Advanced SQL Techniques

Part 1

Selecting the First Records

- Quite often, it's necessary to find the first records based on some criteria.
- For example, let's take the `car_portal` database; suppose you need to find the earliest advertisement for each `car_id` in the advertisement table.

```
car_portal=> SELECT advertisement_id, advertisement_date, adv.car_id, seller_account_id
car_portal-> FROM   car_portal_app.advertisement adv INNER JOIN (SELECT   car_id, min(advertisement_date) min_date
car_portal(>                                                     FROM     car_portal_app.advertisement
car_portal(>                                                     GROUP BY car_id ) first
car_portal->        ON adv.car_id=first.car_id AND
car_portal->           adv.advertisement_date = first.min_date;
```

- However, if the ordering logic is complex and cannot be implemented using the `min` function, this approach won't work.

- Although window functions can solve the problem, they aren't always convenient to use:

```
car_portal=> SELECT DISTINCT first_value(advertisement_id) OVER w AS advertisement_id,
car_portal->        min(advertisement_date) OVER w AS advertisement_date,
car_portal->        car_id,
car_portal->        first_value(seller_account_id) OVER w AS seller_account_id
car_portal-> FROM   car_portal_app.advertisement
car_portal-> WINDOW w AS (PARTITION BY car_id ORDER BY advertisement_date);
```

- In the preceding code, `DISTINCT` is used to remove the duplicates that were grouped together in the previous example.

- PostgreSQL provides an explicit way of selecting the first record within each group.
- The `DISTINCT ON` construct is used for this.
- The syntax is as follows:
  - `SELECT DISTINCT ON (<expression_list>) <Select-List>`
    `...`
    `ORDER BY <order_by_list>`

- In the preceding code snippet, for each distinct combination of values of `expression_list`, only the first record will be returned by the `SELECT` statement.
- The `ORDER BY` clause is used to define a rule to determine which record is the first.
- The `expression_list` from the `DISTINCT ON` construct must be included in the `order_by_list` list.

```
car_portal=> SELECT   DISTINCT ON (car_id) advertisement_id, advertisement_date, car_id, seller_account_id
car_portal-> FROM     car_portal_app.advertisement
car_portal-> ORDER BY car_id, advertisement_date;
```

Selecting a Data Sample

- Suppose the `car_portal` database is huge and there is a task to collect some statistics about the data.
- The statistics don't have to be exactly accurate, estimated values would be good enough.
- It's important to collect the data quickly.
- PostgreSQL provides a way to query a random fraction of a table's records.

- The syntax is as follows:
  - ```
    SELECT ...
    FROM <table>
    TABLESAMPLE <sampling_method> ( <argument> [, ...] )
    [ REPEATABLE ( <seed> ) ]
    ```
    ```

- The `sampling_method` can be either `BERNOULLI` or `SYSTEM`—these are provided with a standard PostgreSQL installation.
- More sampling methods are available as extensions.
- Both the sampling methods take one argument, which is a probability that a row will be selected, as a percentage.

- The difference between the two methods is that `BERNOULLI` implies a full table scan and the algorithm will decide for each individual row whether it should be selected or not.
- The `SYSTEM` method does the same but on the level of blocks of rows.
  - The latter is significantly faster, but the results are less accurate because of the different numbers of rows in different blocks.

- The `REPEATABLE` keyword is used to specify the seed value used by the random functions in the sampling algorithm.
- The results of queries with the same seed value is the same, as long as the data in the table doesn't change.

- Sampling happens at the moment the data from a table is read, before any filtering, grouping, or other operations are performed.

- To illustrate sampling, let's increase the number of records in the advertisement table by executing the following query:

```
car_portal=> INSERT INTO car_portal_app.advertisement (advertisement_date, car_id, seller_account_id)
car_portal-> SELECT advertisement_date, car_id, seller_account_id
car_portal-> FROM   car_portal_app.advertisement;
INSERT 0 784
```

- This simply duplicates the number of records in the table.
- If this is executed many times, the table will be pretty big.
- For the following example, that query was executed 12 more times.
- If we try to select the total count of records, it takes a fair bit of time:

```
car_portal=> \timing
Timing is on.
car_portal=> SELECT count(*) FROM car_portal_app.advertisement;
  count
---------
 6422528
(1 row)

Time: 172.417 ms
```

- Now let's try sampling:

```
car_portal=> SELECT count(*) * 100
car_portal-> FROM   car_portal_app.advertisement
car_portal-> TABLESAMPLE SYSTEM (1);
 ?column?
----------
  6452700
(1 row)

Time: 7.763 ms
```

- The second query is much faster; however, the result is not very accurate.
- We multiply the count function by 100 because only one percent of the rows is selected.

- If this query is performed several times, the results will always be different and, sometimes, can be quite different from the real count of rows in the table.
- To improve quality, the fraction of the rows that are selected can be increased:

```
car_portal=> SELECT count(*) * 10
car_portal-> FROM   car_portal_app.advertisement
car_portal-> TABLESAMPLE SYSTEM (10);
 ?column?
----------
  6604990
(1 row)

Time: 62.669 ms
```

- The time increased proportionally.

Set-Returning Functions

- Functions in PostgreSQL can return not only single values, but also relations.
- Such functions are called **set-returning functions**.

- This is a typical task for every SQL developer: generate a sequence of numbers, each in a separate record.
- This sequence or relation can have many use cases.
- For example, suppose you need to select data from the `car_portal` database to count the number of cars for each year of manufacture, from `2010` till `2015`, and you need to show zero for the years where no cars exist in the system at all.
- A simple SELECT query from the only car table cannot be used to implement this logic.
- It isn't possible to count records that are not present in the table data.

- That's why it would be useful if there was a table with the numbers from `2010` to `2015`.
- Then it could be outer-joined to the results of the query.

- You could create this table in advance, but that's not a very good idea, because the number of records for such a table wouldn't be known in advance, and if you created a very big table, it would be a waste of disk space in most cases.

- There is a function, `generate_series`, that serves exactly this purpose: it returns a set of integers with the given start and stop values.
- The query for the example would be as follows:

```
car_portal=> SELECT   years.manufacture_year, count(car_id)
car_portal-> FROM     generate_series(2010, 2015) as years (manufacture_year) LEFT JOIN car_portal_app.car
car_portal->          ON car.manufacture_year = years.manufacture_year
car_portal-> GROUP BY years.manufacture_year
car_portal-> ORDER BY 1;
 manufacture_year | count
------------------+-------
             2010 |    11
             2011 |    12
             2012 |    12
             2013 |    21
             2014 |    17
             2015 |     0
(6 rows)
```

- It's possible to specify a step when calling `generate_series`:

```
car_portal=> SELECT *
car_portal-> FROM   generate_series(5, 11, 3);
 generate_series
-----------------
               5
               8
              11
(3 rows)
```

- The `generate_series` function can also return a set of `timestamp` values:

```
car_portal=> SELECT *
car_portal-> FROM   generate_series('2015-01-01'::date, '2015-01-31'::date, interval '7 days');
    generate_series
------------------------
 2015-01-01 00:00:00+08
 2015-01-08 00:00:00+08
 2015-01-15 00:00:00+08
 2015-01-22 00:00:00+08
 2015-01-29 00:00:00+08
(5 rows)
```

- There are a couple of other set-returning functions designed to work with arrays:
  - `generate_subscripts`: This generates numbers that can be used to reference the elements in a given array for the given dimension(s). This function is useful for enumerating the elements of an array in a SQL statement.
  - `unnest`: This transforms a given array into a set or rows where each record corresponds to an element of the array.

- Set-returning functions are also called **table functions**.

- Table functions can return sets of rows of a predefined type, like `generate_series` returns a set of `int` or `bigint` (depending on the type of input argument).
- Moreover, they can return a set of abstract type records.
- This allows a function to return different numbers of columns depending on their arguments.
- SQL requires that the row structure of all input relations is defined so that the structure of the result set will also be defined.
- That's why table functions that return sets of records, when used in a query, can be followed by a definition of their row structure:
  - `function_name (<arguments>) [AS] alias (column_name column_type [, ...])`

- The output of several set-returning functions can be combined together as if they were joined on the position of each row.
- The `ROWS FROM` syntax construct is used for this, as follows:
  - `ROWS FROM (function_call [,…]) [[AS] alias (column_name [,...])]`
- The preceding construct will return a relation, therefore, it can be used in a `FROM` clause just like any other relation.
- The number of rows is equal to the largest output of the functions.
- Each column corresponds to the respective function in the `ROWS FROM` clause.
- If a function returns fewer rows than other functions, the missing values will be set to `NULL`.

```
car_portal=> SELECT foo.a, foo.b
car_portal-> FROM   ROWS FROM (generate_series(1,3), generate_series(1,7,2)) AS foo(a, b);
 a | b
---+---
 1 | 1
 2 | 3
 3 | 5
   | 7
(4 rows)
```