statement list as long as the expression following WHILE is true. WHILE is always paired with END WHILE, which appears later in the stored procedure.

3) What is keyword C?

into

Check Show answer

Correct

INTO

INTO appears between SELECT and FROM. In this statement, INTO assigns RatingCount with the value of COUNT(*).

4) What is keyword D?

if

Check Show answer

Correct

IF

IF always precedes ELSE and END IF.

5) What is keyword E?

set

Check Show answer

Correct

SET

SET sets a variable with an expression. In this statement, SET assigns Rating with Rating - 1.

Feedback?

Cursors

A cursor is a special variable that identifies an individual row of a result table. Cursors are necessary to process queries that return multiple rows. Cursor syntax in procedural SQL is similar to embedded SQL:

- `DECLARE CursorName CURSOR FOR Statement` creates a cursor named CursorName that is associated with the query Statement. The Statement may not have an INTO clause.
- `DECLARE CONTINUE HANDLER FOR NOT FOUND Statement` specifies a Statement that executes when the cursor eventually moves past the last row and cannot fetch any more data. Reading data from a cursor is performed within a loop, so the Statement usually sets a variable indicating the loop should terminate.
- `OPEN CursorName` executes the query associated with CursorName and positions the cursor before the first row of the result table. Multiple cursors with different names can be open at the same time.
- `FETCH FROM CursorName INTO variable [, variable, ... ]` advances CursorName to the next row of the result table and copies selected values into the variables. The number of INTO clause variables must match the number of SELECT clause columns.
- `CLOSE CursorName` releases the result table associated with the cursor.

The animation below shows a stored procedure that uses a cursor to change departure times for all flights.

PARTICIPATION ACTIVITY 8.3.6: Cursor modifies flight departure times.

1 2 3 4 5 6 7 8 2x speed

```
CREATE PROCEDURE ChangeDepartureTime(IN airportIn CHAR(3))
BEGIN

   DECLARE flightNum INT;
   DECLARE airport CHAR(3);
   DECLARE departTime TIME;
   DECLARE finishedReading BOOL DEFAULT FALSE;

   DECLARE flightCursor CURSOR FOR
      SELECT FlightNumber, AirportCode, DepartureTime
      FROM Flight;

   DECLARE CONTINUE HANDLER FOR NOT FOUND SET finishedReading = TRUE;

   OPEN flightCursor;
   FETCH FROM flightCursor INTO flightNum, airport, departTime;

   WHILE NOT finishedReading DO
      IF airport = airportIn THEN
         UPDATE Flight
         SET DepartureTime = ADDTIME(departTime, '1:00')
         WHERE FlightNumber = flightNum;
      ELSE
         UPDATE Flight
         SET DepartureTime = SUBTIME(departTime, '0:30')
         WHERE FlightNumber = flightNum;
      END IF;

      FETCH FROM flightCursor INTO flightNum, airport, departTime;
   END WHILE;

   CLOSE flightCursor;

END;
```

CLOSE releases cursor resources when no longer needed.

Captions

1. The cursor named flightCursor retrieves data for all flights.
2. The finishedReading variable is initially FALSE. The handler sets finishedReading to TRUE when the cursor moves past the last row.
3. OPEN executes flightCursor's associated query and sets the cursor prior to the first result table row.
4. FETCH moves the cursor to the first row and assigns column values to procedure variables.
5. The WHILE loop repeats until the cursor moves past the last row.
6. The IF statement changes the departure time for the selected flight.
7. FETCH moves the cursor to the next row.
8. CLOSE releases cursor resources when no longer needed.

Feedback?

PARTICIPATION ACTIVITY 8.3.7: Cursors.

Refer to code in the above animation.

1) Which name refers to a single value selected by flightCursor?

- airportIn
- airport
- AirportCode

Correct

airport follows the INTO clause of the FETCH statements. The INTO clause assigns a selected value to a variable.

2) What happens to flights that depart from the input airport?

- Departure time is one hour later.
- Departure time is a half-hour earlier.
- Departure time is not changed.

Correct

Flights that depart from the input airport are updated in the first clause of the IF-ELSE statement. ADDTIME() adds one hour to departure time.

3) Where does flightCursor point after the WHILE loop executes three times?

- The third row of the result table
- The fourth row of the result table
- Between the third and fourth row of the result table

Correct

The FETCH prior to WHILE moves the cursor to the first row. Each of the three passes moves the cursor to the next row, so the cursor points to the fourth row after the third pass.

Feedback?

Stored functions

A ***stored function*** is like a stored procedure that returns a single value. A stored function is declared with a ***CREATE FUNCTION*** statement. `CREATE FUNCTION` syntax is similar to `CREATE PROCEDURE` syntax, with several differences:

- Function parameters are for input only and therefore have no `IN`, `OUT`, or `INOUT` keyword.

- Functions return a single value. The value follows the `RETURN` keyword in the stored function body. The value data type follows the `RETURNS` keyword in the stored function heading.
- Functions are invoked within an SQL expression rather than with a `CALL` statement.

Unlike CREATE PROCEDURE statements, CREATE FUNCTION requires at least one of the following keywords:

- `NO SQL` - Function contains no SQL statements.
- `READS SQL DATA` - Function contains SQL statements that read, but not write, table data.
- `DETERMINISTIC` - Function contains SQL statements that write table data.

Figure 8.3.7: CREATE FUNCTION statement.

```
CREATE FUNCTION FunctionName( <parameter-declaration>, <parameter-declaration>,
... )
   RETURNS Type
   [ NO SQL | READS SQL DATA | DETERMINISTIC ]
Body;

<paramenter-declaration>:
ParameterName Type
```

Feedback?

Stored functions behave like built-in SQL functions, such as COUNT(), SUM(), and AVG(). The only difference is that stored functions are defined by the programmer, while built-in functions are defined by the database.

Stored functions are limited compared to stored procedures. Stored functions return only one value, while stored procedures can return multiple values through different `OUT` parameters. However, stored functions, like built-in functions, are useful when a computed value is needed within an SQL expression.

PARTICIPATION ACTIVITY 8.3.8: Stored function that computes tax.

1 2 3 4 5 2x speed

```
CREATE FUNCTION ComputeTax(employeeName VARCHAR(20))
   RETURNS INT
   READS SQL DATA
BEGIN
```

```
   DECLARE income INT;

   SELECT Salary
   INTO income
   FROM Employee
   WHERE Name = employeeName;

   IF income <= 25000 THEN
      RETURN 0;
   ELSE
      RETURN (income - 25000) * 0.1;
   END IF;

END;
```

```
mysql> SELECT ComputeTax('Lisa Ellison');
+-----------------------------+
| ComputeTax('Lisa Ellison') |
+-----------------------------+
|                        1500 |
+-----------------------------+
1 row in set (0.00 sec)
```

Lisa Ellison's salary is $40,000. Tax is ($40,000 - $25,000) * 10% = $1,500, displayed as a result table.

Captions

1. The stored function computes an employee's tax and returns an integer. The function reads SQL data with a SELECT statement.
2. The employee's salary is retrieved and assigned to the income variable.
3. Tax is computed as 10% of income exceeding $25,000 and returned.
4. Stored functions are called within any SQL expression, like built-in functions.
5. Lisa Ellison's salary is $40,000. Tax is ($40,000 - $25,000) * 10% = $1,500, displayed as a result table.

Feedback?

PARTICIPATION ACTIVITY 8.3.9: Stored functions.

1) Stored functions have no parameters.

- True
- False (selected)

Correct

Stored functions may have input parameters, but not output parameters.

2) RETURN and RETURNS are equivalent keywords.

- True
- (selected)

Correct

RETURN assigns a return value to the stored function and appears within the body. RETURNS specifies a data

False

type for the return value and appears between the CREATE FUNCTION clause and the body.

3) The function below requires no additional keywords to execute.

```
CREATE FUNCTION
Smallest(x INT, y INT)
   RETURNS INT
BEGIN
   IF x < y THEN
      RETURN x;
   ELSE
      RETURN y;
   END IF;
END;
```

True

False

Correct

All CREATE FUNCTION statements require the keywords NO SQL, READS SQL DATA, or DETERMINISTIC. Smallest() does not contain SQL statements, so NO SQL should be specified after "RETURNS INT".

4) Stored functions may be executed from the MySQL command line.

True

False

Correct

A stored function may appear in an expression within an SQL statement. When SQL statements are executed from the command line, embedded stored functions are also executed.

5) The <paramenter-declaration> has the same syntax in stored procedures and stored functions.

True

False

Correct

The <parameter-declaration> may contain IN, OUT, and INOUT parameters in stored procedures, but not in stored functions.

Feedback?

Triggers

A ***trigger*** is like a stored procedure or a stored function, with two differences:

- Triggers have neither parameters nor a return value. Triggers read and write tables but do not communicate directly with a calling program.
- Triggers are not explicitly invoked by a `CALL` statement or within an expression. Instead triggers are associated with a specific table and execute whenever the table is changed.

Triggers are used to enforce business rules automatically as data is updated, inserted, and deleted.

The ***CREATE TRIGGER*** statement specifies a TriggerName followed by four required keywords:

- `ON TableName` identifies the table associated with the trigger.
- `INSERT`, `UPDATE`, or `DELETE` indicates that the trigger executes when the corresponding SQL operation is applied to the table.
- `BEFORE` or `AFTER` determines whether the trigger executes before or after the insert, update, or delete operation.
- `FOR EACH ROW` indicates the trigger executes repeatedly, once for each row affected by the insert, update, or delete operations.

MySQL triggers do not support all features of the SQL standard. Ex: The standard includes keywords `FOR EACH STATEMENT` as an alternative to `FOR EACH ROW`. `FOR EACH STATEMENT` indicates the trigger executes once only, rather than repeatedly for each affected row.

Like stored procedures, the trigger body may be a simple or compound statement. Within the body, keywords `OLD` and `NEW` represent the name of the table associated with the trigger. `OLD` represents the table values prior to an update or delete operation. `NEW` represents the table values after an insert or update operation.

Figure 8.3.8: CREATE TRIGGER statement.

```
CREATE TRIGGER TriggerName
[ BEFORE | AFTER ]  [ INSERT | UPDATE | DELETE ]
ON TableName
FOR EACH ROW
Body;
```

Feedback?

PARTICIPATION ACTIVITY 8.3.10: Trigger example.

1 2 3 4 5 2x speed

```
CREATE TRIGGER ExamGrade
BEFORE INSERT
ON Exam
FOR EACH ROW
BEGIN
```

```
mysql> INSERT INTO Exam (ID, Score)
    -> VALUES (1, 79), (2, 92), (3, 85)
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
    IF NEW.Score >= 90 THEN
       SET NEW.Grade = 'A';
    ELSEIF NEW.Score >= 80 THEN
       SET NEW.Grade = 'B';
    ELSEIF NEW.Score >= 70 THEN
       SET NEW.Grade = 'C';
    ELSE
       SET NEW.Grade = 'F';
    END IF;

END;
```

```
mysql> SELECT * FROM Exam;
+----+-------+-------+
| ID | Score | Grade |
+----+-------+-------+
|  1 |    79 | C     |
|  2 |    92 | A     |
|  3 |    85 | B     |
+----+-------+-------+
```

The Grade column contains values set by the ExamGrade trigger.

Captions

1. The ExamGrade trigger executes before inserts to the Exam table.
2. One insert statement can create multiple rows. The trigger executes once for each new row.
3. The trigger automatically determines exam grade and updates the Grade column in new table rows.
4. Inserting new rows causes the ExamGrade trigger to execute before the rows are inserted.
5. The Grade column contains values set by the ExamGrade trigger.

Feedback?

PARTICIPATION ACTIVITY 8.3.11: Triggers.

Refer to the trigger in the animation above.

1) What happens when an exam score is updated from 65 to 83?

   - Grade changes from C to B.
   - (selected) Grade does not change.
   - Grade is set to NULL.

Correct

Grade does not change because an UPDATE does not activate the trigger.

2) NULLs are allowed in Score. What happens when a new row is

Correct

The trigger executes. No expressions in the IF and ELSEIF clauses are true, so the ELSE clause executes, assigning a grade of 'F'.

inserted with a NULL Score?

- The insert fails and no row is created.
- The insert succeeds and Grade is set to the column default value.
- The insert succeeds and ExamGrade is set to 'F'

3) An INSERT is attempted, but the inserted primary key is not unique. Does the trigger execute?

- Yes, the trigger executes, and the INSERT succeeds.
- Yes, the trigger executes, but the INSERT fails.
- No, the trigger never executes.

Correct

The trigger executes successfully before the INSERT is attempted. The INSERT fails because the INSERT violates primary key uniqueness.

Feedback?

CHALLENGE ACTIVITY 8.3.1: Procedural SQL.

379958.2369558.qx3zqy7

Jump to level 1

1

2

The following procedure uses a cursor to add a seat with number 0 to every room that seats. Fill in the missing keywords.

```
CREATE PROCEDURE MarkEmpty()
BEGIN
```

```
DECLARE nam VARCHAR(40);
DECLARE num INT;
DECLARE finished BOOL DEFAULT FALSE;

DECLARE roomCursor CURSOR ___(A)___
   SELECT Name FROM Room;
DECLARE ___(B)___ HANDLER FOR NOT FOUND SET finished = TRUE;
OPEN roomCursor;
FETCH ___(C)___ roomCursor ___(D)___ nam;

WHILE NOT finished DO
   SELECT COUNT(*) INTO num FROM Seat WHERE RoomName = nam;
   IF num = 0 ___(E)___
      INSERT INTO Seat VALUES (nam, 0);
   END IF;
   FETCH FROM roomCursor INTO nam;
END WHILE;
___(F)___ roomCursor;
END;
```

(A)	for	(D)	into
(B)	continue	(E)	then
(C)	from	(F)	close

1 2

Check Next **Done**. Click any level to practice more. Completion is preserved.

Expected:
(A) `FOR`
(B) `CONTINUE`
(C) `FROM`
(D) `INTO`
(E) `THEN`
(F) `CLOSE`

(A) `FOR` follows `CURSOR` when declaring a cursor.
(B) `CONTINUE` follows `DECLARE` when declaring a handler.
(C) `FETCH FROM roomCursor` retrieves the next result from `roomCursor`.
(D) `INTO` follows the cursor name to copy selected values into `nam`.
(E) `THEN` follows the logical expression `num = 0` of the `IF` statement.
(F) `CLOSE` closes the cursor before the procedure exits.

Feedback?

Exploring further:

- MySQL CREATE PROCEDURE and CREATE FUNCTION statements
- MySQL CALL statement
- MYSQL compound statements and cursors
- MySQL CREATE TRIGGER statement

How was this section? Provide feedback