PROCEDURAL SQL

Topics

- Database Subroutines
- Stored Procedures
- Stored Functions
- Cursors

Database Subroutines

- Many relational databases allow you to define subroutines in the database
- Approaches:
  - Store subroutine written in a traditional language (Java, C#)
  - Store subroutine written in SQL

Two Faces of SQL

- SQL originally a set-oriented declarative query language
- Initially lacked subroutines and control (if/while) statements
- DBMS vendors added proprietary extensions to SQL to provide procedural functionality

Common Procedural Extensions

- **control statements**
  - WHILE statement
  - IF statement
  - CASE statement
- **stored procedures**
  - User-defined subroutines stored in the database
- **triggers**
  - execute SQL commands when data is modified in the database

SQL/PSM

- SQL standard includes procedural extensions called SQL/PSM
  - Implemented by MySQL, Postgres, DB/2, SQL Server, Oracle
  - Vendor conformance to standard remains an issue
  - This presentation: MySQL

Stored Procedures

- Define stored procedure
  - CREATE PROCEDURE MakeNewStudent()
    BEGIN
      INSERT INTO student VALUES(1, 'Frank');
    END
- Call stored procedure
  - CALL MakeNewStudent()

Define Stored Procedure in MySQL Editor

- In standard MySQL editor, must temporarily change end of statement delimiter using the **delimiter** directive to define a stored procedure:

```
delimiter $$
CREATE PROCEDURE MakeNewStudent()
BEGIN
  INSERT INTO student VALUES(1, 'Frank');
END$$
```

  - When using stored procedure editor, MySQL adds the delimiter commands for you
- After creating the stored procedure, to make changes to it, must drop and re-create stored procedure
  - DROP PROCEDURE MakeNewStudent;

Stored Procedure Parameters

- A stored procedure can receive parameters from the caller
- Define stored procedure with parameters

```
CREATE PROCEDURE MakeNewStudent(stu_id INTEGER, stu_name VARCHAR(40))
BEGIN
  INSERT INTO student VALUES(stu_id, stu_name);
END
```

- Call stored procedure
  - CALL MakeNewStudent(5, 'Frank')

Storing Queries

- A stored procedure can contain a query
- Example:

```
CREATE PROCEDURE GetProductsToReorder(reorder_quantity INTEGER)
BEGIN
    SELECT * FROM products
    WHERE  products.quantity < reorder_quantity;
END
```

- Execute query:
  - CALL GetProductsToReorder(5);

Stored Queries, continued

- We now have two ways to store a query in the database
- Stored Procedure
  - allows query to take parameters
  - can access tables the user does not have permission to access
- Views
  - allows query to be reused in defining other queries
  - more easily ported to other DBMS

Stored Procedure Parameters

- Stored procedures can use flow control statements like IF, WHILE, and CASE

```
CREATE PROCEDURE MakeNewStudent(stu_id INTEGER, stu_name VARCHAR(40))
BEGIN
  IF stu_name <> 'Test' THEN
    INSERT INTO student VALUES(stu_id, stu_name);
  END IF;
END
```

IF Statements

- Full syntax:

```
IF condition THEN
    ... statements ...
ELSEIF condition THEN
    ... statements ...
ELSE
    ... statements ...
END IF;
```

- *condition* can be any Boolean expression that could appear in a WHERE clause of a SELECT statement

Variables

- Create variables using DECLARE

```
CREATE PROCEDURE MakeNewStudents(num_stu INTEGER)
BEGIN
  DECLARE i INTEGER DEFAULT 0;   -- define i as integer with initial value 0

  WHILE i < num_stu DO
    INSERT INTO student VALUES(i, 'Test Student');
    SET i = i + 1;  -- increment i
  END WHILE;
END
```

- Alter variable values using SET

Creating Variables

- DECLARE statement
  - Use SQL type for variable data type
  - Initial variable value is NULL unless you specify a DEFAULT
- Examples:
  - DECLARE balance NUMERIC(8,2) DEFAULT 0.0;
  - DECLARE name VARCHAR(40) DEFAULT ‘’;

Retrieving Data Into Variables

- Use the INTO clause of SELECT

```
DECLARE fname, lname VARCHAR(20);

SELECT FirstName, LastName
INTO fname, lname
FROM Employee
WHERE EmployeeID = 1;
```

Employee

EmployeeID	FirstName	LastName
1	Fred	Jones
2	Amy	Corey

- Query should be designed to produce <u>at most 1 row</u>
  - If no rows produced, fname and lname are not modified
  - If 1 row produced, fname and lname receive the values of FirstName and LastName, respectively
  - If multiple rows produced, error occurs

Example 1: Enhanced MakeNewStudents


```
CREATE PROCEDURE MakeNewStudents(num_stu INTEGER)
BEGIN
  DECLARE i INTEGER DEFAULT 0;
  DECLARE next_id INTEGER;

  WHILE i < num_stu DO
     SELECT MAX(StudentID) + 1
     INTO next_id
     FROM student;
    INSERT INTO student VALUES(next_id, 'Test Student');
    SET i = i + 1;

  END WHILE;
END
```

StudentID	StudentName
1	Fred
2	Alice

Example 2: Debit Account Balance


```
CREATE PROCEDURE withdraw(from_acct_id INT, amt NUMERIC(18,2))
BEGIN
  DECLARE cur_amt NUMERIC(18,2);
  -- Retrieve current account balance from account with account_id = from_acct_id
  SELECT  _____;
  -- Does account exist?
  IF _____ THEN
    -- Sufficient funds?
    IF _____ THEN
      UPDATE account SET balance = cur_amt - amt WHERE account_id = from_acct_id;
    END IF;
  END IF;
END
```

account

account_id	balance
100	5000.00
101	3000.00

Example 2: Debit Account Balance (solution)


```
CREATE PROCEDURE withdraw(from_acct_id INT, amt NUMERIC(18,2))
BEGIN
  DECLARE cur_amt NUMERIC(18,2);
  SELECT account.balance INTO cur_amt
  FROM account
  WHERE account.account_id = from_acct_id;
  IF cur_amt IS NOT NULL THEN
    IF cur_amt - amt >= 0 THEN
      UPDATE account SET balance = cur_amt - amt
      WHERE account_id = from_acct_id;
    END IF;
  END IF;
END
```

Example 2: Debit Account Balance (alt. solution)

- Can we rewrite using a single update statement?

```
CREATE PROCEDURE withdraw(from_acct_id INT, amt NUMERIC(18,2))
BEGIN
  UPDATE account
  SET balance = cur_amt - amt
  WHERE account_id = from_acct_id AND balance – amt >= 0;
END
```

Returning Values

- Define an OUT parameter to return a result from a stored procedure

```
CREATE PROCEDURE AddNumbers(num1 INTEGER, num2 INTEGER,
  OUT sum INTEGER)
BEGIN
  SET sum = num1 + num2;
END
```

- In MySQL Query Editor, call stored procedure using an @variable
  - CALL AddNumbers(2, 3, @answer);
- Display result in query editor
  - SELECT @answer;

Example: Returning a success code


```
CREATE PROCEDURE withdraw(from_acct_id INT, amt NUMERIC(18,2), OUT result INT)
BEGIN
  DECLARE curamt NUMERIC(18,2);
  SELECT balance INTO curamt FROM account WHERE account_id = from_acct_id;
  IF curamt IS NULL THEN
    -- account doesn't exist
    SET result = -1;
  ELSEIF curamt - amt < 0 THEN
    -- insufficient funds
    SET result = -2;
  ELSE
      UPDATE account SET balance = curamt - amt WHERE account_id = from_acct_id;
      SET result = 0;
  END IF
END
```

In-Class Practice

- Write TransferFunds:
  - CREATE PROCEDURE TransferFunds(from_acct_id INTEGER, to_acct_id INTEGER, xfer_amt NUMERIC(8, 2), OUT success INTEGER)
  - If account from_acct_id has balance >= xfer_amt:
    - decrement balance of from_acct_id by xfer_amt
    - increment balance of to_acct_id by xfer_amt
    - set success = 1
  - Otherwise:
    - set success = 0

Summary: Returning Results

A stored procedure can return results in two ways:

1. Return a result set using an embedded SELECT statement (with no INTO clause)
   - Useful for storing queries in database
2. Return individual values through OUT parameters
   - Useful for returning success codes

Calling Stored Procedures from Python

Stored Procedure

```
CREATE PROCEDURE AddNums(
   num1 INTEGER, num2 INTEGER,
   OUT sum INTEGER)
BEGIN
  SET sum = num1 + num2;
END
```

Call from Python

```
result = cursor.callproc('AddNums', (5, 3, 0))
print("Got stored procedure result: " + str(result[2]))
```

Stored Functions

Stored Functions

- Define a stored function

```
CREATE FUNCTION Area(length INTEGER, width INTEGER)
RETURNS INTEGER -- return type
DETERMINISTIC
BEGIN
  RETURN ( length * width ) ;
END
```

- Invoke stored function in a SELECT query
  - Example 1:

```
SELECT Area(2, 3);
```

  - Example 2:

```
SELECT id, prod_length, prod_width, Area(prod_length, prod_width)
FROM product;
```

product

id	prod_length	prod_width
1	3	5
2	10	20

About Stored Functions

- Stored functions compute a single value for use in a SELECT
- Useful for removing complex calculations from SELECT
- Consider this query:

```
SELECT ProdID, ProdName,
 ( CASE
    WHEN Quantity < 3 THEN 'Reorder today'
    WHEN Quantity < 10 THEN 'Reorder soon'
    ELSE 'No reorder'
    END ) AS ReorderStatus
FROM products;
```

Simplify Query with a Stored Function

- Define function to calculate reorder status

```
CREATE FUNCTION GetReorderStatus(qty INTEGER)
RETURNS VARCHAR(20)
DETERMINISTIC
BEGIN
  IF qty < 3 THEN
    RETURN 'Reorder today'
  ELSEIF qty < 10 THEN
    RETURN 'Reorder soon'
  ELSE
    RETURN 'No reorder'
  END IF;
END
```

- Rewrite query to use function

```
SELECT ProdID,
       ProdName,
       GetReorderStatus(Quantity)
FROM products;
```

Stored Procedures vs. Stored Functions

Stored Procedures

- Execute using CALL statement
- Used to store queries or perform modifications to database
- No return type or RETURN statement

Stored Functions

- Execute using SELECT statement
- Compute a value for use in a SELECT or WHERE clause
- Specify a return type and include a RETURN statement

Cursors

Processing Result Sets

- We've seen how a stored procedure can use SELECT ... INTO to process data retrieved from a SELECT statement
  - Limitation: SELECT statement must produce only one row
- What about processing data from SELECT statements that retrieve multiple rows?
  - A cursor can help!

Cursor

- A cursor is a procedural SQL mechanism that allows stored procedures to process data in SELECT result sets that contain more than one row
- Using a cursor involves several steps:
  - Define a cursor variable
  - Define a continue handler for the cursor
  - Open the cursor
  - Write a loop to fetch data from the cursor
  - Close the cursor

Cursor Example

Step 1: Declare cursor and related variables

```
-- Declare variables we'll need
DECLARE cur_id INT;
DECLARE cur_balance NUMERIC(18,2);
DECLARE finished INT DEFAULT 0; -- flag for end of record

-- Declare cursor
DECLARE acct_cursor CURSOR FOR
   SELECT id, balance
   FROM account;

-- Define a handler that executes when cursor moves past end
DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET finished = 1;
```

Step 2: Open the cursor (Executes query)

```
OPEN acct_cursor;
```

Step 3: Process each row

```
-- Fetch first row
FETCH acct_cursor INTO cur_id, cur_balance;
-- Loop through results
WHILE finished = 0 DO
  -- Do something with data retrieved from cursor
  IF cur_balance < 1000 THEN
    -- Update current record
    UPDATE account
      SET balance = cur_balance - 100
      WHERE id = cur_id;
  END IF;

  -- Fetch next row
  FETCH acct_cursor INTO cur_id, cur_balance;
END WHILE
```

Step 4: Close the cursor

```
CLOSE acct_cursor;
```

Cursor Processing

- DECLARE defines cursor variable and associates it with a SELECT statement
  - DECLARE acct_cursor CURSOR FOR SELECT id, balance FROM account;
- OPEN executes the query and positions cursor at first row of the query's result set
  - OPEN acct_cursor;
- FETCH retrieves data from the current row of the result set and moves cursor to next row
  - FETCH acct_cursor INTO cur_id, cur_balance;

End of Result Set

- FETCH must be used in a loop to retrieve all of the rows from the query's result set
- Eventually, FETCH will fail to retrieve another row because all of the result set rows have been retrieved
- When this happens, the continue handler is executed
  - DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET finished = 1;
- The continue handler must set a variable that can be checked in the loop to cause the loop to exit

Cursors vs. Set-Based Approaches

- With some thought, a procedural cursor-based solution can often be rewritten using a set-based approach
- Challenge: Rewrite the cursor example using a single UPDATE statement

Cursors vs. Set-Based Approaches, cont.

- Given the cursor below, rewrite using a set-based solution without a cursor

```
DECLARE cur_id INT;
DECLARE cur_balance, deduct_amt
            NUMERIC(18,2);

DECLARE acct_cursor CURSOR FOR
   SELECT id, balance
   FROM account
   WHERE balance < 2000;
…
OPEN acct_cursor;
FETCH acct_cursor INTO cur_id, cur_balance;
```

```
WHILE finished = 0 DO
  IF cur_balance < 1000 THEN
    SET deduct_amt = 150;
  ELSE
    SET deduct_amt = 100
  END IF;
  UPDATE account
      SET balance = cur_balance – deduct_amt
      WHERE id = cur_id;
  FETCH acct_cursor INTO cur_id, cur_balance;
END WHILE;
```

Cursor Pros and Cons

Advantages

- Gives programmer a way to process multiple rows from a select statement

Disadvantages

- Performance is not as good as set-based statements
- Often abused by developers who aren't comfortable with solving problems using UPDATE/INSERT/DELETE approaches.
- Use cursors only when set-based statements aren’t powerful enough

Additional Reading

- http://www.mysqltutorial.org/mysql-stored-procedure-tutorial.aspx
  Helpful stored procedure tutorial
- https://dev.mysql.com/doc/refman/8.0/en/sql-syntax-compound-statements.html
  MySQL Stored Procedure Reference