Advanced Teradata

In this lecture

- Useful system tables: data dictionary.
- Macros.
- Stored Procedures.
- Recursive Queries.

Data dictionary

Teradata Data Dictionary

- Teradata data dictionary tables are metadata tables present in the DBC database.
- It can be used for variety of things such as checking table size, query bottleneck and database size etc.

Useful tables

- **dbc.tables:** Objects present in a database and their related information
- **dbc.columns:** Column informatiom of tables, views, join index & hash index etc.
- **dbc.indices:** Stores all the index related informatiom for tables, views, join index , hash index & secondary index etc.
- **dbc.errormsgs:** To error message for an error code.

Table / Database size

- **Table Size:** Table size can be determined from multiple tables for example : `dbc.allspace` & `dbc.tablesize`.
- **Database Size:** Database size can determined using Dbc.Diskspace.

Nodes and AMP info

```
/*Number of Nodes*/
SELECT COUNT(DISTINCT nodeid) FROM dbc.resusagescpu;

/*Number of Amps on each Node*/
SELECT nodeid,COUNT(DISTINCT vproc) number_of_amps
FROM dbc.ResCpuUsageByAmpView
GROUP BY nodeid;

/*Number of AMPs in the system*/
SELECT HASHAMP()+1;
```

- Number of rows in each AMP for a specific table:

```
SELECT
HASHAMP(HASHBUCKET(HASHROW(PIcolumn))), COUNT(*)
FROM tablename GROUP BY 1;
```

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 <procedurename> ( [parameter 1 data type, parameter 2 data type..] ) 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);
```

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 results in rows being 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.

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 manager, the list of all its indirect subordinates.

Employee	Name	ReportsTo
1	Tom	3
2	Jim	1
3	Will	0
4	Marius	1
5	Lucy	2
6	Julia	3

Hint

http://walkingoncoals.blogspot.com/2009/12/fun-with-recursive-sql-part-1.html