Subqueries and Joins

Subqueries

Subqueries in Teradata

- Subqueries are nested `SELECT` statement in order to provide output to outer query for data filtering purpose.
    - All subqueries must be enclosed in parentheses.
    - Subqueries can have multiple columns to match with main query.
    - Subqueries will always return unique list of values.
- Subqueries can be broadly classified into 2 categories:
    - Basic / Noncorrelated subquery
    - Correlated subquery

Restrictions for subqueries

- Subqueries can be nested up to a depth of 64(maximum) else it will fail with below error.
- `TOP n` cannot be used.
- `ORDER BY` cannot be used.

Basic subquery

- A basic subquery is a subquery that is independent of outer query but provides data to outer query to restrict result of final main query.
- **Example:**

```
SELECT * FROM table1
WHERE id IN
      ( SELECT
           id
        FROM
           table2
      );
```

Basic subquery (cont.)

```
SELECT * FROM table1
WHERE (id,transaction_dt) NOT IN
      ( SELECT
           id,
           transaction_dt
        FROM
           table2
      );
```

IN vs EXISTS

- Most selective filter in the subquery: use `IN`
- Most selective filter in the parent query: use `EXISTS`

Examples

- Which employee gave the order to a given customer?

```
SELECT e.employee_id, e.first_name,
e.last_name, e.salary
FROM employees e
WHERE EXISTS (SELECT 1
FROM orders o
WHERE e.employee_id = o.sales_rep_id
AND o.customer_id = 144);
```

```
SELECT e.employee_id, e.first_name,
e.last_name, e.salary
FROM employees e
WHERE e.employee_id IN (SELECT o.sales_rep_id
FROM orders o
WHERE o.customer_id = 144);
```

- **Trivia:** Which query is faster? (assume PI's in the join columns).

Examples

- Sales reps from department 80 that have sold something.

```
SELECT e.employee_id, e.first_name,
e.last_name, e.department_id, e.salary
FROM employees e
WHERE e.department_id = 80
AND e.job_id = 'SA_REP'
AND e.employee_id IN (SELECT o.sales_rep_id
FROM orders o);
```

```
SELECT e.employee_id, e.first_name,
e.last_name, e.salary
FROM employees e
WHERE e.department_id = 80
AND e.job_id = 'SA_REP'
AND EXISTS (SELECT 1
FROM orders o
WHERE e.employee_id = o.sales_rep_id);
```

- **Trivia:** Which query is faster? (assume PI's in the join columns).

Correlated subqueries: Example

- Same table referenced in the internal and external query.

```
/*Alias outside*/
SELECT *
FROM table_1 AS a
WHERE x < (SELECT AVG(table_1.x)
                FROM table_1
                WHERE table_1.n = a.n);

/* Alias inside */
SELECT *
FROM table_1
WHERE x < (SELECT AVG(a.x)
                FROM table_1 AS a
                WHERE table_1.n = a.n);
```

How are correlated subqueries processed

emp_no	emp_name	sex	age
101	Friedrich	F	23
102	Harvey	M	47
103	Agrawal	M	65
104	Valduriez	M	34

```
SELECT *
FROM employee AS e1
WHERE age < (SELECT MAX(age)
                    FROM employee AS e2
                    WHERE e1.sex = e2.sex);
```

How are correlated subqueries processed (cont.)

- Two copies of the table described earlier are generated, one as e1 and the other as e2.
- Evaluation of the inner query requires data from the outer, containing, query.

```
SELECT 101, 'Friedrich', 'F', 23
     FROM employee AS e1
     WHERE 23 < (SELECT MAX(age)
                 FROM employee AS e2
                 WHERE 'F' = e2.sex;

SELECT 108, 'Ghazal', 'F', 26
FROM employee as e1
WHERE 26 < (SELECT MAX(age)
                 FROM employee AS e2
                 WHERE 'F' = e2.sex;
```

Comparing Correlated and noncorrelated Subqueries

```
SELECT column_list_1
FROM table_list_1
WHERE predicate_1 (SELECT column_list_2
                   FROM table_list_2
                   WHERE predicate_2);
```

- If `predicate_2` does not include anything from `table_list_1`, non-correlated subquery (local).
- This restricts the number of its iterations to one. The results of the query are then joined with the results of the query made by the outer SELECT statement.

The dark side of correlated subqueries

- Correlated subqueries perform the subquery in parentheses once for each result row of the outer query.
- It does not necessarily produce a unique result for each of those iterations.

Example

Assume that `table_1` has columns `col_1` and `col_2`, while `table_2` has columns `col_3` and `col_4`. The following four rows exist in the two tables.

col_1	col_2	col_3	col_4
100	1	100	1
50	1	50	1
20	2	20	2
40	2	40	2

Example (cont.)

```
SELECT *
FROM table_1
WHERE col_1 IN (SELECT MAX(col_3)
                FROM table_2
                WHERE table_1.col_2=table_2.col_4);
```

- The subquery is performed four times: once for each row in table_1.

Example (cont.)

The result contains only 2 response rows because of the `MAX(col_3)` aggregation constraint and two of the subquery executions return a response row where col_1 is not in the result.

- The two rows returned are:
  col_1=100, col_2=1
  col_1=40, col_2=2.

Example (cont.)

- The four executions of the subquery return the following response rows:

col_3	col_4
100	1
100	1
40	2
40	2

- Only the first and fourth rows of `table_1` have a value for `col_1` in this result set.
- Without the `MAX` aggregate function, then all four rows of `table_1` would have been returned.

Subquery to Join

```sql
/*Sub query*/
SELECT e.*
FROM employee
WHERE DeptNo IN
(SELECT DeptNo
FROM department
WHERE DeptName LIKE 'IT');
```

```sql
/*Subquery to JOIN*/
SELECT e.*
FROM employee e
INNER JOIN department d
ON e.DeptNo = d.DeptNo
WHERE d.DeptName LIKE 'IT';
```

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>;
```