```
-- WINDOW Functions

Warm up: Create a running total by month

SELECT standard_qty,
       DATE_TRUNC('Month', occurred_at),
           -- SUM(standard_qty) OVER (ORDER BY occurred_at)
AS running_total
           -- take the sum of standard_qty across all rows
leading up to a given in order by occurred_at
           SUM(standard_qty) OVER (PARTITION BY
DATE_TRUNC('Month', occurred_at) ORDER BY occurred_at) as
running_total
FROM orders

-- Observations: PARTITION BY creates a sub-set where the
aggregation is to be done.
                 Running this Query without ORDER BY
results in a column where all rows are the aggregations of
the entire column.


11. Create a running total of standard_amt_usd (in the
orders table) over order time with no date truncation. Your
final table should have two columns: one with the amount
being added for each new row, and a second with the running
total.

SELECT standard_amt_usd,
       SUM(standard_amt_usd) OVER (ORDER BY occurred_at) AS
running_total
FROM orders

12. Create a running total of standard_amt_usd (in the
orders table) over order time, but this time, date truncate
occurred_at by year and partition by that same year-
truncated occurred_at variable. Your final table should
have three columns: One with the amount being added for
each row, one for the truncated date, and a final column
with the running total within each year.

SELECT standard_amt_usd,
       DATE_TRUNC('year',occurred_at) trunc_year,
       SUM(standard_amt_usd) OVER (PARTITION BY
```

```
DATE_TRUNC('year',occurred_at) ORDER BY occurred_at) AS
yearly_running_total
FROM orders

-- This time using a CTE, I just wanted to see the yearly
total!

WITH t1 AS (
             SELECT standard_amt_usd,
             DATE_TRUNC('year',occurred_at) trunc_year,
             SUM(standard_amt_usd) OVER (PARTITION BY
DATE_TRUNC('year',occurred_at) ORDER BY occurred_at) AS
yearly_running_total
             FROM orders
           )

SELECT trunc_year,
       SUM(yearly_running_total) yearly_total
FROM t1
GROUP BY 1


-- ROW_NUMBER & RANK

Warm up: Examples

A.

SELECT id,
       account_id,
       DATE_TRUNC('month',occurred_at) AS month,
       -- ROW_NUMBER() OVER(ORDER BY occurred_at) AS
continuos_row_num,
       -- ROW_NUMBER() OVER (PARTITION BY account_id ORDER
BY occurred_at) AS partitioned_by_acct_id
       -- RANK() OVER(PARTITION BY account_id ORDER BY
DATE_TRUNC('month',occurred_at)) AS rank
       DENSE_RANK() OVER(PARTITION BY account_id ORDER BY
DATE_TRUNC('month',occurred_at)) AS dense_rank
FROM orders


B.
```

```
SELECT id,
       account_id,
       standard_qty,
       DATE_TRUNC('month',occurred_at) AS month,
       DENSE_RANK() OVER (PARTITION BY account_id ORDER BY
DATE_TRUNC('month',occurred_at)) AS dense_rank,
       RANK() OVER (PARTITION BY account_id ORDER BY
DATE_TRUNC('month',occurred_at)) AS rank,
       SUM(standard_qty) OVER (PARTITION BY account_id
ORDER BY DATE_TRUNC('month',occurred_at)) AS running_total,
       COUNT(standard_qty) OVER (PARTITION BY account_id
ORDER BY DATE_TRUNC('month',occurred_at)) AS
count_standard_qty,
       AVG(standard_qty) OVER (PARTITION BY account_id
ORDER BY DATE_TRUNC('month',occurred_at)) AS
avg_standard_qty,
       MIN(standard_qty) OVER (PARTITION BY account_id
ORDER BY DATE_TRUNC('month',occurred_at)) AS
min_standard_qty,
       MAX(standard_qty) OVER (PARTITION BY account_id
ORDER BY DATE_TRUNC('month',occurred_at)) AS
max_standard_qty
FROM orders
```

C. Use Window Alias to re-write the above queries:

```
SELECT id,
       account_id,
       standard_qty,
       DATE_TRUNC('year',occurred_at) AS year,
       DENSE_RANK() OVER account_year_window AS dense_rank,
       RANK() OVER account_year_window AS rank,
       SUM(standard_qty) OVER account_year_window AS
running_total,
       COUNT(standard_qty) OVER account_year_window AS
count_standard_qty,
       AVG(standard_qty) OVER account_year_window AS
avg_standard_qty,
       MIN(standard_qty) OVER account_year_window AS
min_standard_qty,
       MAX(standard_qty) OVER account_year_window AS
max_standard_qty
```

```
FROM orders
WHERE account_id = 1021
WINDOW account_year_window AS (PARTITION BY account_id
ORDER BY DATE_TRUNC('year',occurred_at))
```

13. Select the id, account_id, and total variable from the orders table, then create a column called total_rank that ranks this total amount of paper ordered (from highest to lowest) for each account using a partition. Your final table should have these four columns.

```
SELECT id,
       account_id,
       total,
       RANK() OVER(PARTITION BY account_id ORDER BY total)
AS total_rank
FROM orders

-- the resulting query is partitioned by account_id and
ordered by total.


-- LAG & LEAD
-- LAG is previous, LEAD is next
```

14. Comparing a Row to Previous Row
In the previous video, Derek outlines how to compare a row to a previous or subsequent row. This technique can be useful when analyzing time-based events. Imagine you're an analyst at Parch & Posey and you want to determine how the current order's total revenue ("total" meaning from sales of all types of paper) compares to the next order's total revenue.

```
SELECT account_id,
       standard_sum,
       LAG(standard_sum) OVER (ORDER BY standard_sum) AS
lag,
       LEAD(standard_sum) OVER (ORDER BY standard_sum) AS
lead,
       standard_sum - LAG(standard_sum) OVER (ORDER BY
standard_sum) AS lag_difference,
```

```
       LEAD(standard_sum) OVER (ORDER BY standard_sum) -
standard_sum AS lead_difference
FROM (
       SELECT account_id,
              SUM(standard_qty) AS standard_sum
       FROM orders
       GROUP BY 1
     ) sub
```

Modify Derek's query from the previous video in the SQL Explorer below to perform this analysis. You'll need to use occurred_at and total_amt_usd in the orders table along with LEAD to do so. In your query results, there should be four columns: occurred_at, total_amt_usd, lead, and lead_difference.

My Solution:

```
SELECT occurred_at,
       total_amt_usd,
       LEAD(total_amt_usd) OVER(ORDER BY occurred_at) -
total_amt_usd as difference_from_next_order
FROM (
       SELECT occurred_at,
              SUM(total_amt_usd) AS total_amt_usd
       FROM orders
       GROUP BY 1
     ) sub


-- PERCENTILES
```

Warm up:

```
SELECT account_id,
       date_trunc('month',occurred_at) AS month,
       total,
       NTILE(4) OVER(ORDER BY total) AS quartile,
       NTILE(5) OVER(ORDER BY total) AS quintile,
       NTILE(100) OVER(ORDER BY total) AS percentile
FROM orders
-- WHERE account_id = 1001
```

15. Use the NTILE functionality to divide the accounts into 4 levels in terms of the amount of standard_qty for their orders. Your resulting table should have the account_id, the occurred_at time for each order, the total amount of standard_qty paper purchased, and one of four levels in a standard_quartile column.

```
SELECT account_id,
       occurred_at,
       standard_qty,
       NTILE(4) OVER(PARTITION BY account_id ORDER BY
standard_qty) AS standard_quartile
FROM orders
```

16. Use the NTILE functionality to divide the accounts into two levels in terms of the amount of gloss_qty for their orders. Your resulting table should have the account_id, the occurred_at time for each order, the total amount of gloss_qty paper purchased, and one of two levels in a gloss_half column.

```
SELECT account_id,
       occurred_at,
       gloss_qty,
       NTILE(2) OVER(PARTITION BY account_id ORDER BY
gloss_qty) AS gloss_half
FROM orders
```

17. Use the NTILE functionality to divide the orders for each account into 100 levels in terms of the amount of total_amt_usd for their orders. Your resulting table should have the account_id, the occurred_at time for each order, the total amount of total_amt_usd paper purchased, and one of 100 levels in a total_percentile column.

```
SELECT account_id,
       occurred_at,
       total_amt_usd,
       NTILE(100) OVER(PARTITION BY account_id ORDER BY
total_amt_usd) AS total_percentile
FROM orders
```