Stored Procedures

Stored Procedures

- A stored procedure contains a set of SQL statements *and* procedural statements.
- Teradata procedures can be of two types:
  - **General Procedure:** Procedure that performs some action in the backgorund.
  - **Dynamic procedure:** Procedures that returns resultset/query result.

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.
- Good maintenance since the business logic is tested and stored in the database.

Example

```
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.

`SET`, `IN` and `OUT` Parameters

```
CREATE PROCEDURE CompareValues
(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.

General Syntax

```
REPLACE PROCEDURE [database_name.procedurename]
[(Input_variable [datatype],Output_variable [datatype])]
[DYNAMIC RESULT SETS 1]
BEGIN
DECLARE var1 [datatype];
DECLARE var2 [datatype];
--Error handling block
DECLARE EXIT HANDLER FOR SQLEXCEPTION
BEGIN
 ROLLBACK;
 -- error handling code
END;

DECLARE CONTINUE HANDLER FOR SQLSTATE '42000'
BEGIN
-- error handling code
END;

--Procedure Logic --
END;
```

Remarks

- `REPLACE` command will create if not present.
- Table does not exist: `SQLSTATE` 42000

`IF` inside a Stored Procedure

```
CREATE PROCEDURE CompareValuesWithIf
(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
,eventtime Time
) Primary Index (cntr);

CREATE PROCEDURE InsertFiveRecords( )
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.
- `LEAVE` , `UNTIL`

Multiple `BEGIN/END` statements

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

Cursors

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

- `SCROLL` goes to the beginning of results set.

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

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

```
CLOSE cname;
```

- Without checking for SQLSTATE '02000' (no rows to fetch), we get an infinite loop.

Several examples

- Time to look at the scripts that start with `07` in the `scripts` folder.

Useful links

- Teradata documentation.
- Fancy stored procedure from Teradata

Triggers

Why do we need them?

- *Event-driven* operations.
- Useful for data integrity checks and auto updates.

Types

- **Row-level:** Execute once per row modified by the event.
- **Statement-level:** Only once per event.
- `BEFORE` and `AFTER` the triggering event is completed.

What can be an event?

- `INSERT`
- `UPDATE`
- `DELETE`
- `INSERT-SELECT`

What can be a response?

- `INSERT`
- `UPDATE`
- `DELETE`
- `INSERT SELECT`
- `ABORT/ROLLBACK`
- `EXEC (macro)`

Example

Check `08 Triggers.sql`

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;
```

Wait, a loop?

- Wasn't this the point of stored procedures?
- **Yes**. But having stored procedures doing queries *beats the purpose of parallelization*.
- Teradata is optimized for working in parallel. While it is a bit trickier to think in terms of result sets than in terms of procedures, it is worth it.

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	Mariusz	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>;
```