Advanced SQL Techniques

Part 2

Lateral Subqueries

- Let's look at the `car_portal` database again.
- Suppose you need to query the `car` table and, for each car, you need to assess its age by comparing it with the age of other cars of the same model.
- Furthermore, you want to query the number of cars of the same model.

```
car_portal=> SELECT car_id, manufacture_year,
car_portal->        CASE WHEN manufacture_year <= (SELECT avg(manufacture_year)
car_portal(>                                       FROM   car_portal_app.car
car_portal(>                                       WHERE  car_model_id = c.car_model_id) THEN 'old'
car_portal->             ELSE 'new'
car_portal->        END as age,
car_portal->        (SELECT count(*)
car_portal(>         FROM   car_portal_app.car
car_portal(>         WHERE car_model_id = c.car_model_id) AS same_model_count
car_portal-> FROM   car_portal_app.car c;
```

car_id	manufacture_year	age	same_model_count
1	2008	old	3
2	2014	new	6
3	2014	new	2
4	2010	new	2
5	2009	new	3
6	2007	new	4
7	2009	new	3
8	2013	new	2
9	2001	old	4
10	2013	new	3

220	2011	new	4
221	2006	old	6
222	2003	old	2
223	2013	new	4
224	2011	new	2
225	2007	new	2
226	2004	new	2
227	2000	old	2
228	2009	new	3
229	2004	old	4
231	2014	old	1

(230 rows)

- The power of these subqueries is that they can refer to the main table in their `WHERE` clause.
- This makes them easy.
- It's also very simple to add more columns to the query by adding more subqueries.
- On the other hand, there's a problem here: performance.
- The `car` table is scanned by the database server once for the main query, and then it's scanned twice for each retrieved row, that is, for the `age` and `same_model_count` columns.

```
car_portal=> SELECT car_id, manufacture_year,
car_portal->        CASE WHEN manufacture_year <= avg_year THEN 'old'
car_portal->             ELSE 'new'
car_portal->        END as age,
car_portal->        same_model_count
car_portal-> FROM   car_portal_app.car INNER JOIN (SELECT   car_model_id, avg(manufacture_year) avg_year, count(*) same_model_count
car_portal(>                                       FROM     car_portal_app.car
car_portal(>                                       GROUP BY car_model_id) subq
car_portal->        USING (car_model_id);
```

- The result is the same and the query is 20 times faster.
- However, this query is good only when retrieving many rows from the database.
- If you need to get information about only one car, the first query will be faster.

- You can see that the first query could perform better if it were possible to select two columns in the same subquery in the select-list, but this isn't possible.
- Scalar queries can return only one column.

- There is another way of using subqueries: it combines the advantages of the subqueries in the select-list, which can refer to the main table in their `WHERE` clause, with the subqueries in the `FROM` clause, which can return multiple columns.
- This can be done via lateral subqueries.
- Putting the `LATERAL` keyword before a subquery in the `FROM` clause makes it possible to reference any preceding item of the `FROM` clause from that subquery.

```
car_portal=> SELECT car_id, manufacture_year,
car_portal->        CASE WHEN manufacture_year <= avg_year THEN 'old'
car_portal->             ELSE 'new'
car_portal->        END as age,
car_portal->        same_model_count
car_portal-> FROM   car_portal_app.car c,
car_portal->        LATERAL (SELECT avg(manufacture_year) avg_year, count(*) same_model_count
car_portal(>                 FROM   car_portal_app.car
car_portal(>                 WHERE  car_model_id = c.car_model_id) subq;
```

- This query is approximately twice as fast as the first one, and is the best one to retrieve only one row from the car table.

- The `JOIN` syntax is also possible with `LATERAL` subqueries, though in many cases, `LATERAL` subqueries have a `WHERE` clause that makes their records depend on other subqueries or tables, which effectively joins them.

- When it comes to set-returning functions, it isn't necessary to use the `LATERAL` keyword.
- All functions that are mentioned in the `FROM` clause can already use the output of any preceding functions or subqueries:

```
car_portal=> SELECT a, b
car_portal-> FROM   generate_series(1,3) AS a, generate_series(a, a+2) AS b;
 a | b
---+---
 1 | 1
 1 | 2
 1 | 3
 2 | 2
 2 | 3
 2 | 4
 3 | 3
 3 | 4
 3 | 5
(9 rows)
```

- In the preceding query, the first function that has the a alias returns three rows.
- For each of these three rows, the second function is called, returning three more rows.

Advanced Grouping

- Databases are often used as a data source for any kind of reporting.
- In reports, it's quite common to display subtotals, totals, and grand totals that summarize the data rows in the same table, implying, in fact, grouping, and aggregation.

- Consider the following report, which displays the number of advertisements by car make and quarter, and displays the totals for each quarter (aggregating all makes), and the grand total.

```
car_portal=> SELECT   to_char(advertisement_date, 'YYYY-Q') as quarter, make, count(*)
car_portal-> FROM     advertisement a INNER JOIN car c
car_portal->          ON a.car_id = c.car_id
car_portal->          INNER JOIN car_model m
car_portal->          ON m.car_model_id = c.car_model_id
car_portal-> GROUP BY quarter, make;
```

quarter	make	count
2014-1	Alfa Romeo	16384
2014-1	Audi	40960
2014-1	BMW	106496
2014-1	Citroen	24576
2014-1	Eagle	57344
2014-1	Fiat	8192
2014-1	GMC	16384
2014-1	Infiniti	40960
2014-1	Jeep	40960
2015-1	KIA	40960
2015-1	Lincoln	8192
2015-1	Mercedes Benz	8192
2015-1	Nissan	8192
2015-1	Opel	16384
2015-1	Peugeot	32768
2015-1	Renault	57344
2015-1	Skoda	49152
2015-1	Toyota	24576
2015-1	Volvo	32768

(100 rows)

- To make the subtotals, you would need to execute additional queries, as follows:

```
car_portal=> SELECT   to_char(advertisement_date, 'YYYY-Q') as quarter, count(*)
car_portal-> FROM     advertisement a INNER JOIN car c
car_portal->          ON a.car_id = c.car_id
car_portal->          INNER JOIN car_model m
car_portal->          ON m.car_model_id = c.car_model_id
car_portal-> GROUP BY quarter;
 quarter |  count
---------+---------
 2014-1  |  991232
 2014-2  | 2179072
 2014-3  | 1810432
 2014-4  |  974848
 2015-1  |  466944
(5 rows)
```

```
car_portal=> SELECT count(*)
car_portal-> FROM   advertisement a INNER JOIN car c
car_portal->        ON a.car_id = c.car_id
car_portal->        INNER JOIN car_model m
car_portal->        ON m.car_model_id = c.car_model_id;
  count
---------
 6422528
(1 row)
```

- We could `UNION` them to produce the requested report, but PostgreSQL provides a better way of combining the three queries in a single one using a special construct, `GROUP BY GROUPING SETS`:

```
car_portal=> SELECT   to_char(advertisement_date, 'YYYY-Q') as quarter, make, count(*)
car_portal-> FROM     advertisement a INNER JOIN car c
car_portal->          ON a.car_id = c.car_id
car_portal->          INNER JOIN car_model m
car_portal->          ON m.car_model_id = c.car_model_id
car_portal-> GROUP BY GROUPING SETS ((quarter, make), (quarter), ())
car_portal-> ORDER BY quarter NULLS LAST, make NULLS LAST;
```

quarter	make	count
2014-1	Alfa Romeo	16384
2014-1	Audi	40960
2014-1	BMW	106496
2014-1	Citroen	24576
2014-1	Eagle	57344
2014-1	Fiat	8192
2014-1	GMC	16384
2014-1	Infiniti	40960
2014-1	Jeep	40960
2014-1	KIA	32768
2014-1	Lincoln	24576
2014-1	Mercedes Benz	32768
2014-1	Nissan	98304
2014-1	Opel	49152
2014-1	Peugeot	122880
2014-1	Renault	73728
2014-1	Toyota	131072
2014-1	Volvo	73728
2014-1		991232
2014-2	Audi	147456
2014-2	BMW	262144
2014-2	Citroen	327680

```
 2014-4  | Volkswagen     |   40960
 2014-4  | Volvo          |    8192
 2014-4  |                |  974848
 2015-1  | Alfa Romeo     |   24576
 2015-1  | Audi           |   40960
 2015-1  | BMW            |   40960
 2015-1  | Ferrari        |   40960
 2015-1  | Infiniti       |    8192
 2015-1  | Jeep           |   32768
 2015-1  | KIA            |   40960
 2015-1  | Lincoln        |    8192
 2015-1  | Mercedes Benz  |    8192
 2015-1  | Nissan         |    8192
 2015-1  | Opel           |   16384
 2015-1  | Peugeot        |   32768
 2015-1  | Renault        |   57344
 2015-1  | Skoda          |   49152
 2015-1  | Toyota         |   24576
 2015-1  | Volvo          |   32768
 2015-1  |                |  466944
         |                | 6422528
(106 rows)
```

- The `GROUPING SETS ((quarter, make), (quarter), ())` construct makes the query work as a `UNION ALL` operation over the same query, repeated several times with different `GROUP BY` clause:
  - `GROUP BY quarter, make`
  - `GROUP BY quarter`—here, the `make` field will get the `NULL` value
  - All the rows are grouped into a single group; both the `quarter` and `make` fields will get `NULL` values

- Generally speaking, the `GROUP BY` clause takes not just expressions, but also **grouping elements**, which could be expressions or constructs, such as GROUPING SETS.
- Other possible grouping elements are `ROLLUP` and `CUBE`:
  - `ROLLUP (a, b, c)` is equivalent to `GROUPING SETS ((a, b, c), (a, b), (a), ())`
  - `CUBE (a, b, c)` is equivalent to `GROUPING SETS ((a, b, c), (a, b), (a, c), (b, c), (a), (b), (c), ())`, which is all possible combinations of the argument expressions

- It is also possible to use `CUBE` and `ROLLUP` inside `GROUPING SETS`, as well as combining different grouping elements in one `GROUP BY` clause, as follows:
  - `GROUP BY a, CUBE (b, c)`
  - This would be equivalent to `GROUP BY GROUPING SETS ((a, b, c), (a, b), (a, c), (a))`, which is basically a combination of the grouping elements.

- Although the result of `GROUPING SETS` is the same as of `UNION ALL` operation over several queries with different `GROUP BY` clauses, in the background, it works differently from the `SET` operation.
- PostgreSQL will scan the tables and perform joining and filtering only once.
- This means that the use of these grouping techniques can help optimize the performance of your solutions.

- The documentation for `GROUPING SETS` is available at https://www.postgresql.org/docs/current/static/queries-table-expressions.html#QUERIES-GROUPING-SETS

Advanced Aggregation

- There are several aggregating functions that are executed in a special way.

- The first group of such aggregating functions is called **ordered-set aggregates**.
- They take into account not just the values of the argument expressions, but also their order.
- They are related to statistics and calculate percentile values.

- A percentile is the value of a group of numbers when the given percentage of other values is less than the percentile.
- For example, if a value is at the 95th percentile, this means that it's greater than 95 percent of the other values.
- In PostgreSQL, you can calculate a continuous or discrete percentile using the `percentile_cont` and `percentile_disc` functions, respectively.

- A discrete percentile is one of the actual values of a group, while a continuous percentile is an interpolated value between two actual values.
- It's possible to calculate the percentile for a given fraction, or several percentile values for a given array of fractions.

```
joshua@joshua-Virtual-Machine:~/Documents/Learning-PostgreSQL-11-Third-Edition-master/Chapter06/Advanced Query Writing$ sudo -u postgres psql -f schema.sq
l
[sudo] password for joshua:
psql:schema.sql:3: ERROR:  role "car_portal_app" already exists
DROP DATABASE
CREATE DATABASE
You are now connected to database "car_portal" as user "postgres".
CREATE SCHEMA
SET
SET
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
joshua@joshua-Virtual-Machine:~/Documents/Learning-PostgreSQL-11-Third-Edition-master/Chapter06/Advanced Query Writing$ sudo -u postgres psql -f data.sql
You are now connected to database "car_portal" as user "postgres".
SET
SET
INSERT 0 481
INSERT 0 99
INSERT 0 229
INSERT 0 146
INSERT 0 784
```

```
car_portal=> SELECT percentile_disc (ARRAY[0.25, 0.5, 0.75]) WITHIN GROUP (ORDER BY cnt)
car_portal-> FROM   (SELECT   count(*) cnt
car_portal(>         FROM     car_portal_app.advertisement
car_portal(>         GROUP BY car_id) subq;
 percentile_disc
-----------------
 {2,3,5}
(1 row)
```

- Another ordered-set aggregating function is `mode`.
- It returns the most-frequently-appearing value of the group.
- If two values appear the same number of times, the first of them will be returned.
- For example, the following query gets the ID of the most frequent car model from the database:

```
car_portal=> SELECT mode() WITHIN GROUP (ORDER BY car_model_id)
car_portal-> FROM   car_portal_app.car;
 mode
------
   64
(1 row)
```

- Another group of aggregates that use the same syntax are the **hypothetical-set aggregating functions**.
- They are `rank`, `dense_rank`, `percent_rank`, and `cume_dist`.
- There are window functions with the same names.
- Window functions take no argument and return the result for the current row.
- Aggregate functions have no current row because they are evaluated for a group of rows.
- However, they take an argument: the value for the hypothetical current row.

- For example, the `rank` aggregate function returns the rank of a given value in the ordered set as if that value existed in the set:

```
car_portal=> SELECT rank(2) WITHIN GROUP (ORDER BY a)
car_portal-> FROM   generate_series(1,10,3) a;
 rank
------
    2
(1 row)
```

- Another aggregate-function topic worth mentioning is the `FILTER` clause.
- The `FILTER` clause filters the rows that are passed to the particular aggregating function based on a given condition.

- For example, suppose you need to count the number of cars in the database for each car model separately, for each number of doors.
- If you group the records according to these two fields, the result will be correct but not very convenient to use in reporting:

```
car_portal=> SELECT   car_model_id, number_of_doors, count(*)
car_portal-> FROM     car_portal_app.car
car_portal-> GROUP BY car_model_id, number_of_doors;
```

- A better query, which is suitable for reporting, can be created using CASE operators:

```
car_portal=> SELECT   car_model_id,
car_portal->          count(CASE WHEN number_of_doors = 2 THEN 1 END) doors2,
car_portal->          count(CASE WHEN number_of_doors = 3 THEN 1 END) doors3,
car_portal->          count(CASE WHEN number_of_doors = 4 THEN 1 END) doors4,
car_portal->          count(CASE WHEN number_of_doors = 5 THEN 1 END) doors5
car_portal-> FROM     car_portal_app.car
car_portal-> GROUP BY car_model_id, number_of_doors;
```

car_model_id	doors2	doors3	doors4	doors5
62	0	0	1	0
54	0	0	0	1
12	0	0	2	0
72	0	0	1	0
61	0	2	0	0
91	0	0	1	0
88	0	0	2	0
85	0	0	2	0
53	0	1	0	0
16	0	0	0	2
87	0	0	1	0
65	0	0	0	3
2	0	0	0	1
90	0	0	0	3
33	0	1	0	0
76	0	1	0	0
74	0	0	0	2
15	0	0	2	0
98	0	0	0	2

(161 rows)

- The `FILTER` clause makes the query much clearer:

```
car_portal=> SELECT   car_model_id,
car_portal->          count(*) FILTER (WHERE number_of_doors = 2) doors2,
car_portal->          count(*) FILTER (WHERE number_of_doors = 3) doors3,
car_portal->          count(*) FILTER (WHERE number_of_doors = 4) doors4,
car_portal->          count(*) FILTER (WHERE number_of_doors = 5) doors5
car_portal-> FROM     car_portal_app.car
car_portal-> GROUP BY car_model_id;
```

car_model_id	doors2	doors3	doors4	doors5
55	0	1	1	0
27	0	0	1	3
23	0	2	1	1
56	0	0	0	1
91	0	1	1	0
58	0	0	2	1
8	0	0	1	0
87	0	1	1	0
74	0	0	1	2
29	0	1	0	1
54	0	0	1	1
49	0	0	1	0
47	0	0	1	2
3	0	2	0	0
17	0	0	2	1
28	0	1	0	0
20	0	2	0	2
33	0	1	0	0
76	0	1	1	1
5	0	0	0	1
64	0	2	1	4

(86 rows)