Stored Procedures, Macros and Recursive Queries

Macros

- A **macro** is a set of SQL statements which are stored and executed by calling the macro name.
- The definition of macros is stored in Data Dictionary. Users only need `EXEC` privilege to execute the macro.
- Users do not need separate privileges on the database objects used inside the macro.

Macros

- Macro statements are executed as a single transaction:
  - If one of the SQL statements fails, then all the statements are rolled back.
  - Macros can accept parameters.
  - Macros can contain DDL statements, but that should be the last statement.

Create Macros

```
CREATE MACRO <macroname> [(parameter1, parameter2,...)]
(
<sql statements>
);
```

Example

```
CREATE MACRO Get_Emp AS
(
SELECT
EmployeeNo,
FirstName,
LastName
FROM
employee
ORDER BY EmployeeNo;
);
```

```
EXEC Get_Emp;
```

Parameterized Macros

Macro parameters are referenced with `:Param;` .

```
CREATE MACRO Get_Emp_Salary(EmployeeNo INTEGER) AS
(
SELECT
EmployeeNo,
NetPay
FROM
Salary
WHERE EmployeeNo = :EmployeeNo;
);
```

```
EXEC Get_Emp_Salary(101);
```

Stored Procedures

Stored Procedures

- A stored procedure contains a set of SQL statements and procedural statements.
- The definition of stored procedure is stored in database and the parameters are stored in data dictionary tables.

Stored Procedures (cont.)

Advantages

- Stored procedures reduce the network load between the client and the server.
- Provides better security since the data is accessed through stored procedures instead of accessing them directly.
- Gives better maintenance since the business logic is tested and stored in the server.

Example

```
CREATE PROCEDURE <pname> ( [par1 dtype1, par2 dtype2] )
BEGIN
<SQL or SPL statements>;
END;
```

Example (cont.)

```
CREATE PROCEDURE InsertSalary(
 IN in_EmployeeNo INTEGER, IN in_Gross INTEGER,
 IN in_Deduction INTEGER, IN in_NetPay INTEGER
)
BEGIN

INSERT INTO Salary
( EmployeeNo, Gross, Deduction, NetPay )
VALUES
(:in_EmployeeNo, :in_Gross, :in_Deduction, :in_NetPay);

END;
```

```
CALL InsertSalary(105,20000,2000,18000);
```

- The `BEGIN` and `END` statements are required in all Stored Procedures.

Multiple `BEGIN/END` statements

```
CREATE PROCEDURE Second_Procedure()
BEGIN
INSERT INTO Customer_Table DEFAULT VALUES;
SecondSection:BEGIN
DELETE FROM Customer_Table
WHERE Customer_Number is NULL;
END SecondSection;
END;
```

- When you have multiple BEGIN and END statements, you have to label them all (except for the first BEGIN and END statements).

Declare a Variable

```
CREATE PROCEDURE Declare_Procedure( )
BEGIN
DECLARE var1 INTEGER DEFAULT 111;
DELETE FROM Customer_Table
WHERE Customer_Number = :var1;
END;
```

- When you DECLARE a variable and then reference that variable later, a colon is always in front of the variable.

Declare a Variable and then SET the Variable

```
CREATE PROCEDURE SetVar_Procedure( )
BEGIN
DECLARE var1 INTEGER ;
SET var1 = 313 ;
DELETE FROM Customer_Table
WHERE Customer_Number = :var1;
END;
```

- Once a variable and the data type is defined the value must be assigned.

An IN Variable is passed to the Procedure during the CALL

```
CREATE PROCEDURE PassInput_Procedure(IN var1 INTEGER )
BEGIN
DELETE FROM
Customer_Table WHERE Customer_Number = :var1;
END;
```

```
CALL PassInput_Procedure (123 ) ;
```

- The Variable Var1 was not assigned with SET, but instead passed as a parameter.
- There are three types of parameters (IN, OUT, INOUT). In this example, an IN is being used.

`IN` and `OUT` Parameters

```
CREATE PROCEDURE Test_Proc
(IN var1 BYTEINT, IN var2 BYTEINT, OUT Msg CHAR(20) )
BEGIN
CASE WHEN var1 = var2 THEN Set Msg = 'They are equal' ;
WHEN var1 < var2 THEN Set Msg = 'Variable 1 less' ;
ELSE Set Msg = 'Variable 1 greater' ;
END CASE;
END;
```

```
CALL Test_Proc (1,2, Msg ) ;
```

- This Stored Procedure will take a parameter in and then send something out.

IF inside a Stored Procedure

```
CREATE PROCEDURE TestIF_Proc
(IN var1 BYTEINT, IN var2 BYTEINT, OUT Msg CHAR(20) )
BEGIN
IF var1 = var2 THEN SET Msg = 'They are equal';
END IF;
IF var1 < var2 THEN SET Msg = 'Variable 1 less';
END IF;
IF var1 > var2 THEN SET Msg = 'Variable 1 greater';
END IF;
END;
```

```
CALL TestIF_Proc (2,2, Msg ) ;
```

Loops

```
CREATE Table My_Log_Tbl
(
Cntr Integer
,TheTime Time
) Primary Index (Cntr);

CREATE PROCEDURE Inserter_Five( )
LOOPER:BEGIN
DECLARE Cntr INTEGER DEFAULT 0;
Loopit:LOOP
SET Cntr = Cntr + 1;
IF Cntr > 5 THEN LEAVE Loopit;
END IF;
INSERT INTO My_Log_Tbl
VALUES (:Cntr);
END LOOP Loopit ;
END LOOPER;
```

- `LOOP`s require Labeling, like when you have more than one `BEGIN/END`.

LEAVE

```
CREATE PROCEDURE Ins5( )
LOOPER:BEGIN
DECLARE Cntr INTEGER
DEFAULT 0;
Loopit:LOOP
SET Cntr = Cntr + 1;
IF Cntr > 5 THEN LEAVE Loopit;
END IF;
INSERT INTO My_Log_Tbl
VALUES (:Cntr);
END LOOP Loopit ;
END LOOPER;
```

UNTIL

```
CREATE PROCEDURE Ins5A( )
LOOPER:BEGIN
DECLARE Cntr INTEGER
DEFAULT 0;
Loopit:REPEAT
SET Cntr = Cntr + 1;
INSERT INTO My_Log_Tbl
VALUES (:Cntr);
UNTIL Cntr > 4
END REPEAT Loopit ;
END LOOPER;
```

- **Trivia:** Differences between these two?

Passing result sets

- Result sets with one row can be easily assigned into variables

```
SELECT COUNT(*) INTO variable_name  FROM table_name;
```

- How about something like:

```
SELECT column_name INTO variable_name  FROM table_name;
```

Cursors

```
DECLARE cname [SCROLL|NOSCROLL] CURSOR FOR sql_select;
```

```
OPEN cursor_name [USING parameter,…];
```

```
FETCH [NEXT|FIRST] FROM cursor_name INTO [var|param];
```

```
CLOSE cursor_name;
```

- Two important SQLSTATES:

No rows to fetch: ‘02000’

Cursor is closed: ‘24501’

Example

```
BEGIN
DECLARE c, n INTEGER DEFAULT 0;
DECLARE cname CURSOR FOR SELECT PK FROM table_int_pk;
OPEN cname;
label1:
LOOP
FETCH cursor_name INTO c;
IF (SQLSTATE = '02000') THEN
  LEAVE label1;
END IF;
SET n = n + c;
END LOOP label1;
CLOSE cname;
SET m = n;
END;
```

- Without checking for SQLSTATE '02000', we get an infinite loop.

Dynamic SQL

```
REPLACE PROCEDURE CURSOR_SAMPLE (IN  m INTEGER)
BEGIN
DECLARE n INTEGER DEFAULT 0;
DECLARE my_sql VARCHAR(1000);
DECLARE cursor_name NO SCROLL CURSOR  FOR  my_statement;
SET my_sql='SELECT PK FROM table_name  WHERE PK = ? ;';
PREPARE my_statement from my_sql;
OPEN cursor_name USING m;
END;
```

- `PREPARE` statements should be deallocated

  `DEALLOCATE PREPARE statement_name;`

FOR LOOP CURSOR

```
FOR loop_variable AS [cursor_name CURSOR FOR]
cursor_sql DO statement
END FOR;
```

Macros vs Procedures

Differences between macros and procedures

- The macro contains only SQL and maybe dot commands that are only for use in BTEQ.
- A marco is normally a `SELECT` statement with rows returned to the user.
- A stored procedure does not return rows to the user like a macro. Instead, the selected column or columns must be used within the procedure.

Differences between macros and procedures (cont.)

- Like a macro, stored procedures allow parameter values to be passed to it at execution time.
- Unlike a macro that allows only input values, a stored procedure also provides output capabilities.
- A stored procedure only returns output values to a user client as output parameters, not as rows.
- Stored procedures are harder to debug, since there is no live connection to the database.

Exercise

- Write a macro / stored procedure that, given a user as an input returns the next best movie to recommend.
- The reccommended movie should be on the list of movies that the most similar user to him has watched, if the similarity score is above 0.5. If not, reccommend one of the top 10 rated movies that are not on his top 5.

Recursive Queries

Motivation: Holidays

```
CREATE TABLE flights (
origin char(3) not null,
destination char(3) not null, cost int);

INSERT INTO flights VALUES ('PRG', 'WRO', 300);
INSERT INTO flights VALUES ('PRG', 'SOF', 100);
INSERT INTO flights VALUES ('SOF', 'WAW', 275);
INSERT INTO flights VALUES ('WAW', 'WRO', 180);
INSERT INTO flights VALUES ('PRG', 'CDG', 250);
INSERT INTO flights VALUES ('CDG', 'WRO', 140);
```

Flights at one stop from an airport

```
/*Create a table containing
all flights originating at PRG with one stop*/

create table flights_1stop_prg
(origin, destination, cost)
as
(
select a.origin, b.destination, a.cost + b.cost
from flights a inner join flights b
on a.destination = b.origin
and a.origin = 'PRG'
)
with data;
```

Two stops

```
/*List all flights with two stops originating at PRG*/
select b.origin, a.destination, a.cost + b.cost
from flights a
inner join flights_1stop_prg b
on b.destination = a.origin;
```

Alternative: Recursive queries

```
WITH RECURSIVE All_Trips
(Origin,
Destination,
Cost,
Depth) AS
(
SELECT Origin, Destination, Cost, 0
FROM Flights
WHERE origin = 'PRG'
UNION ALL
SELECT All_Trips.Origin,
       Flights.Destination,
       All_Trips.Cost + Flights.Cost,
       All_Trips.Depth + 1
FROM All_Trips INNER JOIN Flights
ON All_Trips.Destination = Flights.Origin
AND All_Trips.Origin = 'PRG'
WHERE Depth < 2 )
SELECT * FROM All_Trips ORDER BY Depth;
```

General syntax

```
WITH RECURSIVE [recursive_table] (
(
[column_list]
) AS
(
[seed statement]
UNION ALL
[recursive statement]
)
SELECT * FROM [recursive_table];
```

Exercise

- Write a recursive query that returns, for a given employee, the list of all its indirect subordinates.

emp_id	emp_name	mgr_id
1	Tom	3
2	Jim	1
3	Will	0
4	Marius	1
5	Lucy	2
6	Julia	3

Solution

```
WITH RECURSIVE emp_hier (emp_id, mgr_id, level) AS
(
SELECT a.emp_id, a.mgr_id, 0
FROM   employee a
WHERE  a.emp_id = <id>
UNION ALL
SELECT b.emp_id, b.mgr_id, c.level+1
FROM   employee b,
       emp_hier c
WHERE  b.mgr_id = c.emp_id
)
SELECT e.emp_id, e.mgr_id, h.level
FROM   employee e,
       emp_hier h
WHERE  e.emp_id = h.emp_id
  AND  e.emp_id <> <id>;
```