SQL window functions

Favorite	☐
Archived	☐
Fleeting	☐
Area/Resource	
Project	
Type	
Review Date	
Image	
URL	
Created	@March 22, 2023 9:26 AM
Updated	@March 24, 2023 11:37 AM
Root Area	
Project Area	
Updated (short)	03/24/2023
Pulls	
Resource Pulls	
Project Archived	
URL Base	
🍳 Recipe Divider	🥗🥗🥗 RECIPE BOOK PROPERTIES 🥗🥗🥗
🍳 Recipe Tags	
📚 Book Divider	📚📚📚 BOOK TRACKER PROPERTIES 📚📚📚
📚 Author	
📚 Date Started	
📚 Date Finished	
📚 Book Status	
📚 Rating	

💡 I need to understand SQL window functions for test prep and general SQL literacy.

Steps to understand a window function

Understand the syntax

```
function_name() OVER (PARTITION BY partition_clause ORDER BY order_clause
ROWS BETWEEN frame_start AND frame_end;
```

💡 The syntax specifies the function to be applied, the partitioning of the data, the ordering of the rows within each partition, and the frame of rows over which the function should be computed.

Choose a function

💡 There are many functions to choose from in a window function.

- SUM()
- AVG()
- MIN()
- MAX()
- ROW_NUMBER()
- RANK()
- DENSE_RANK()
- LEAD()
- LAG()
- PERCENT_RANK()
- CUME_DIST()
- NTH_VALUE()

- NTILE()
- FIRST_VALUE()

ROW_NUMBER

⚙️ This function assigns a unique integer value to each row in the result set, based on the order specified in the ORDER_BY clause. It is useful when you need to number rows for grouping or ranking purposes. For example, use ROW_NUMBER to assign a unique ID to each row in a result set for further processing.

- unique_id to each row

RANK

⚙️ This function assigns a rank to each row in the result set, based on the order specified in the ORDER_BY clause. If multiple rows have the same value, they will receive the same rank, and the next rank will be skipped. For example, if three rows have the same value, and have the rank 1, 2, and 2, then the next rank will be 4 not 3. This is useful when you want to identify the top or bottom N rows in a result set, based on some criteria.

- Identify the top or bottom N rows

DENSE_RANK

⚙️ This function assigns a rank to each row in the result set, based on the order specified in the ORDER_BY clause. If multiple rows have the same value, they will receive the same rank, and the next rank will be the next consecutive integer. For example, if three rows have the same value and have the rank, 1, 2, and 2, then the next rank will be 3. This is useful when you want to identify the top or bottom N rows in a result set, based on some criteria but you want to include ties.

- Identify the top or bottom N rows and identify ties.

LEAD

⚙️ This function returns the value of a column in the **next** row of the result set, based on the order specified in the ORDER_BY clause. You can also specify an offset to retrieve the value from a row farther **ahead**. LEAD is useful when you want to compare a value in a row to the **next** or **future** values in the result set. For example to calculate a percentage change or to identify trends.

- Calculate percentage change based on future values

LAG

⚙️ This function returns the value of a column in the **previous** row of the result set, based on the order specified in the ORDER_BY clause. You can also specify an offset to retrieve the value from a row farther **behind**. LEAD is useful when you want to compare a value in a row to the **previous** or **past** values in the result set. For example to calculate a percentage change or to identify trends.

- Calculate percentage change based on previous values

Specify the partitioning

💡 The PARTITION_BY clause specifies how the rows should be partitioned. This means that the function will be computed separately for each partition. For example, if you want to compute a moving average for each customer, you might partition the data by Customer ID:

Specify the ordering

💡 The ORDER_BY clause determines the order in which the rows are processed within each partition. This means that the window function will be computed in the order specified by the ORDER_BY clause.

- If you use a sum function to calculate a running total of sales for each month, you might order the rows by month so that the function is applied in chronological order.

Specify the frame

💡 ROWS_BETWEEN clause specifies the frame of rows over which the function should be computed. This means that the function will be applied to a subset of rows within each partition, based on their position relative to the current row. The function takes two key words ti specify the start and end of the frame.

- You might specify 'ROWS BETWEEN 2 PRECEDING AND CURRENT ROW' clause to compute a moving average based on the last 3 months of data

UNBOUNDED PRECEDING

💡 Includes all rows from the beginning of the partition up to and include the current row.

n PRECEDING

💡 includes n rows before the current row up to and including the current row.

CURRENT ROW

💡 Includes only the current row

n FOLLOWING

💡 includes the current row up to and including n rows after the current row

UNBOUNDED FOLLOWING

💡 Includes all rows from the current row up to the end of the partition.

Window function visualized

Review

💡 Unlike normal aggregation functions with grouping, which only return one row per group, window functions return all rows, each having their calculated value based on their defined window.

💡 Window function use OVER keyword and it’s argument to define the calculation window

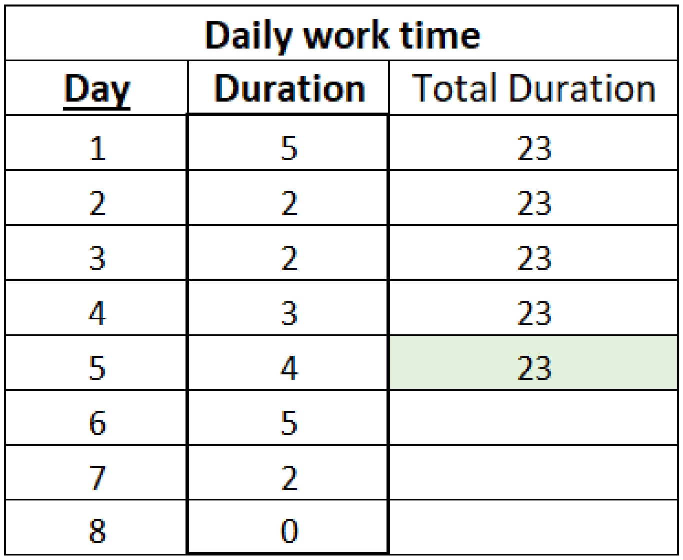
Three parts to a window function:

1. Grouping: This defines the group that each row belongs to (PARTITION BY)
2. Order: This sort values within each group and make the window expands incrementally within each group (ORDER BY)
3. Range: This is use to further define the window size, within each group (ROWS or RANGE)

OVER

```
SELECT
  day,
  duration,
  SUM(duration) OVER () AS total_duration
FROM
  work;
```

⚙️ Defining a simple SUM window function OVER duration with no args. Meaning every row’s window is the same and it is the whole table.

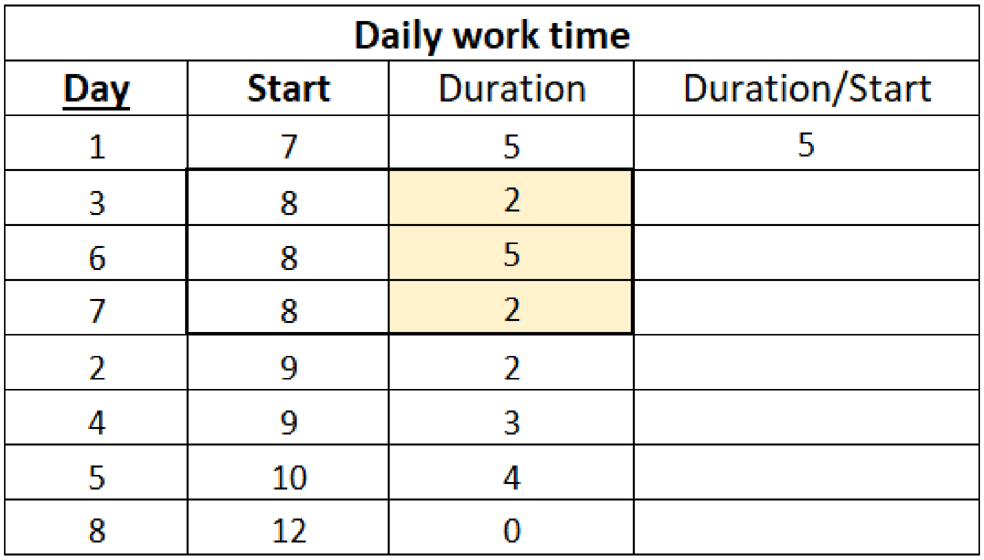


Daily work time		
Day	Duration	Total Duration
1	5	23
2	2	23
3	2	23
4	3	23
5	4	23
6	5	
7	2	
8	0	

OVER (PARTITION BY..)

```
SELECT
  day,
  duration,
  SUM(duration) OVER (PARTITION BY start) AS total_duration
FROM
  work;
```

This query sums duration of rows PARTITION by start_time. This means that every start value will share the same sum of the associated duration values that the rest of their group. Rows are re-ordered based on their grouping when you use Partition.



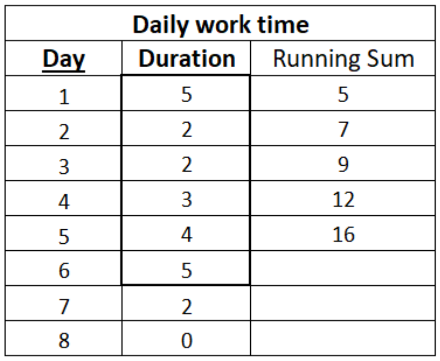
Daily work time			
Day	Start	Duration	Duration/Start
1	7	5	5
3	8	2	
6	8	5	
7	8	2	
2	9	2	
4	9	3	
5	10	4	
8	12	0	

- The start group 8, will apply the sum function to all duration values associated to the start rows with value 8. Meaning duration/start will be 9.

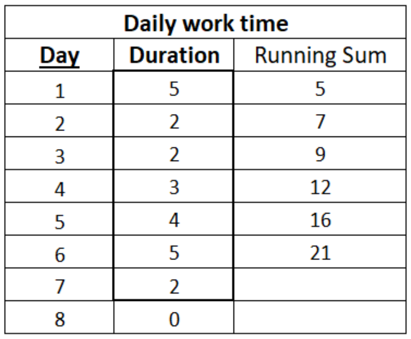
OVER(ORDER BY ..)

```
SELECT
  day,
  duration,
  SUM(duration) OVER (ORDER BY day) AS sum_duration_order_by_day
FROM
  work;
```

ORDER BY will order rows within their respective group. and ORDER BY will force the window to expand gradually (one row at a time) within each group.



Daily work time		
Day	Duration	Running Sum
1	5	5
2	2	7
3	2	9
4	3	12
5	4	16
6	5	
7	2	
8	0	



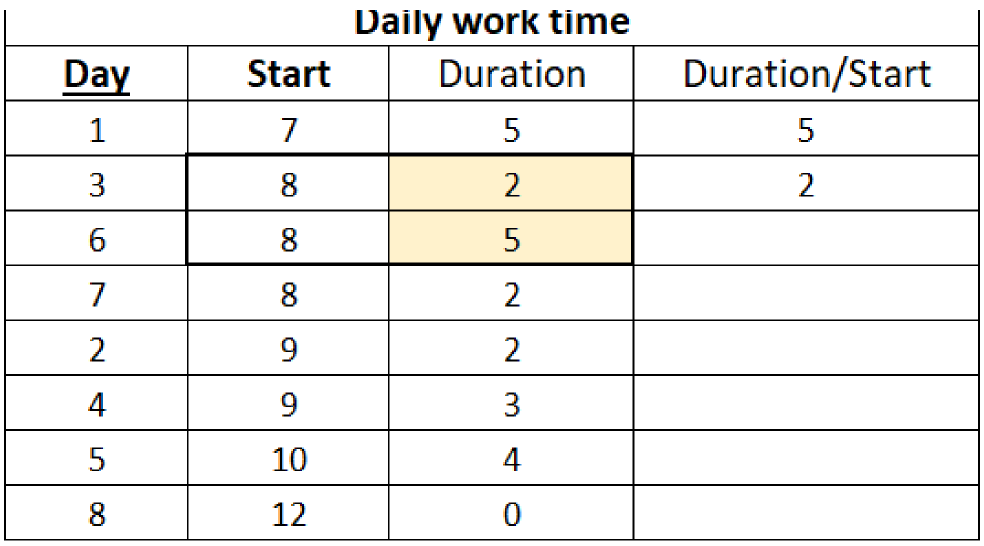
Daily work time		
Day	Duration	Running Sum
1	5	5
2	2	7
3	2	9
4	3	12
5	4	16
6	5	21
7	2	
8	0	

- Since we did not use a partition by the grouping applies to all rows, and since we used an order by the window function expands by one for each row, giving us a running sum.

OVER(PARTITION BY..SORT BY..)

```
SELECT
    day,
    start,
    duration,
    SUM(duration) OVER(PARTITION BY start ORDER BY day)
      AS sum_duration_sort_by_day
FROM
    work;
```

This query uses the partition by start and the order by day. Meaning that it will group every window by start value, and instead of returning the sum of all start_values such as the example above. It will reset the incremental window sum back to 1 for each new group.



Daily work time

Day	Start	Duration	Duration/Start
1	7	5	5
3	8	2	2
6	8	5	
7	8	2	
2	9	2	
4	9	3	
5	10	4	
8	12	0	

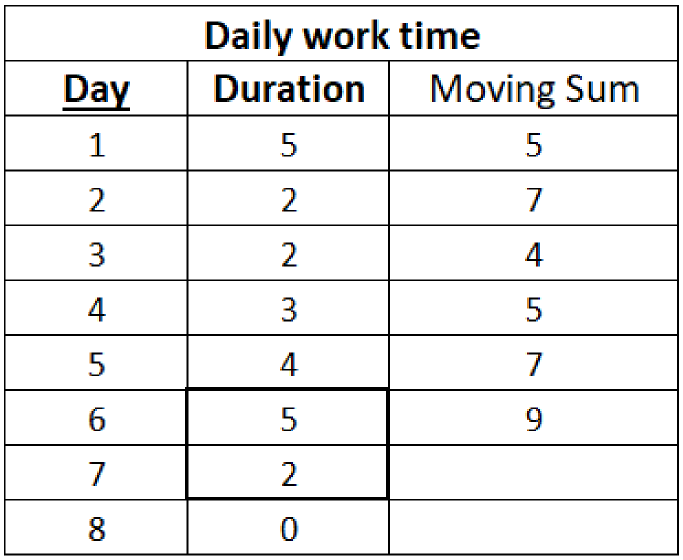
- The first observation is that the PARTITION BY takes precedent so while it is ordered by day it is only ordered by day in the context of the groups.
- Additionally the partition by group is ordered as the default.
- The window resets back to 1 for each new group and expands the window by the ordered day for each same value in the group

Moving aggregation with Rows

ROWS and RANGE provide us with more fine grained control over window size.

```
SELECT
    day,
    duration,
    SUM(duration) OVER(ORDER BY day ROWS BETWEEN 1 PRECEDING AND CURRENT ROW ) AS moving_sum
FROM
    work;
```

This query will show use the amount of work hours done for the last two days. As we define the sum over a window of size 2. Result in a moving summation with a window size 2.



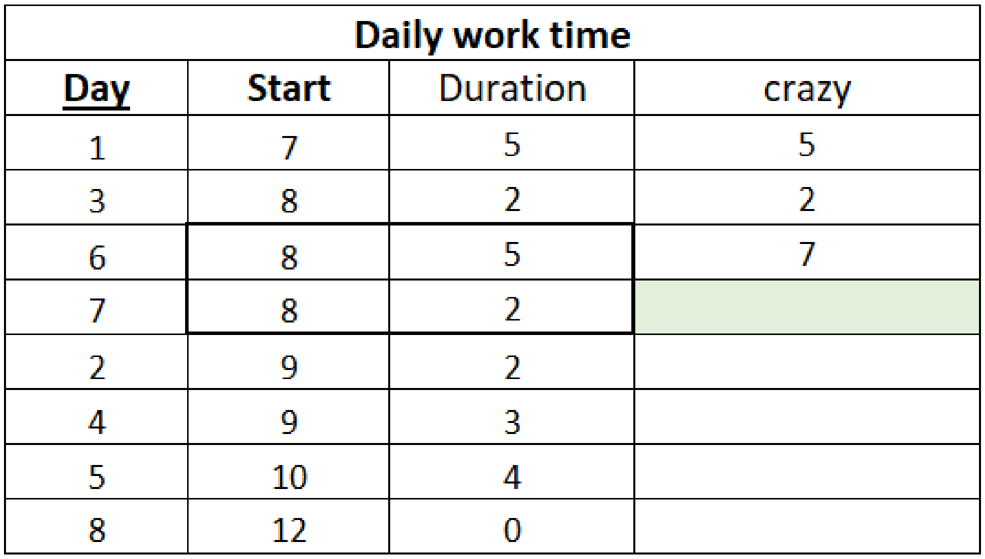
Daily work time		
Day	Duration	Moving Sum
1	5	5
2	2	7
3	2	4
4	3	5
5	4	7
6	5	9
7	2	
8	0	

- Since we are not partitioning there is no grouping and we are ordering by days. However we have specified a range of between preceding row 1 and the current row, so it will not be a running sum, but a sum of the two rows at a time.

All together

```
SELECT

day,

start,

duration,

SUM(duration) OVER(PARTITION BY start ORDER BY day ROWS BETWEEN 1 PRECEDING AND CURRENT ROW) AS crazy

FROM

work;
```

This code will show you the running average of duration grouped by start but limiting the window to 2. SO it means that the start time of 8, will not encompass all three but instead will show the sum of the current and preceding value.



Daily work time

Day	**Start**	Duration	crazy
1	7	5	5
3	8	2	2
6	8	5	7
7	8	2	
2	9	2	
4	9	3	
5	10	4	
8	12	0	

- Start grouping ordering takes precedent
- Day is ordered within subsequent group
- the window function is limited to the current and preceding row.