-- WINDOW FUNCTION

-- **A window function** performs a calculation across a set of table rows that are somehow **related to the current row.** This is comparable to the type of calculation that can be done with an aggregate function. But unlike regular aggregate functions, use of **a window function does not cause rows to become grouped into a single output row — the rows retain their separate identities.** Behind the scenes, the window function is able to access more than just the current row of the query result.

-- In SQL, window functions operate on a set of rows called a window frame. They return a single value for each row from the underlying query. The window frame (or simply window) is defined using the **OVER()** clause. This clause also allows defining a window based on a specific column (similar to GROUP BY). To calculate the returned values, window functions may use aggregate functions, but they will use them with the OVER() clause. Note that **the rows are not collapsed;** we still have one row for each of our transactions.

Window functions differ from aggregate functions used with GROUP BY in that they:

- Use `OVER()` instead of `GROUP BY()` to define a set of rows.
- May use many functions other than aggregates (e.g. `RANK()`, `LAG()`, or `LEAD()`).
- Groups rows on the row's rank, percentile, etc. as well as its column value.
- Do not collapse rows.
- May use a sliding window frame (which depends on the current row).
-
- GROUP BY --> no usage of DINSTICT clause
- WF --> optional
- GROUP BY --> requires an AGGREGATE Function
- WF --> optional
- GROUP BY --> Ordering invalid
- WF --> ordering valid
- GROUP BY --> low performance
- WF --> high performance

******* WINDOW FUNCTIONS USED WITH "OVER" ******

* AGG() + OVER() AS
* AGG() + OVER(OREDER BY.....) AS
* AGG() + OVER(PARTITION BY.....) AS
* AGG() + OVER(PARTITION BY..... OREDER BY.....) AS

-- Even if we do not write "ORDER BY" after OVER(), by default it operates according to the previous value.

```
select product_id, model_year, sum(list_price) as total_price
from product.product
group by product_id, list_price, model_year
```

	product_id	model_year	total_price
1	1	2018	379.99
2	2	2018	749.99
3	3	2018	999.99
4	4	2018	2899.99
5	5	2018	1320.99
6	6	2018	469.99
7	7	2018	3999.99
8	8	2018	1799.99
9	9	2018	2999.99
10	10	2018	1549.00

****** VALUES THAT CANNOT BE DISPLAYED ON A ROW BASIS WITH GROUP BY CAN BE DISPLAYED WITH WINDOW FUNCTIONS *****

In the table below, unlike GROUP BY, we were able to print the total and annual total sales amounts of that product, the cumulative sales totals, and the sum of each line with the previous and next sales separately.

```
select product_id, model_year, list_price,
sum(list_price) over() as total_price,
sum(list_price) over(partition by model_year) as price_by_year,
sum(list_price) over(partition by model_year order by product_id) as cumulative,
sum(list_price) over(partition by model_year order by product_id rows between 1 preceding and
1 following) as window
from product.product;
```

1.SUM 2.SUM 3.SUM 4.SUM

	product_id	model_year	list_price	total	price_by_year	cumulative	window
1	1	2018	379.99	488109.84	25487.78	379.99	1129.98
2	2	2018	749.99	488109.84	25487.78	1129.98	2129.97
3	3	2018	999.99	488109.84	25487.78	2129.97	4649.97
4	4	2018	2899.99	488109.84	25487.78	5029.96	5220.97
5	5	2018	1320.99	488109.84	25487.78	6350.95	4690.97
6	6	2018	469.99	488109.84	25487.78	6820.94	5790.97
7	7	2018	3999.99	488109.84	25487.78	10820.93	6269.97
8	8	2018	1799.99	488109.84	25487.78	12620.92	8799.97
9	9	2018	2999.99	488109.84	25487.78	15620.91	6348.98

```
-- LEAD, LAG, FIRST_VALUE, LAST_VALUE, NTH_VALUE
-- OREDER BY mandatory, PARTITION BY optional

Select list_price, model_year,
first_value(list_price) over(order by model_year) as first_price_value_of_model_year,
last_value(list_price) over(order by model_year) as last_price_value_of_model_year,
first_value(list_price) over(order by model_year rows between 3 preceding and 3 following)
first_of_3rows,
last_value(list_price) over(order by model_year rows between 3 preceding and 3 following) as
last_of_3rows
From product.product;
```

	list_price	model_year	first_price_value_of_model_year	last_price_value_of_model_year	first_of_3rows	last_of_3rows
172	1499.98	2018	159.99	2197.99	99.99	65.00
173	199.99	2018	159.99	2197.99	299.99	49.93
174	50.50	2018	159.99	2197.99	25.99	2197.99
175	65.00	2018	159.99	2197.99	1499.98	161.99
176	49.93	2018	159.99	2197.99	199.99	53.05
177	2197.99	2018	159.99	2197.99	50.50	66.17
178	161.99	2019	159.99	67.99	65.00	349.95
179	53.05	2019	159.99	67.99	49.93	199.99
180	66.17	2019	159.99	67.99	2197.99	299.99
181	349.95	2019	159.99	67.99	161.99	116.99
182	199.99	2019	159.99	67.99	53.05	66.99
183	299.99	2019	159.99	67.99	66.17	39.99
184	116.99	2019	159.99	67.99	349.95	699.98

```
select model_year, list_price,
first_value(list_price) over(partition by model_year order by list_price) as
first_value_of_model_year,
last_value(list_price) over(partition by model_year order by list_price) as
last_value_of_model_year,
first_value(list_price) over(partition by model_year order by list_price rows between 3
preceding and 3 following) first_of_3rows,
last_value(list_price) over(partition by model_year order by list_price rows between 3
preceding and 3 following) as last_of_3rows
from product.product;
```

	model_year	list_price	first_value_of_model_year	last_value_of_model_year	first_of_3rows	last_of_3rows
172	2018	2799.99	2.00	2799.99	2197.99	3137.95
173	2018	2799.99	2.00	2799.99	2199.98	3989.99
174	2018	2998.00	2.00	2998.00	2498.00	4295.98
175	2018	3137.95	2.00	3137.95	2799.99	4295.98
176	2018	3989.99	2.00	3989.99	2799.99	4295.98
177	2018	4295.98	2.00	4295.98	2998.00	4295.98
178	2019	1.00	1.00	1.00	1.00	2.00
179	2019	1.00	1.00	1.00	1.00	3.00
180	2019	1.00	1.00	1.00	1.00	12.49
181	2019	2.00	1.00	2.00	1.00	19.99
182	2019	3.00	1.00	3.00	1.00	22.08
183	2019	12.49	1.00	12.49	1.00	28.88
184	2019	19.99	1.00	19.99	2.00	29.99
185	2019	22.08	1.00	22.08	3.00	34.58
186	2019	28.88	1.00	28.88	12.49	39.99
187	2019	29.99	1.00	29.99	19.99	41.35
188	2019	34.58	1.00	34.58	22.08	45.99

```
-- LEAD(to next rows), LAG(to previous rows)  default:1
select product_id, list_price,
sum(list_price) over() as total,
sum(list_price) over(order by product_id) as cumulative,
lag(list_price, 2) over(order by product_id) as previous_two_list_price,
lead(list_price, 3) over(order by product_id) as next_three_list_price,
sum(list_price) over(order by product_id rows between 1 preceding and 1 following) as
total_list_price_of_tria
from product.product
order by product_id;
```

```
-- lag(list_price, 2) over(order by product_id) as previous_two_list_price:
```

	product_id	list_price	total	cumulative	previous_two_list_price
1	1	23.99	234294.11	23.99	NULL
2	2	136.99	234294.11	160.98	NULL
3	3	599.00	234294.11	759.98	23.99
4	4	151.99	234294.11	911.97	136.99
5	5	199.99	234294.11	1111.96	599.00
6	6	89.95	234294.11	1201.91	151.99
7	7	59.99	234294.11	1261.90	199.99
8	8	99.99	234294.11	1361.89	89.95
9	9	121.99	234294.11	1483.88	59.99
10	10	174.99	234294.11	1658.87	99.99
11	11	29.99	234294.11	1688.86	121.99
12	12	499.99	234294.11	2188.85	174.99

```
-- lead(list_price, 3) over(order by product_id) as next_three_list_price:
```

	product_id	list_price	total	cumulative	previous_two_list_price	next_three_list_price
1	1	23.99	234294.11	23.99	NULL	151.99
2	2	136.99	234294.11	160.98	NULL	199.99
3	3	599.00	234294.11	759.98	23.99	89.95
4	4	151.99	234294.11	911.97	136.99	59.99
5	5	199.99	234294.11	1111.96	599.00	99.99
6	6	89.95	234294.11	1201.91	151.99	121.99
7	7	59.99	234294.11	1261.90	199.99	174.99
8	8	99.99	234294.11	1361.89	89.95	29.99
9	9	121.99	234294.11	1483.88	59.99	499.99
10	10	174.99	234294.11	1658.87	99.99	99.99
11	11	29.99	234294.11	1688.86	121.99	249.99
12	12	499.99	234294.11	2188.85	174.99	67.99

```
-- sum(list_price) over(order by product_id rows between 1 preceding and 1 following) as
total_list_price_of_tria:
```

	product_id	list_price	total	cumulative	previous_two_list_price	next_three_list_price	total_list_price_of_tria
1	1	23.99	234294.11	23.99	NULL	151.99	160.98
2	2	136.99	234294.11	160.98	NULL	199.99	759.98
3	3	599.00	234294.11	759.98	23.99	89.95	887.98
4	4	151.99	234294.11	911.97	136.99	59.99	950.98
5	5	199.99	234294.11	1111.96	599.00	99.99	441.93
6	6	89.95	234294.11	1201.91	151.99	121.99	349.93
7	7	59.99	234294.11	1261.90	199.99	174.99	249.93
8	8	99.99	234294.11	1361.89	89.95	29.99	281.97
9	9	121.99	234294.11	1483.88	59.99	499.99	396.97
10	10	174.99	234294.11	1658.87	99.99	99.99	326.97
11	11	29.99	234294.11	1688.86	121.99	249.99	704.97
12	12	499.99	234294.11	2188.85	174.99	67.99	629.97

```
-- ROW_NUMBER, RANK, DENSE_RANK, (GIVES SEQUENCE NUMBER)
-- RANK(WHEN GIVING THE INDEX NUMBER OF THE FIRST VALUE TO ALL OF THE SAME VALUES; COUNTER WORKS BEHIND)
-- DENSE_RANK(COUNTER CONTINUES FROM WHERE IT LEFT)
-- Sorts by the column ORDER(ed) BY.
-- OREDER BY mandatory, PARTITION BY optional
```

```
-- RANK and DENSE_RANK will assign the grades the same rank depending on how they fall
compared to the other values.
-- However, RANK will then skip the next available ranking value whereas DENSE_RANK would
still use the next chronological ranking value.

select product_id, model_year, list_price,
row_number() over(order by product_id) as row_1,
row_number() over(partition by model_year order by product_id) as row_2
from product.product
```

	product_id	model_year	list_price	row_1	row_2
169	251	2018	99.99	251	169
170	252	2018	299.99	252	170
171	253	2018	25.99	253	171
172	254	2018	1499.98	254	172
173	255	2018	199.99	255	173
174	256	2018	50.50	256	174
175	257	2018	65.00	257	175
176	258	2018	49.93	258	176
177	259	2018	2197.99	259	177
178	260	2019	161.99	260	1
179	261	2019	53.05	261	2
180	262	2019	66.17	262	3
181	263	2019	349.95	263	4
182	264	2019	199.99	264	5
183	265	2019	299.99	265	6
184	266	2019	116.99	266	7
185	267	2019	66.99	267	8

```
select model_year, list_price,
rank() over(order by list_price) as ranked,
dense_rank() over(order by list_price) as dense_ranked
from product.product
```

	model_year	list_price	ranked	dense_ranked
1	2019	1.00	1	1
2	2019	1.00	1	1
3	2019	1.00	1	1
4	2019	2.00	4	2
5	2018	2.00	4	2
6	2019	3.00	6	3
7	2018	7.99	7	4
8	2020	10.99	8	5
9	2021	11.79	9	6
10	2021	11.99	10	7

```
-- CUME_DIST, PERCENT_RANK, NTILE
-- CUME_DIST(SPLIT BY TOTAL NUMBER OF ROWS)
-- PERCENT_RANK(SPLIT BY TOTAL NUMBER OF ROWS-1)
-- NTILE(INT) : DIVIDES EQUAL GROUPS

select product_id,
cume_dist() over(order by product_id) as cume_disted,
percent_rank() over(order by product_id) as percent_ranked
from product.product
```

product_id	cume_disted	percent_ranked
1	0.00192307692307692	0
2	0.00384615384615385	0.00192678227360308
3	0.00576923076923077	0.00385356454720617
4	0.00769230769230769	0.00578034682080925
5	0.00961538461538462	0.00770712909441233
6	0.0115384615384615	0.00963391136801541
7	0.0134615384615385	0.0115606936416185
8	0.0153846153846154	0.0134874759152216
9	0.0173076923076923	0.0154142581888247
10	0.0192307692307692	0.0173410404624277

```
select product_id, list_price, model_year,
ntile(10) over(partition by model_year order by list_price desc) as splitted_group_no
from product.product
```

	product_id	list_price	model_year	splitted_group_no
504	73	68.99	2021	8
505	43	68.75	2021	9
506	15	67.99	2021	9
507	49	65.89	2021	9
508	7	59.99	2021	9
509	63	54.99	2021	9
510	69	54.99	2021	9
511	62	39.99	2021	9
512	50	34.99	2021	9
513	74	33.79	2021	10
514	71	29.99	2021	10
515	11	29.99	2021	10
516	18	24.99	2021	10
517	22	23.99	2021	10
518	1	23.99	2021	10
519	17	11.99	2021	10
520	30	11.79	2021	10