

## Window Functions, Data Granularity

*Instructor: Aditya Parameswaran**Covers Lecture(s): 11 – 12*

### I Window Functions

1. Let's revisit a fictional scenario where instructors order boba for every enrolled Data 101 student. In the spirit of boosting morale during midterm season, the instructors have decided to allow multiple boba orders per student. Order IDs are assigned in the same order that orders are received.

- `orders (student_id, student_name, order_id, order_info, price)`

To keep track of the spirited spending, we need to calculate the cumulative total spent by each student up to and including each of their orders. Write a SQL query that returns the student's name, order ID, and their cumulative total spend from the orders table. Make sure the orders are returned in ascending order of `student_name` and `order_id`.

```
SELECT
  student_name,
  order_id,
  SUM(price) OVER (PARTITION BY student_id ORDER BY order_id ASC) AS
  cumulative_total_spend
FROM orders
ORDER BY student_name, order_id;
```

The `SUM()` window function calculates the cumulative total spend for each student as we progress through their orders, ordered by the `order_id`. The `PARTITION BY` clause ensures that the cumulative sum is calculated separately for each student.

Moreover, we cannot omit the `ORDER BY` keyword, because having the keyword changes the scope of the window frame for each row. According to the PostgreSQL documentation:

“For each row, there is a set of rows within its partition called its window frame. Some window functions act only on the rows of the window frame, rather than of the whole partition. By default, if `ORDER BY` is supplied then the frame consists of all rows from the start of the partition up through the current row, plus any following rows that are equal to the current row according to the `ORDER BY` clause. When `ORDER BY` is omitted the default frame consists of all rows in the partition.”

2. Let's also track how much the students' order price compares with a previous order. For each order in `orders` table, find the price of the previous order placed by the same student. Show the student's name, the order ID, the current order's price, and the previous order's price. Sort in ascending order of the student's name and then order ID.

```
SELECT student_name, order_id, price AS current_price,
  LAG(price, 1) OVER (PARTITION BY student_id
  ORDER BY order_id) AS previous_price
FROM orders ORDER BY student_name, order_id;
```

The LAG() window function looks 1 row back based on the `order_id` for the previous order. Similar to question 1, we use the `PARTITION BY` clause to only consider orders from the same student.

3. Assume that we can process at most 10 orders at a time. For each order in the `orders` table, compute the sum of prices of all orders whose `order_id` is 10 before the current order's `order_id` (i.e., current order and previous orders with `order_id` difference  $\leq 10$ ), including the current order. Show the student's name, the order ID, and a column called `rolling_sum_10_orders` showing this total. Return the results ordered by `order_id`.

```
SELECT student_name, order_id, price,
SUM(price) OVER (
ORDER BY order_id
RANGE BETWEEN 10 PRECEDING AND CURRENT ROW
) AS rolling_sum_10_orders
FROM orders
ORDER BY order_id;
```

The `SUM()` is calculated over the `RANGE` between the current order ID minus 10 and the current order ID. `ROWS` also works if order IDs are assumed to be consecutive (i.e., there are no gaps).

4. In Project 1, we worked with a `movie_sample` table to explore gross earnings for each movie and stored the results in a view called `movie_gross`. Let's explore how to leverage window functions to compute the average earnings **per genre** across movies, as well as the movie's rank within its genre based on gross earnings.

| id      | movie_id | info_type_id | info                                  |
|---------|----------|--------------|---------------------------------------|
| 6215794 | 2022105  | 3            | Comedy                                |
| 6890556 | 2441099  | 3            | Drama                                 |
| 1457358 | 1924463  | 107          | \$22,946,000 (USA) (25 December 2005) |

**Table 1:** Sample rows of `movie_info_sample`.

| gross       | movie_id | title        |
|-------------|----------|--------------|
| 760507625.0 | 1704289  | Avatar       |
| 658672302.0 | 2438179  | Titanic      |
| 623357910.0 | 2346436  | The Avengers |

**Table 2:** Sample rows of `movie_gross`

Create a new view called `movie_avg_genre` with the following columns: `movie_id`, `title`, `gross`, `genre`, `avg_gross_genre`, `genre_rank`. Following the view definition, write a `SELECT` query to return the rows for the movie 'Mr. & Mrs. Smith' ordered by genre alphabetically.

- **gross:** The gross earnings of the given movie
- **avg\_gross\_genre:** The *average* gross earnings of movies with the same genre as the given movie. If a movie belongs to multiple genres, it should appear in multiple rows, with each row corresponding to a different genre.
- **genre\_rank:** The rank of each movie within its genre based on gross earnings (highest gross = 1). If two movies have the same gross earnings, they should share the same rank and skip the next rank. For example, if Avatar and Titanic were in the same genre and were ranked 1, the next rank available would be rank 3.

Hint: Check out the movie genre info type, with `info_type_id = 3` from `movie_info_sample`.

```
DROP VIEW IF EXISTS movie_avg_genre;
CREATE VIEW movie_avg_genre AS

-- create a CTE here, which gets the genre(s) and gross of each movie --
WITH movie_genre AS (
SELECT mg.movie_id, mg.title, mis.info AS genre, mg.gross
FROM movie_gross mg
INNER JOIN movie_info_sample mis
ON mg.movie_id = mis.movie_id
WHERE mis.info_type_id = 3
)

-- for each movie, get its gross, average gross, and rank for its genre(s) --
SELECT movie_id, title, genre, gross,
AVG(gross) OVER (PARTITION BY genre) AS average_gross_genre,
RANK() OVER (PARTITION BY genre ORDER BY gross DESC) as genre_rank
FROM movie_genre;

SELECT * FROM movie_avg_genre
WHERE title = 'Mr. & Mrs. Smith'
ORDER BY genre;
```

The `AVG()` window function calculates the average gross earnings for each movie genre. The `RANK()` window functions calculates the ranking for each movie within the genre. The `PARTITION BY` clause ensures that the average gross and genre rank is calculated separately for each genre.

5. Expanding on the previous question, besides finding the average gross for movies in each genre but also finding the total gross for each genre.

Create a view called `movie_avg_total_genre` with the following columns: `movie_id`, `title`, `gross`, `genre`, `avg_gross`, `total_gross`.

- **gross**: The gross earnings of the given movie
- **avg\_gross**: The average gross earnings of movies with the same genre as the given movie. If a movie belongs to multiple genres, it should appear in multiple rows, with each row corresponding to a different genre.
- **total\_gross**: The total gross earning of movies with the same genre as the given movie.

Bonus: Instead of writing out two separate windows, is it possible to simplify?

```
DROP VIEW IF EXISTS movie_avg_total_genre;
CREATE VIEW movie_avg_total_genre AS

-- create a CTE here, which gets the genre(s) and gross of each movie --
WITH movie_genre AS (
SELECT mg.movie_id, mg.title, mis.info AS genre, mg.gross
FROM movie_gross mg
INNER JOIN movie_info_sample mis
```

```

ON mg.movie_id = mis.movie_id
WHERE mis.info_type_id = 3
)

-- for each movie, get its gross and average gross for its genre(s) --
SELECT movie_id, title, genre, gross,
AVG(gross) OVER (PARTITION BY genre) AS avg_gross,
SUM(gross) OVER (PARTITION BY genre) AS total_gross
FROM movie_genre;

-- Query to retrieve 'Mr. & Mrs. Smith' data --
SELECT * FROM movie_avg_total_genre
WHERE title = 'Mr. & Mrs. Smith'
ORDER BY genre;

```

#### **Alternative way of using window functions:**

```

-- for each movie, get its gross and average gross for its genre(s) --
SELECT movie_id, title, genre, gross,
AVG(gross) OVER w AS avg_gross,
SUM(gross) OVER w AS total_gross
FROM movie_genre
WINDOW w AS (PARTITION BY genre);

```

## II Data Granularity: Walking a Hierarchy of Arbitrary Depth

In this section, we will work with a table with a complex hierarchy: the taxonomic hierarchy of biology from US National Center for Biotechnology Information. The relevant schema for this exercise is as follows and note that (`tax_id`, `rank`) is what makes a unique node in the table:

- `nodes` (`tax_id` INTEGER, `parent` INTEGER, `rank` TEXT)
- `names` (`tax_id` INTEGER, `name_txt` TEXT)

6. Write a SQL query to find the node representing the *Animalia* kingdom. Your query should return the kingdom name and its corresponding node information.

```
SELECT name_txt, nodes.*
FROM nodes NATURAL JOIN names
WHERE name_txt = 'Animalia';
```

7. Let us drill down into the *Animalia* kingdom. First, find all children nodes of the *Animalia* kingdom (ID 33208). Your query should return the `tax_id` and `rank` of each child node and their parent. Hint: use self-join.

```
SELECT child.tax_id cid, child.rank crank, parent.tax_id pid, parent.rank prank
FROM nodes parent, nodes child
WHERE child.parent = parent.tax_id
AND parent.tax_id = 33208;
```

8. Next, find the names of these children nodes, along with the names of their parents *Animalia*. Your query should return the `tax_id`, `rank`, and `name_txt` of each child and parent node.

```
WITH parent_child AS (
SELECT parent.tax_id pid, parent.rank prank, child.tax_id cid, child.rank crank
FROM nodes parent, nodes child
WHERE child.parent = parent.tax_id
)
SELECT pid, prank, parent.name_txt ptext, cid, crank, child.name_txt ctext
FROM names parent, names child, parent_child
WHERE parent.tax_id = parent_child.pid
AND child.tax_id = parent_child.cid
AND parent.tax_id = 33208
LIMIT 10;
```

9. In the biological taxonomy data, it is common for the same phylum to have multiple synonymous names, all sharing the same `tax_id`. Write a SQL query to return the parent node (*Animalia*) and its direct child nodes. For each child and parent node, aggregate all its synonymous names into a single JSON array.

```
WITH parent_child AS (  
  SELECT parent.tax_id AS pid, parent.rank as prank,  
         child.tax_id AS cid, child.rank as crank  
  FROM nodes parent, nodes child  
  WHERE child.parent = parent.tax_id  
)  
SELECT pid, parent_child.prank, json_agg(parent.name_txt), cid,  
       parent_child.crank, json_agg(child.name_txt)  
FROM names parent, names child, parent_child  
WHERE parent.tax_id = parent_child.pid  
AND child.tax_id = parent_child.cid  
AND parent.tax_id = 33208  
GROUP BY pid, parent_child.prank, cid, parent_child.crank;
```

10. **[Optional, Challenge]** How can we drill down one more layer? What if we want to get the names of all the classes under the *Animalia* kingdom?

```
WITH double_parent_child AS (  
  SELECT grandparent.tax_id AS gid, grandparent.rank as grank,  
         parent.tax_id AS pid, parent.rank as prank,  
         child.tax_id AS cid, child.rank as crank  
  FROM nodes grandparent, nodes parent, nodes child  
  WHERE child.parent = parent.tax_id  
        AND parent.parent = grandparent.tax_id  
)  
SELECT gid, grank, json_agg(grandparent.name_txt),  
       pid, prank, json_agg(parent.name_txt),  
       cid, crank, json_agg(child.name_txt)  
FROM names grandparent, names parent, names child, double_parent_child de  
WHERE parent.tax_id = de.pid  
AND child.tax_id = de.cid  
AND grandparent.tax_id = de.gid  
AND grandparent.tax_id = 33208  
GROUP BY gid, grank, pid, prank, cid, crank;
```