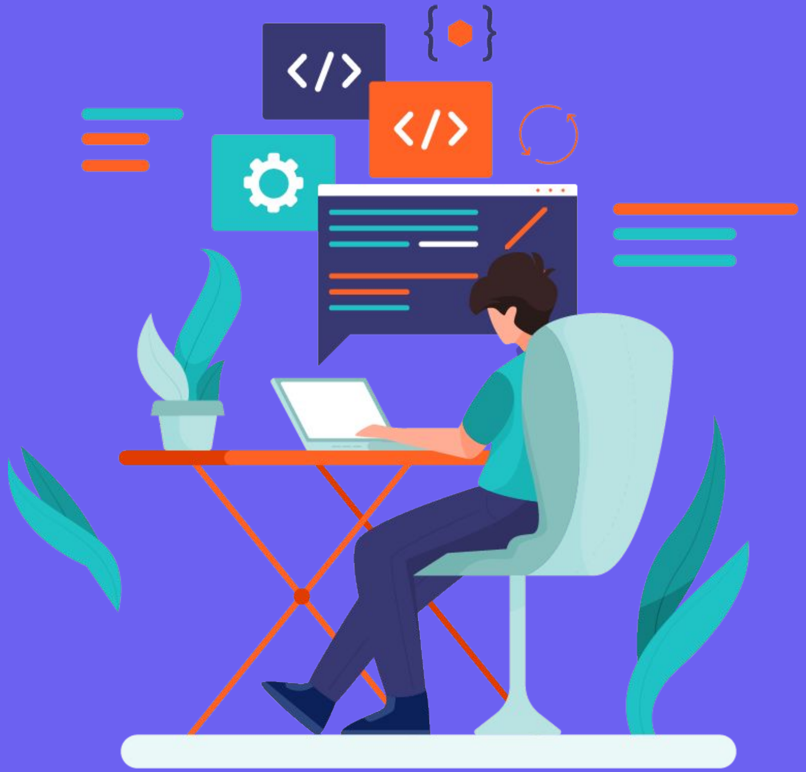
Window Functions - I

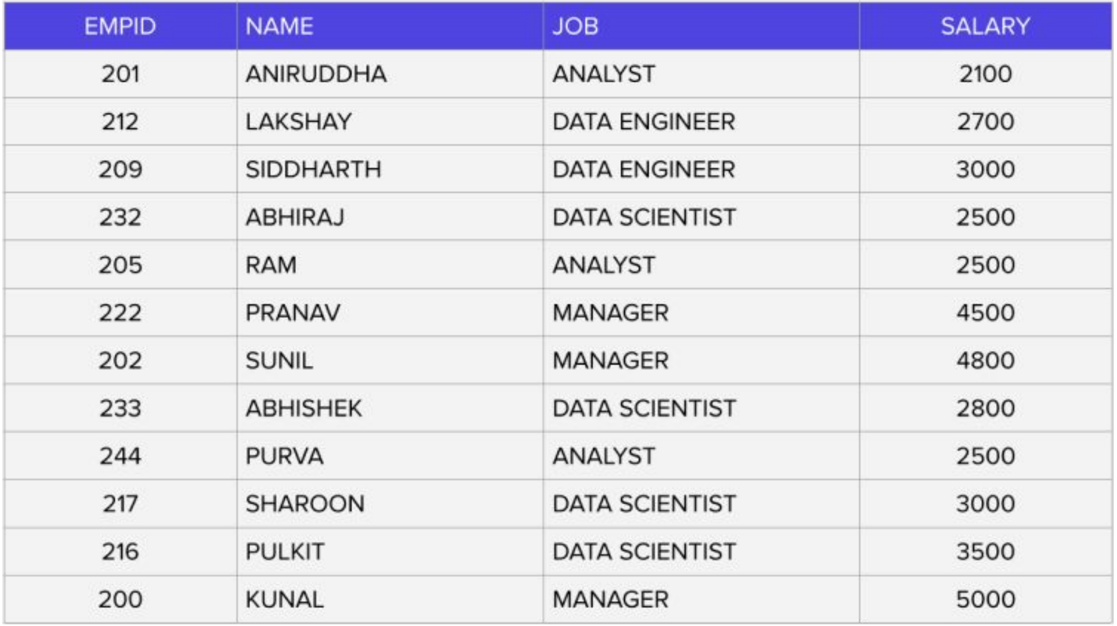


Relevel

by Unacademy

Why do we need window functions?

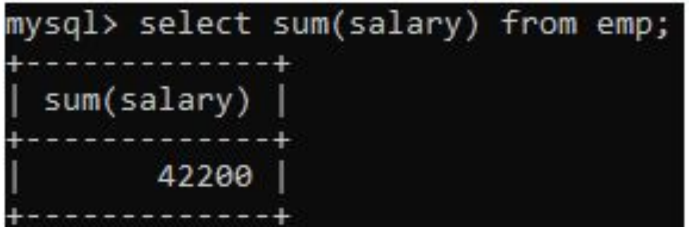
- Let’s assume we have a dataset employee_details:



EMPID	NAME	JOB	SALARY
201	ANIRUDDHA	ANALYST	2100
212	LAKSHAY	DATA ENGINEER	2700
209	SIDDHARTH	DATA ENGINEER	3000
232	ABHIRAJ	DATA SCIENTIST	2500
205	RAM	ANALYST	2500
222	PRANAV	MANAGER	4500
202	SUNIL	MANAGER	4800
233	ABHISHEK	DATA SCIENTIST	2800
244	PURVA	ANALYST	2500
217	SHAROON	DATA SCIENTIST	3000
216	PULKIT	DATA SCIENTIST	3500
200	KUNAL	MANAGER	5000

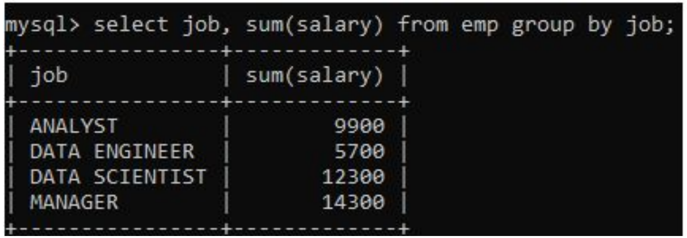
Why do we need Window Functions?

Suppose we want the sum of the total salary of all the employees in the company. We will use the Aggregate function SUM on the column salary and will get desired output.



```
mysql> select sum(salary) from emp;
+-------------+
| sum(salary) |
+-------------+
|       42200 |
+-------------+
```

Suppose we have to determine the total salary of employees per job category. We will again use the Aggregate function SUM on the column salary along with the GROUP BY clause on Job column to get the desired output.

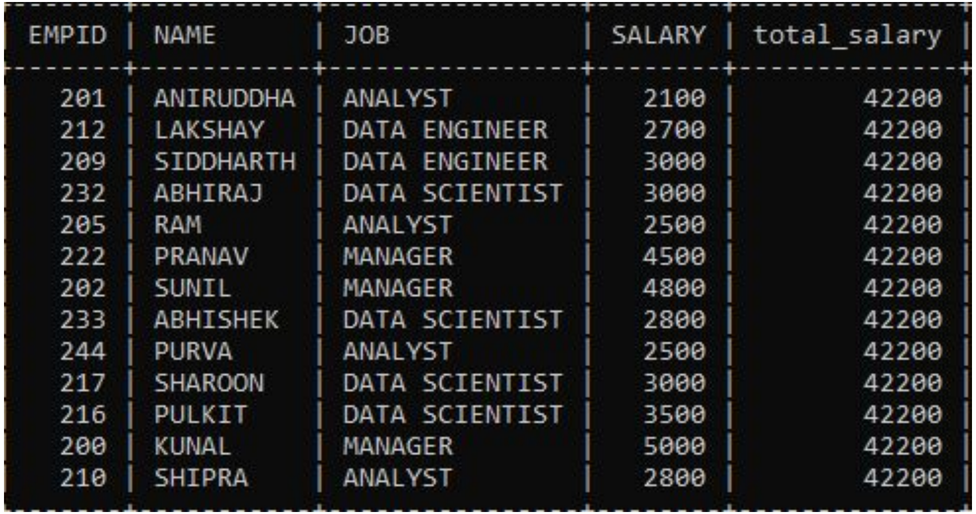


```
mysql> select job, sum(salary) from emp group by job;
+----------------+-------------+
| job            | sum(salary) |
+----------------+-------------+
| ANALYST        |        9900 |
| DATA ENGINEER  |        5700 |
| DATA SCIENTIST |       12300 |
| MANAGER        |       14300 |
+----------------+-------------+
```

Why do we need Window Functions?

However, let’s think of a scenario where we want to display the total salary along with every row value.

Desired Output

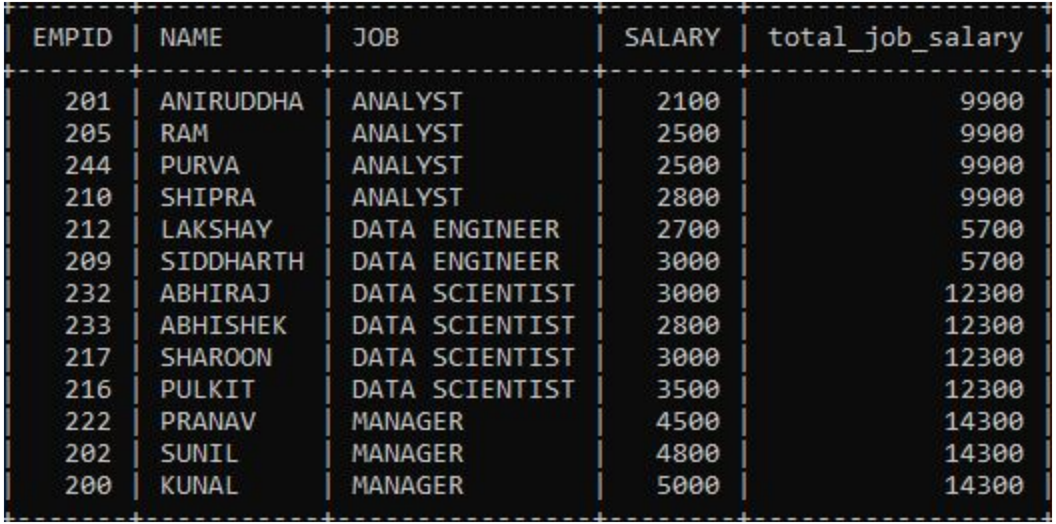


```
+-------+-----------+----------------+--------+--------------+
| EMPID | NAME      | JOB            | SALARY | total_salary |
+-------+-----------+----------------+--------+--------------+
|   201 | ANIRUDDHA | ANALYST        |   2100 |        42200 |
|   212 | LAKSHAY   | DATA ENGINEER  |   2700 |        42200 |
|   209 | SIDDHARTH | DATA ENGINEER  |   3000 |        42200 |
|   232 | ABHIRAJ   | DATA SCIENTIST |   3000 |        42200 |
|   205 | RAM       | ANALYST        |   2500 |        42200 |
|   222 | PRANAV    | MANAGER        |   4500 |        42200 |
|   202 | SUNIL     | MANAGER        |   4800 |        42200 |
|   233 | ABHISHEK  | DATA SCIENTIST |   2800 |        42200 |
|   244 | PURVA     | ANALYST        |   2500 |        42200 |
|   217 | SHAROON   | DATA SCIENTIST |   3000 |        42200 |
|   216 | PULKIT    | DATA SCIENTIST |   3500 |        42200 |
|   200 | KUNAL     | MANAGER        |   5000 |        42200 |
|   210 | SHIPRA    | ANALYST        |   2800 |        42200 |
+-------+-----------+----------------+--------+--------------+
```

Why do we need Window Functions?

Or if we want to display the total salary and the total salary per job category along with every row value. Arrange the salary in decreasing order within each job category.

Desired Output



```
+-------+-----------+----------------+--------+------------------+
| EMPID | NAME      | JOB            | SALARY | total_job_salary |
+-------+-----------+----------------+--------+------------------+
|   201 | ANIRUDDHA | ANALYST        |   2100 |             9900 |
|   205 | RAM       | ANALYST        |   2500 |             9900 |
|   244 | PURVA     | ANALYST        |   2500 |             9900 |
|   210 | SHIPRA    | ANALYST        |   2800 |             9900 |
|   212 | LAKSHAY   | DATA ENGINEER  |   2700 |             5700 |
|   209 | SIDDHARTH | DATA ENGINEER  |   3000 |             5700 |
|   232 | ABHIRAJ   | DATA SCIENTIST |   3000 |            12300 |
|   233 | ABHISHEK  | DATA SCIENTIST |   2800 |            12300 |
|   217 | SHAROON   | DATA SCIENTIST |   3000 |            12300 |
|   216 | PULKIT    | DATA SCIENTIST |   3500 |            12300 |
|   222 | PRANAV    | MANAGER        |   4500 |            14300 |
|   202 | SUNIL     | MANAGER        |   4800 |            14300 |
|   200 | KUNAL     | MANAGER        |   5000 |            14300 |
+-------+-----------+----------------+--------+------------------+
```

Why do we need Window Functions?

None of the previous two scenarios could be solved by aggregate functions alone. It will require writing self-join subqueries along with the aggregator function.

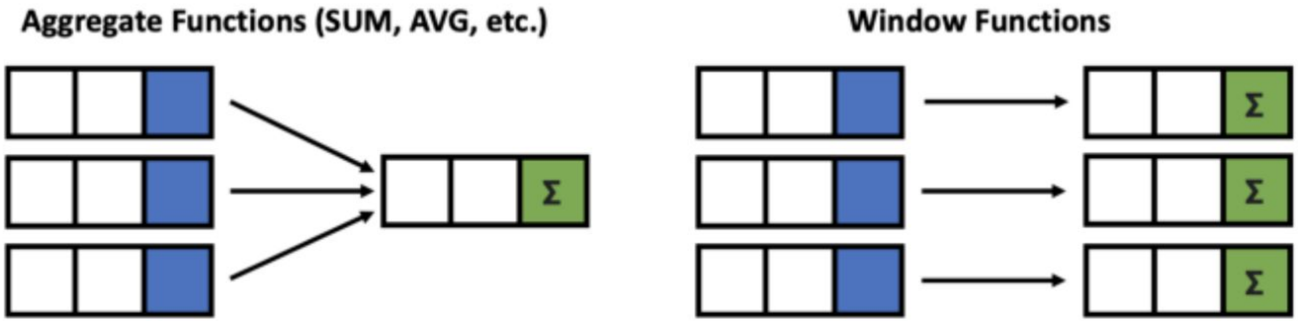
However, this could be easily addressed by an advance functionality in SQL – Windows function.

What is a Window Function?

Window functions conduct calculations on a group of linked rows. Windowing functions, unlike aggregate functions, do not combine the results of multiple rows into a single value. Instead, each row retains its original identity, and the calculated result is returned for each row.

These are similar to aggregate functions, but there is one key distinction. When applying aggregate functions with the GROUP BY clause, the individual rows are "lost." This is not the case when we use SQL window functions: we can obtain a result set that includes some of the properties of a single row and the results of the window function.

What is a Window Function?



Notice how the GROUP BY aggregation on the left-hand side of the picture groups the three rows into one single row. The window function on the right-hand side of the picture is able to output each row with an aggregation value. This may save you from having to do a join after the GROUP BY.

Why Use a Window Function?

Because the rows are not compressed together, window functions allow you to work with both aggregate and non-aggregate values simultaneously.

Window features are also easy to use and understand. That is, they can lessen the complexity of your queries, making them easier to maintain in the long run.

They can also assist with performance difficulties. You can, for example, use a window function instead of a self-join or cross-join.

Window Function Syntax

```
SELECT

<column_1>, <column_2>,

<window_function>(expression)OVER

(PARTITION BY<partition_list> ORDER BY<order_list> ROWS frame_clause

FROM

<table_name>
```

Window Function Syntax

Key definitions in window function are as follows:

- **Window function** is the name of the window function we wish to apply, such as sum, average, or row number (we'll learn more about these later).
- **Expression** is the column's name on which the window function should be applied. Depending on the window function used, this may or may not be required.
- **OVER** simply indicates that this is a window function.
- **PARTITION BY** partitions the rows, allowing us to define which rows to utilise to compute the window function.
- **Partition list** is the name of the column(s) by which we want to partition.
- **ORDER BY** is used to sort the rows within each partition. This is optional and should not be specified.
- **Order list** is the name of the column(s) to be ordered.
- If we want to further limit the rows within our partition, we can utilise **ROWS.** This is optional and is rarely used.
- The **frame clause** specifies how much we should offset from our current row.

Detailed Explanation of OVER Clause

Window functions are distinguished from other analytical and reporting functions by the OVER() clause (window specification).

The **OVER** clause represents a window of rows to which a window function is applied. It can be used with aggregate and non-aggregate functions (such as SUM and COUNT) (discussed later in the session).

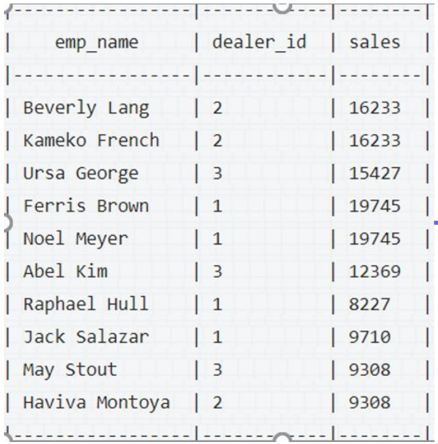
The OVER() clause can perform the following functions:

- Defines window partitions used to build groups of rows (PARTITION BY clause).
- Rows within a partition are ordered (ORDER BY clause).
- ROWS or RANGE clause

OVER Clause – An Example

The following query uses the AVG() window function to calculate the average salary of the employees using the Over clause.

q1_sales

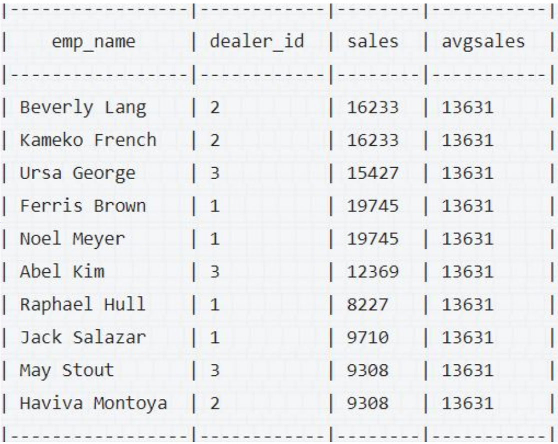


```
|-----------------|------------|--------|
|    emp_name     | dealer_id  | sales  |
|-----------------|------------|--------|
| Beverly Lang    | 2          | 16233  |
| Kameko French   | 2          | 16233  |
| Ursa George     | 3          | 15427  |
| Ferris Brown    | 1          | 19745  |
| Noel Meyer      | 1          | 19745  |
| Abel Kim        | 3          | 12369  |
| Raphael Hull    | 1          | 8227   |
| Jack Salazar    | 1          | 9710   |
| May Stout       | 3          | 9308   |
| Haviva Montoya  | 2          | 9308   |
|-----------------|------------|--------|
```

Query

```
select
emp_name, dealer_id, sales, avg(sales)
over() as Avgsales
FROM q1_sales;
```

q1_sales

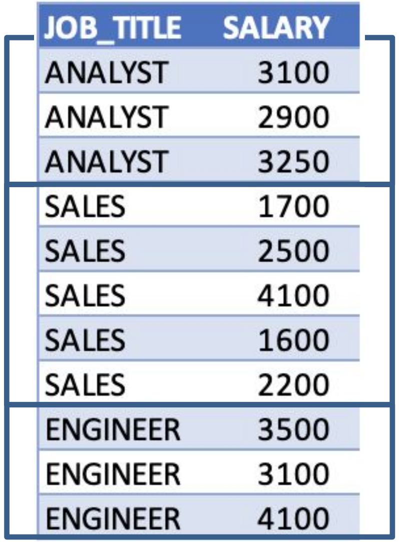


```
|-----------------|------------|--------|-----------|
|    emp_name     | dealer_id  | sales  | avgsales  |
|-----------------|------------|--------|-----------|
| Beverly Lang    | 2          | 16233  | 13631     |
| Kameko French   | 2          | 16233  | 13631     |
| Ursa George     | 3          | 15427  | 13631     |
| Ferris Brown    | 1          | 19745  | 13631     |
| Noel Meyer      | 1          | 19745  | 13631     |
| Abel Kim        | 3          | 12369  | 13631     |
| Raphael Hull    | 1          | 8227   | 13631     |
| Jack Salazar    | 1          | 9710   | 13631     |
| May Stout       | 3          | 9308   | 13631     |
| Haviva Montoya  | 2          | 9308   | 13631     |
|-----------------|------------|--------|-----------|
```

Detailed Explanation of PARTITION BY Clause

- The **PARTITION BY** clause is used in conjunction with the OVER clause. It breaks up the rows into different partitions. These partitions are then acted upon by the window function.

- In the example below, one can see the table could be partitioned into three widows on the job title:

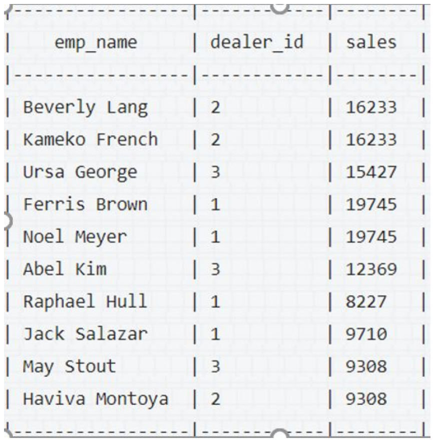


JOB_TITLE	SALARY
ANALYST	3100
ANALYST	2900
ANALYST	3250
SALES	1700
SALES	2500
SALES	4100
SALES	1600
SALES	2200
ENGINEER	3500
ENGINEER	3100
ENGINEER	4100

PARTITION BY Clause – An Example

The following query uses the AVG() window function with the **PARTITION BY** clause to determine the average car sales for each dealer.

q1_sales

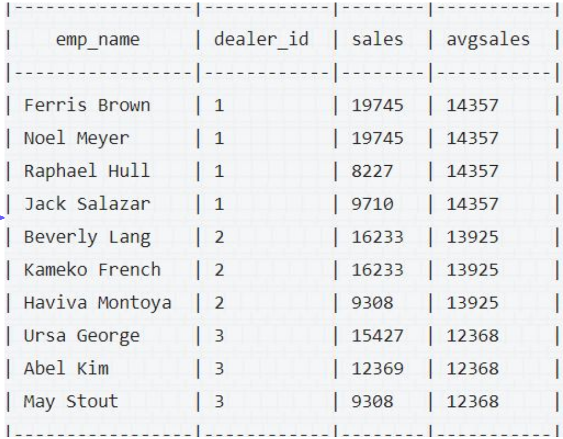


```
|----------------|-----------|--------|
|    emp_name    | dealer_id | sales  |
|----------------|-----------|--------|
| Beverly Lang   | 2         | 16233  |
| Kameko French  | 2         | 16233  |
| Ursa George    | 3         | 15427  |
| Ferris Brown   | 1         | 19745  |
| Noel Meyer     | 1         | 19745  |
| Abel Kim       | 3         | 12369  |
| Raphael Hull   | 1         | 8227   |
| Jack Salazar   | 1         | 9710   |
| May Stout      | 3         | 9308   |
| Haviva Montoya | 2         | 9308   |
|----------------|-----------|--------|
```

Query

select emp_name, dealer_id, sales, avg(sales) **over** (**partition by** dealer_id) as avgsales from q1_sales;

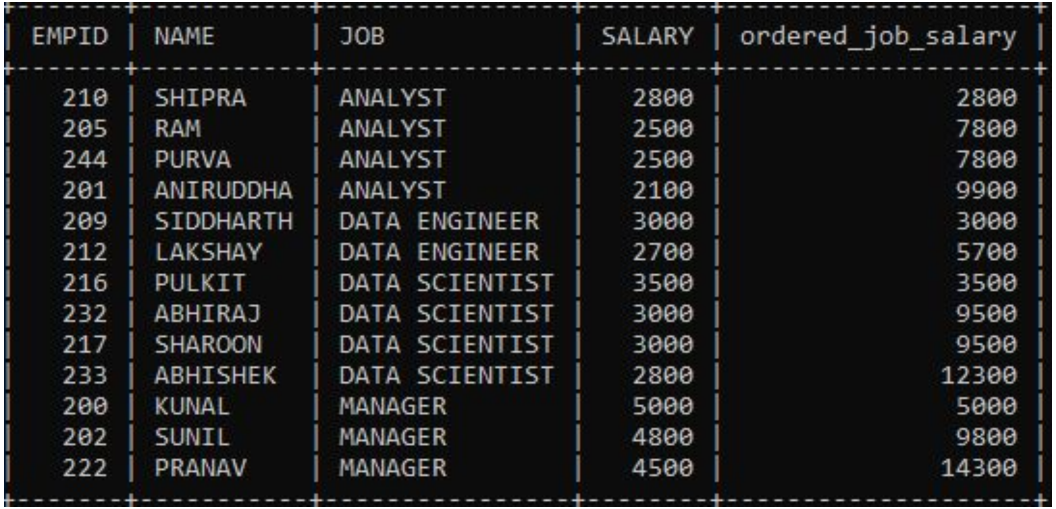
q1_sales



```
|----------------|-----------|--------|-----------|
|    emp_name    | dealer_id | sales  | avgsales  |
|----------------|-----------|--------|-----------|
| Ferris Brown   | 1         | 19745  | 14357     |
| Noel Meyer     | 1         | 19745  | 14357     |
| Raphael Hull   | 1         | 8227   | 14357     |
| Jack Salazar   | 1         | 9710   | 14357     |
| Beverly Lang   | 2         | 16233  | 13925     |
| Kameko French  | 2         | 16233  | 13925     |
| Haviva Montoya | 2         | 9308   | 13925     |
| Ursa George    | 3         | 15427  | 12368     |
| Abel Kim       | 3         | 12369  | 12368     |
| May Stout      | 3         | 9308   | 12368     |
|----------------|-----------|--------|-----------|
```

Detailed Explanation of ORDER BY Clause

- The **ORDER BY** clause is used in conjunction with the OVER clause. It is used to arrange the rows within each partition.
- The below example shows how the salary column is arranged for each partition of the job column:

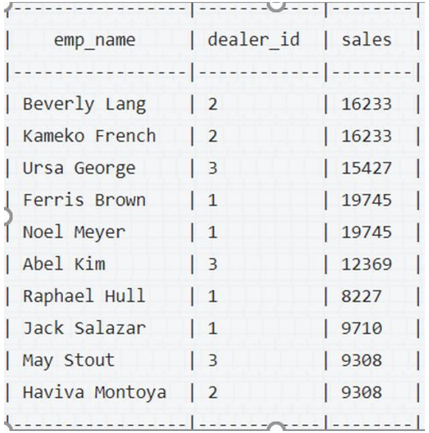


```
+-------+-----------+----------------+--------+--------------------+
| EMPID | NAME      | JOB            | SALARY | ordered_job_salary |
+-------+-----------+----------------+--------+--------------------+
|   210 | SHIPRA    | ANALYST        |   2800 |               2800 |
|   205 | RAM       | ANALYST        |   2500 |               7800 |
|   244 | PURVA     | ANALYST        |   2500 |               7800 |
|   201 | ANIRUDDHA | ANALYST        |   2100 |               9900 |
|   209 | SIDDHARTH | DATA ENGINEER  |   3000 |               3000 |
|   212 | LAKSHAY   | DATA ENGINEER  |   2700 |               5700 |
|   216 | PULKIT    | DATA SCIENTIST |   3500 |               3500 |
|   232 | ABHIRAJ   | DATA SCIENTIST |   3000 |               9500 |
|   217 | SHAROON   | DATA SCIENTIST |   3000 |               9500 |
|   233 | ABHISHEK  | DATA SCIENTIST |   2800 |              12300 |
|   200 | KUNAL     | MANAGER        |   5000 |               5000 |
|   202 | SUNIL     | MANAGER        |   4800 |               9800 |
|   222 | PRANAV    | MANAGER        |   4500 |              14300 |
+-------+-----------+----------------+--------+--------------------+
```

ORDER BY Clause – An Example

The following query uses the AVG() window functions to determine the average car sales for each dealer and assign a row number to each row in a partition:

q1_sales

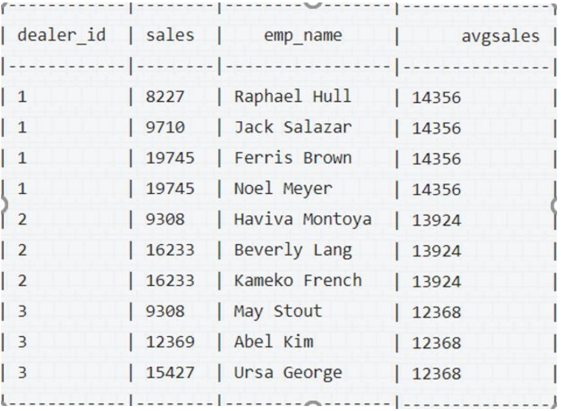


```
|-----------------|------------|--------|
|    emp_name     | dealer_id  | sales  |
|-----------------|------------|--------|
| Beverly Lang    | 2          | 16233  |
| Kameko French   | 2          | 16233  |
| Ursa George     | 3          | 15427  |
| Ferris Brown    | 1          | 19745  |
| Noel Meyer      | 1          | 19745  |
| Abel Kim        | 3          | 12369  |
| Raphael Hull    | 1          | 8227   |
| Jack Salazar    | 1          | 9710   |
| May Stout       | 3          | 9308   |
| Haviva Montoya  | 2          | 9308   |
|-----------------|------------|--------|
```

Query

select emp_name, dealer_id, sales, avg(sales) **over** (**partition by** dealer_id order by sales) as avgsales from q1_sales;

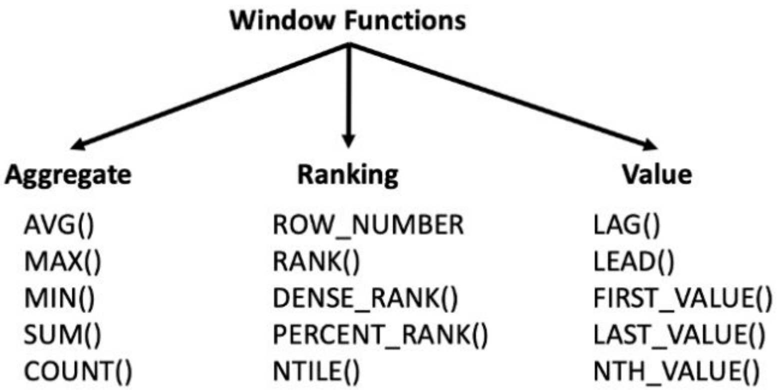
q1_sales



```
|------------|--------|-----------------|---------------|
| dealer_id  | sales  |    emp_name     |      avgsales |
|------------|--------|-----------------|---------------|
| 1          | 8227   | Raphael Hull    | 14356         |
| 1          | 9710   | Jack Salazar    | 14356         |
| 1          | 19745  | Ferris Brown    | 14356         |
| 1          | 19745  | Noel Meyer      | 14356         |
| 2          | 9308   | Haviva Montoya  | 13924         |
| 2          | 16233  | Beverly Lang    | 13924         |
| 2          | 16233  | Kameko French   | 13924         |
| 3          | 9308   | May Stout       | 12368         |
| 3          | 12369  | Abel Kim        | 12368         |
| 3          | 15427  | Ursa George     | 12368         |
|------------|--------|-----------------|---------------|
```

Types of Window Functions

- There are three main types of window functions available to use: aggregate, ranking, and value functions.



Window Functions

Aggregate	Ranking	Value
AVG()	ROW_NUMBER	LAG()
MAX()	RANK()	LEAD()
MIN()	DENSE_RANK()	FIRST_VALUE()
SUM()	PERCENT_RANK()	LAST_VALUE()
COUNT()	NTILE()	NTH_VALUE()

Types of Window Functions – Quick Overview

- **Aggregate functions:** These can be used to calculate aggregations such as average, the total number of rows, maximum or minimum values, or total sum inside each window or partition.

- **Ranking functions:** These are useful for ranking rows within a partition.

- **Value functions:** They allow you to compare values from previous or subsequent rows inside the partition and the first or last value within the partition.

Aggregate Window Functions – Syntax

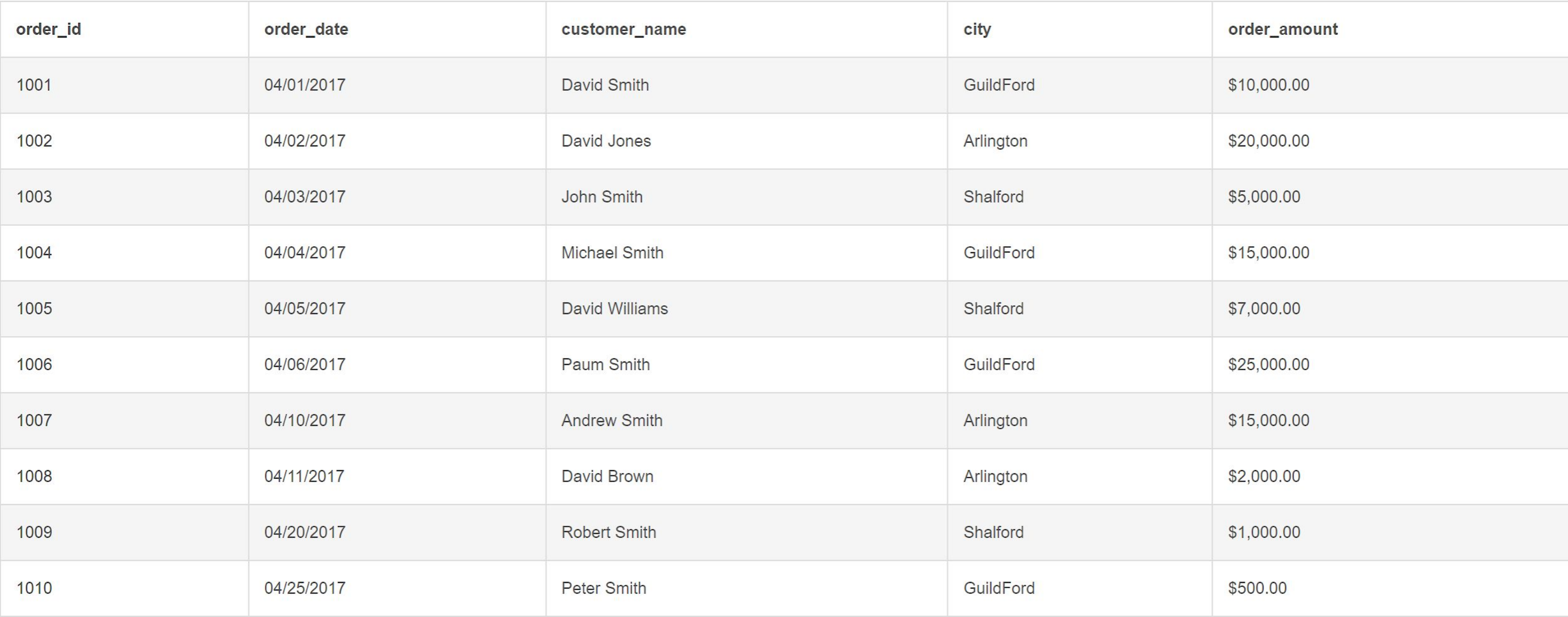
```
SELECT

<column_1>,

<column_2>,

<window_function>(expression)OVER(partition by <partition_list> ORDER BY <order_list>)

FROM

<table_name>
```

Types of Aggregate functions

- **COUNT():** It counts the number of input rows within a window.
- **SUM():** It gives the sum of all the values of expressions for all the rows within a window.
- **AVG():** It gives the average of all the values of expressions for all the rows within a window.
- **MIN():** It gives the minimum of all the values of expressions for all the rows within a window.
- **MAX():** It gives the maximum of all the values of expressions for all the rows within a window.

Understanding Aggregate Functions with an example

We will use the below mentioned table ‘orders’ for understanding aggregate window functions:



order_id	order_date	customer_name	city	order_amount
1001	04/01/2017	David Smith	GuildFord	$10,000.00
1002	04/02/2017	David Jones	Arlington	$20,000.00
1003	04/03/2017	John Smith	Shalford	$5,000.00
1004	04/04/2017	Michael Smith	GuildFord	$15,000.00
1005	04/05/2017	David Williams	Shalford	$7,000.00
1006	04/06/2017	Paum Smith	GuildFord	$25,000.00
1007	04/10/2017	Andrew Smith	Arlington	$15,000.00
1008	04/11/2017	David Brown	Arlington	$2,000.00
1009	04/20/2017	Robert Smith	Shalford	$1,000.00
1010	04/25/2017	Peter Smith	GuildFord	$500.00

Sum() Window function

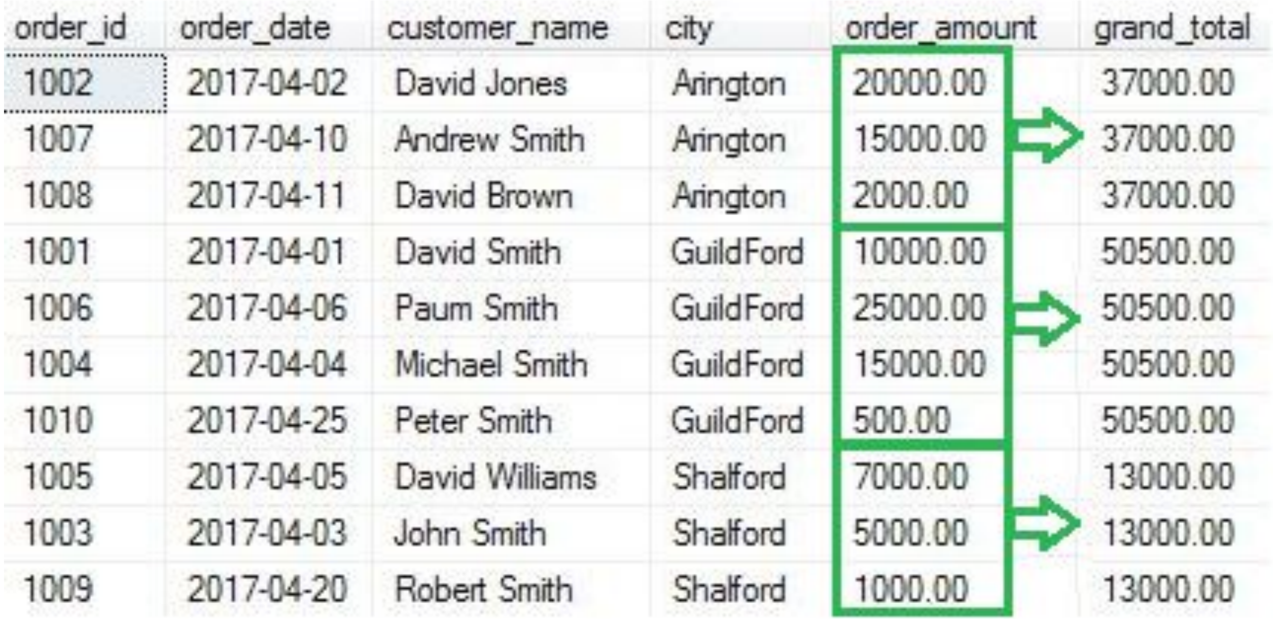
In this example, we will display the total_order amount for a city for each row:

Query

```
SELECT order_id, order_date, customer_name, city, order_amount

SUM(order_amount) OVER(PARTITION BY city) as grand_total

FROM orders
```

Sum() Window function

Output



order_id	order_date	customer_name	city	order_amount	grand_total
1002	2017-04-02	David Jones	Arington	20000.00	37000.00
1007	2017-04-10	Andrew Smith	Arington	15000.00	37000.00
1008	2017-04-11	David Brown	Arington	2000.00	37000.00
1001	2017-04-01	David Smith	GuildFord	10000.00	50500.00
1006	2017-04-06	Paum Smith	GuildFord	25000.00	50500.00
1004	2017-04-04	Michael Smith	GuildFord	15000.00	50500.00
1010	2017-04-25	Peter Smith	GuildFord	500.00	50500.00
1005	2017-04-05	David Williams	Shalford	7000.00	13000.00
1003	2017-04-03	John Smith	Shalford	5000.00	13000.00
1009	2017-04-20	Robert Smith	Shalford	1000.00	13000.00

Practice Question

Instructions for practice questions

- Log into https://mode.com/
- Create a new report
- Access database tutorial.dc_bikeshare_q1_2012

Practice Question

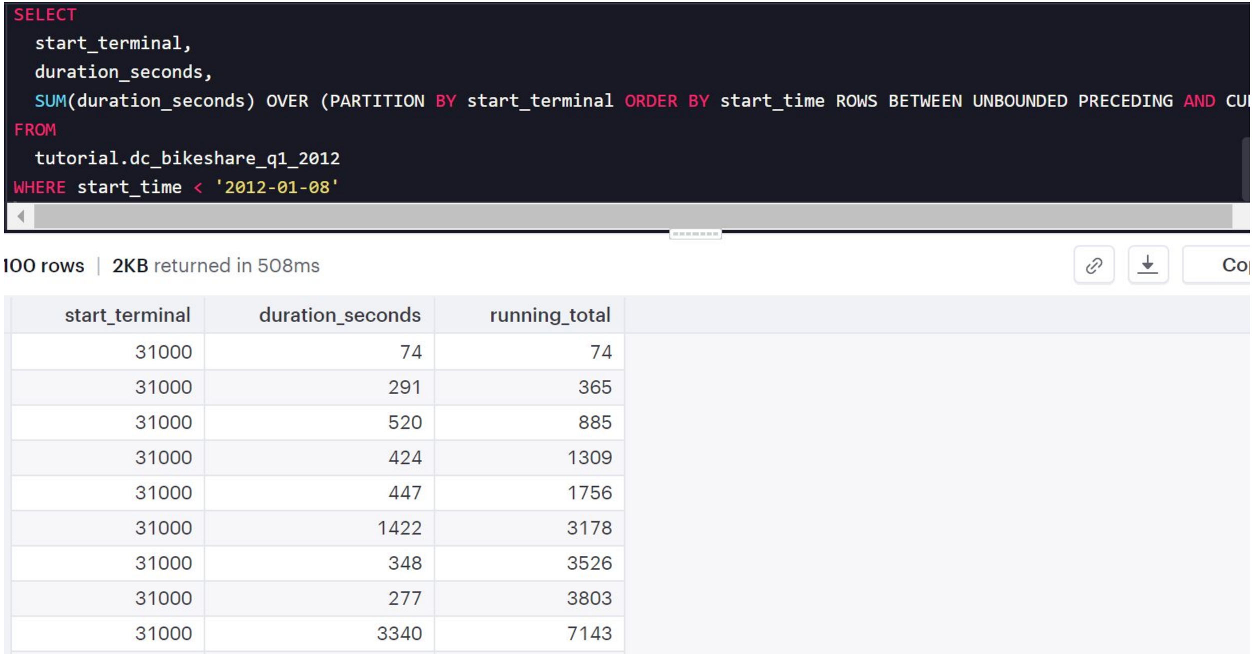
Display the running total (cumulative) of duration_seconds for all the rides per terminal against each row. Partition the data at start_terminal. Consider the data where start_time is before ‘2012-01-08’.

Solution

```sql
SELECT
        start_terminal,
        duration_seconds,
        SUM(duration_seconds) OVER (PARTITION BY start_terminal ORDER BY start_time ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS running_total
FROM
        tutorial.dc_bikeshare_q1_2012
WHERE start_time < '2012-01-08'
```

Solution

Output



```
SELECT
  start_terminal,
  duration_seconds,
  SUM(duration_seconds) OVER (PARTITION BY start_terminal ORDER BY start_time ROWS BETWEEN UNBOUNDED PRECEDING AND CU
FROM
  tutorial.dc_bikeshare_q1_2012
WHERE start_time < '2012-01-08'
```

100 rows | 2KB returned in 508ms

start_terminal	duration_seconds	running_total
31000	74	74
31000	291	365
31000	520	885
31000	424	1309
31000	447	1756
31000	1422	3178
31000	348	3526
31000	277	3803
31000	3340	7143

AVG() Window function

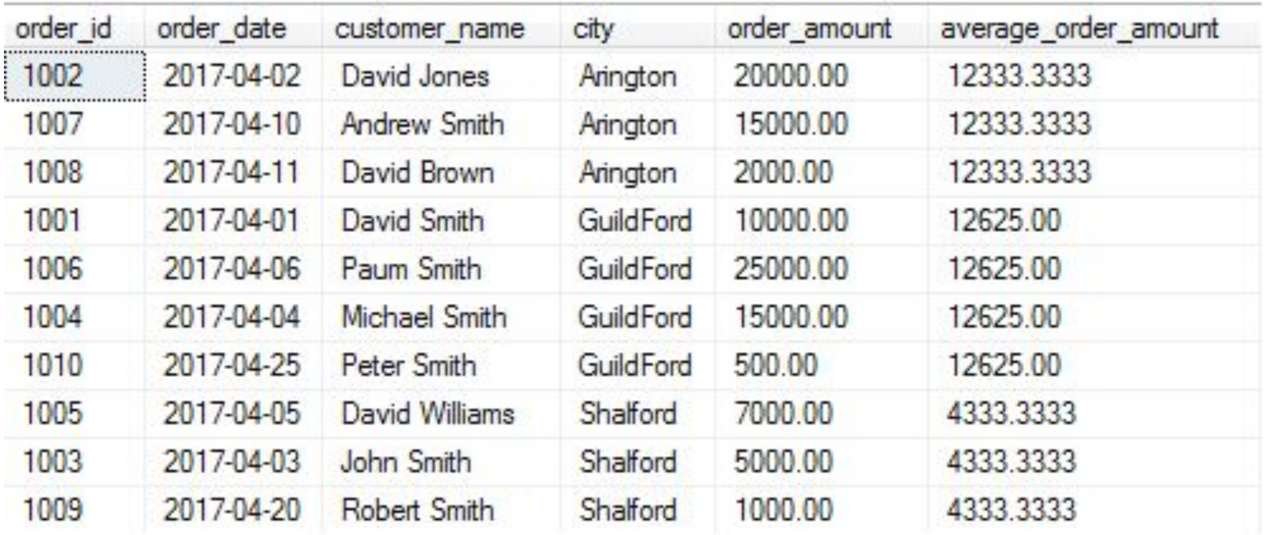
In this example, we will display the average_order_amount for a city for each row:

Query

```
SELECT order_id, order_date, customer_name, city, order_amount

,AVG(order_amount) OVER(PARTITION BY city) as   average_order_amount

FROM Orders
```

AVG() Window function

Output



order_id	order_date	customer_name	city	order_amount	average_order_amount
1002	2017-04-02	David Jones	Arington	20000.00	12333.3333
1007	2017-04-10	Andrew Smith	Arington	15000.00	12333.3333
1008	2017-04-11	David Brown	Arington	2000.00	12333.3333
1001	2017-04-01	David Smith	GuildFord	10000.00	12625.00
1006	2017-04-06	Paum Smith	GuildFord	25000.00	12625.00
1004	2017-04-04	Michael Smith	GuildFord	15000.00	12625.00
1010	2017-04-25	Peter Smith	GuildFord	500.00	12625.00
1005	2017-04-05	David Williams	Shalford	7000.00	4333.3333
1003	2017-04-03	John Smith	Shalford	5000.00	4333.3333
1009	2017-04-20	Robert Smith	Shalford	1000.00	4333.3333

Practice Question

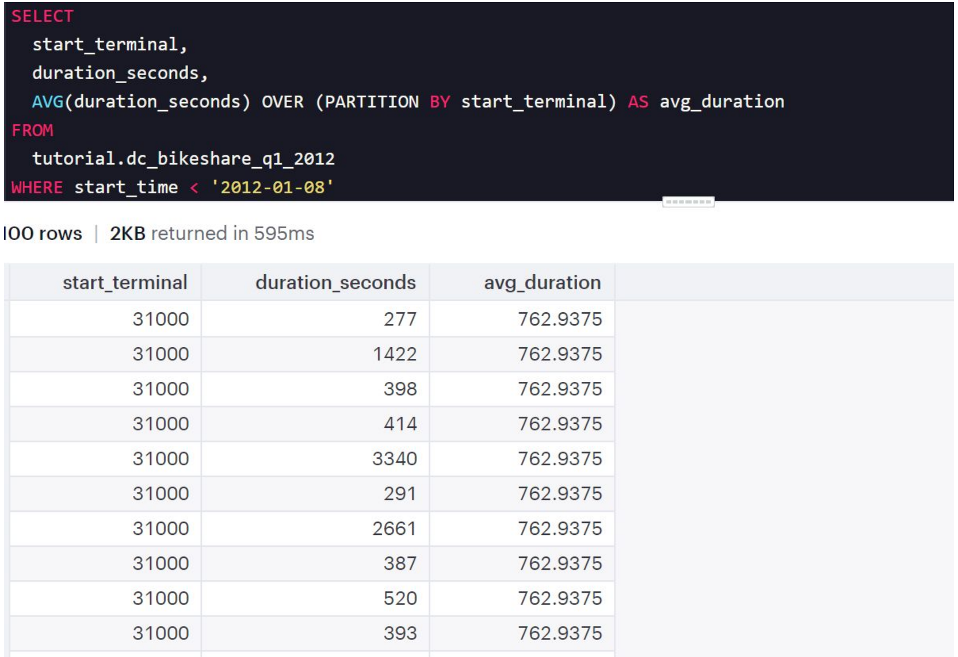
Write a query to find the average of all the durations_seconds for all the trips for each terminal. Display the data against each row in the dataset. Partition the data at the start_terminal. Consider the data where start_time is before ‘2012-01-08’.

Solution

```
SELECT
        start_terminal,
        duration_seconds,
        AVG(duration_seconds) OVER (PARTITION BY start_terminal) AS avg_duration
FROM
        tutorial.dc_bikeshare_q1_2012
WHERE start_time < '2012-01-08'
```

Solution

Output



```sql
SELECT
  start_terminal,
  duration_seconds,
  AVG(duration_seconds) OVER (PARTITION BY start_terminal) AS avg_duration
FROM
  tutorial.dc_bikeshare_q1_2012
WHERE start_time < '2012-01-08'
```

100 rows | 2KB returned in 595ms

start_terminal	duration_seconds	avg_duration
31000	277	762.9375
31000	1422	762.9375
31000	398	762.9375
31000	414	762.9375
31000	3340	762.9375
31000	291	762.9375
31000	2661	762.9375
31000	387	762.9375
31000	520	762.9375
31000	393	762.9375

MIN() Window function

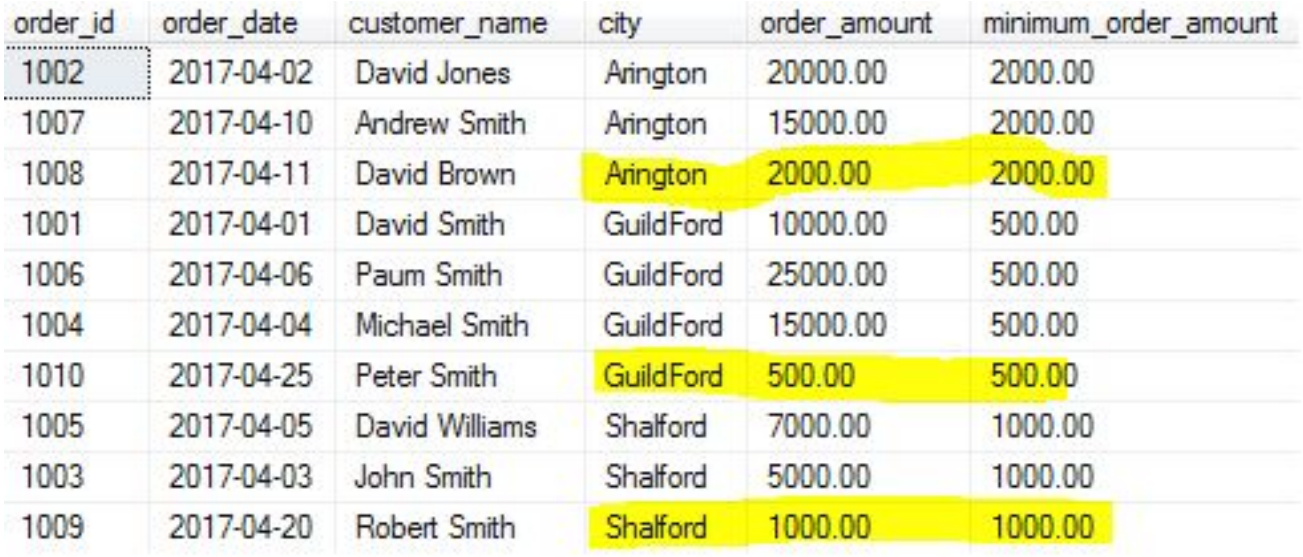
In this example, we will display the minimum_order_amount for a city for each row:

Query

```
SELECT order_id, order_date, customer_name, city, order_amount

,MIN(order_amount) OVER(PARTITION BY city) as minimum_order_amount

FROM orders
```

MIN() Window function

Output



order_id	order_date	customer_name	city	order_amount	minimum_order_amount
1002	2017-04-02	David Jones	Arington	20000.00	2000.00
1007	2017-04-10	Andrew Smith	Arington	15000.00	2000.00
1008	2017-04-11	David Brown	Arington	2000.00	2000.00
1001	2017-04-01	David Smith	GuildFord	10000.00	500.00
1006	2017-04-06	Paum Smith	GuildFord	25000.00	500.00
1004	2017-04-04	Michael Smith	GuildFord	15000.00	500.00
1010	2017-04-25	Peter Smith	GuildFord	500.00	500.00
1005	2017-04-05	David Williams	Shalford	7000.00	1000.00
1003	2017-04-03	John Smith	Shalford	5000.00	1000.00
1009	2017-04-20	Robert Smith	Shalford	1000.00	1000.00

Practice Question

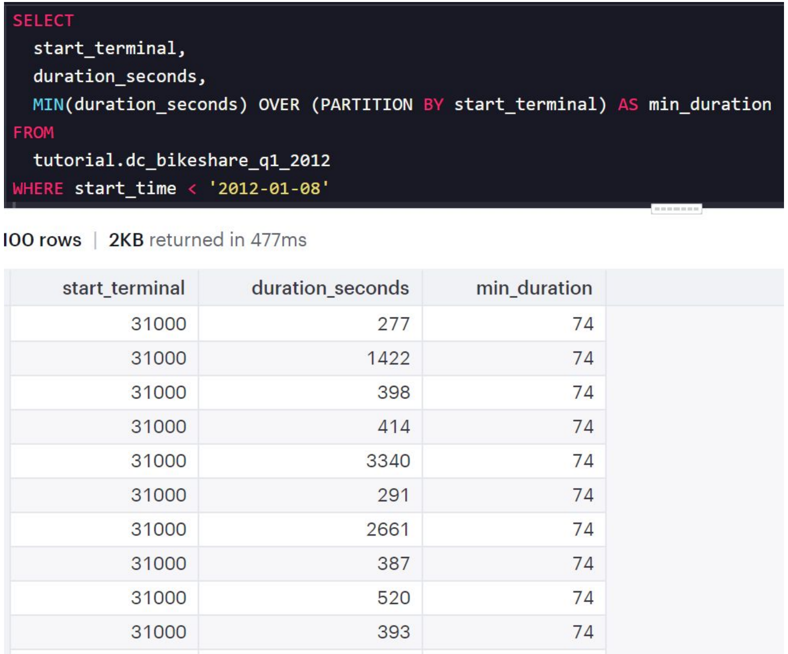
Write a query to display the minimum durations_seconds for all the trips for each terminal at each row. Partition the data at start_terminal. Consider the data where start_time is before ‘2012-01-08’.

Solution

```
SELECT
          start_terminal,
          duration_seconds,
          MIN(duration_seconds) OVER (PARTITION BY start_terminal) AS min_duration
FROM
          tutorial.dc_bikeshare_q1_2012
WHERE start_time < '2012-01-08'
```

Solution

Output



```
SELECT
  start_terminal,
  duration_seconds,
  MIN(duration_seconds) OVER (PARTITION BY start_terminal) AS min_duration
FROM
  tutorial.dc_bikeshare_q1_2012
WHERE start_time < '2012-01-08'
```

100 rows | 2KB returned in 477ms

start_terminal	duration_seconds	min_duration
31000	277	74
31000	1422	74
31000	398	74
31000	414	74
31000	3340	74
31000	291	74
31000	2661	74
31000	387	74
31000	520	74
31000	393	74

MAX() Window function

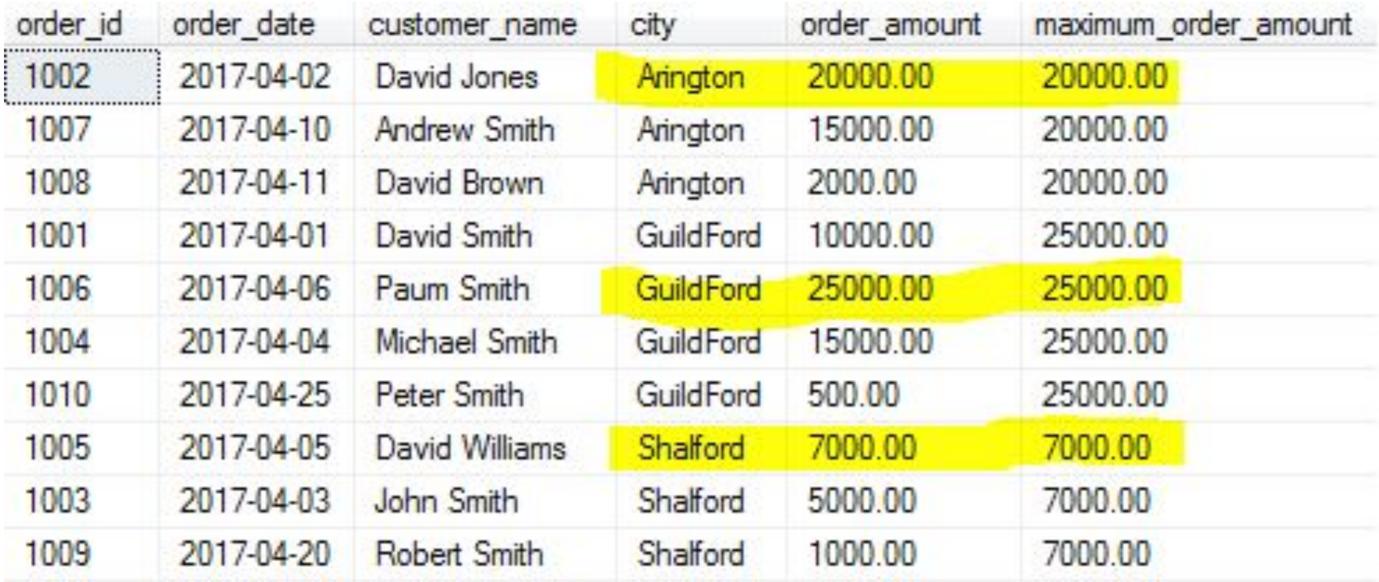
In this example, we will display the maximum_order_amount for a city for each row:

Query

```
SELECT order_id, order_date, customer_name, city, order_amount

,MAX(order_amount) OVER(PARTITION BY city) as minimum_order_amount

FROM orders
```

MAX() Window function

Output



order_id	order_date	customer_name	city	order_amount	maximum_order_amount
1002	2017-04-02	David Jones	Arington	20000.00	20000.00
1007	2017-04-10	Andrew Smith	Arington	15000.00	20000.00
1008	2017-04-11	David Brown	Arington	2000.00	20000.00
1001	2017-04-01	David Smith	GuildFord	10000.00	25000.00
1006	2017-04-06	Paum Smith	GuildFord	25000.00	25000.00
1004	2017-04-04	Michael Smith	GuildFord	15000.00	25000.00
1010	2017-04-25	Peter Smith	GuildFord	500.00	25000.00
1005	2017-04-05	David Williams	Shalford	7000.00	7000.00
1003	2017-04-03	John Smith	Shalford	5000.00	7000.00
1009	2017-04-20	Robert Smith	Shalford	1000.00	7000.00

Practice Question

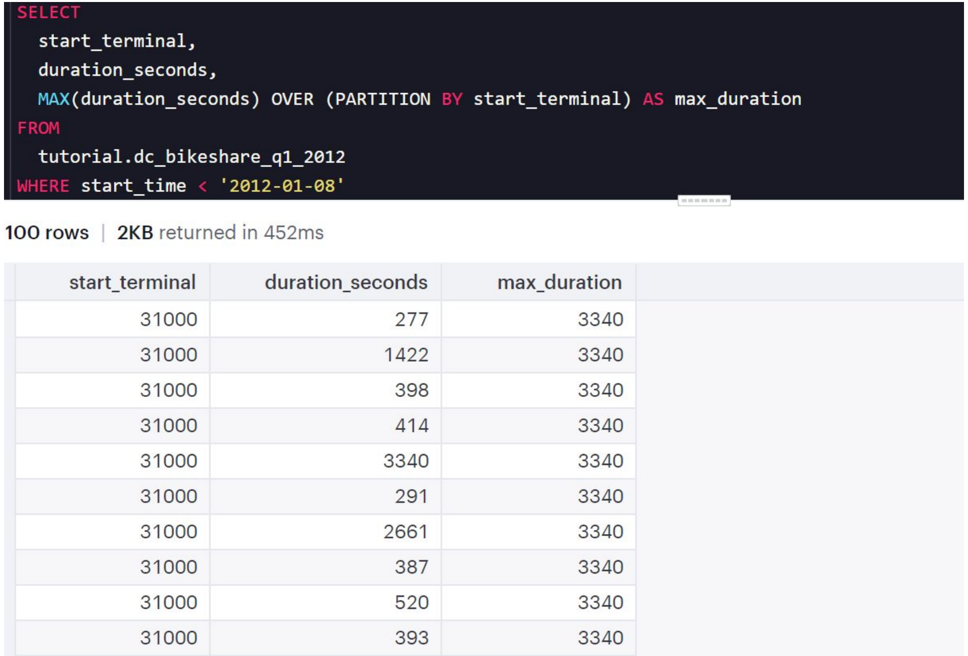
Write a query to display the maximum durations_seconds for all the trips for each terminal at each row. Partition the data at start_terminal. Consider the data where start_time is before ‘2012-01-08’.

Solution

```
SELECT
          start_terminal,
          duration_seconds,
          MAX(duration_seconds) OVER (PARTITION BY start_terminal) AS max_duration
FROM
          tutorial.dc_bikeshare_q1_2012
WHERE start_time < '2012-01-08'
```

Solution

Output



```
SELECT
  start_terminal,
  duration_seconds,
  MAX(duration_seconds) OVER (PARTITION BY start_terminal) AS max_duration
FROM
  tutorial.dc_bikeshare_q1_2012
WHERE start_time < '2012-01-08'
```

100 rows | 2KB returned in 452ms

start_terminal	duration_seconds	max_duration
31000	277	3340
31000	1422	3340
31000	398	3340
31000	414	3340
31000	3340	3340
31000	291	3340
31000	2661	3340
31000	387	3340
31000	520	3340
31000	393	3340

COUNT() Window function

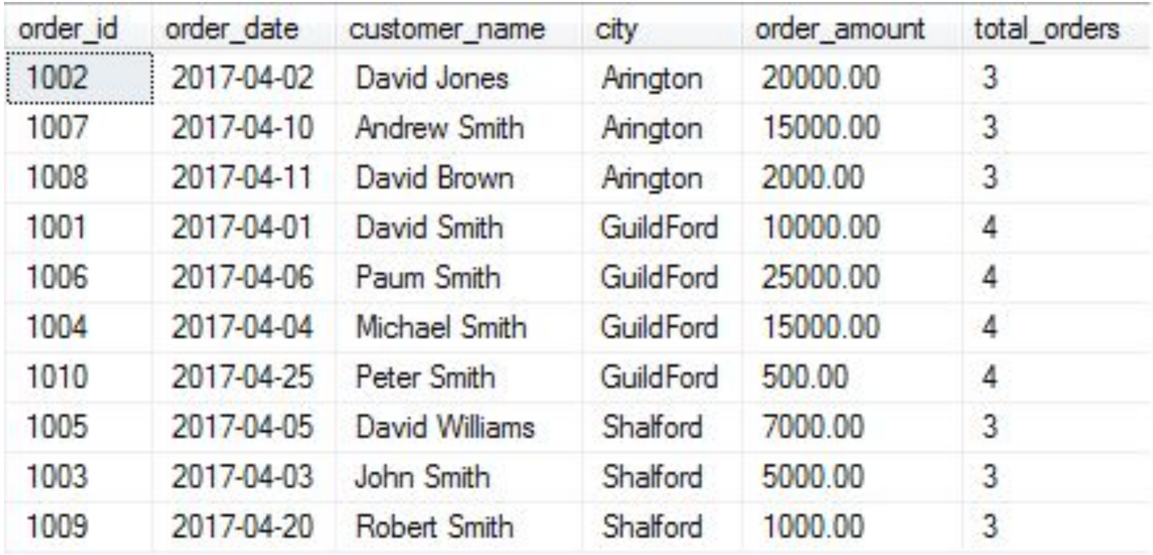
In this example, we will display the total_orders for a city for each row:

Query

```
SELECT order_id, order_date, customer_name, city, order_amount

,COUNT(order_id) OVER(PARTITION BY city) as minimum_order_amount

FROM orders
```

COUNT() Window function

Output



order_id	order_date	customer_name	city	order_amount	total_orders
1002	2017-04-02	David Jones	Arington	20000.00	3
1007	2017-04-10	Andrew Smith	Arington	15000.00	3
1008	2017-04-11	David Brown	Arington	2000.00	3
1001	2017-04-01	David Smith	GuildFord	10000.00	4
1006	2017-04-06	Paum Smith	GuildFord	25000.00	4
1004	2017-04-04	Michael Smith	GuildFord	15000.00	4
1010	2017-04-25	Peter Smith	GuildFord	500.00	4
1005	2017-04-05	David Williams	Shalford	7000.00	3
1003	2017-04-03	John Smith	Shalford	5000.00	3
1009	2017-04-20	Robert Smith	Shalford	1000.00	3

Practice Question

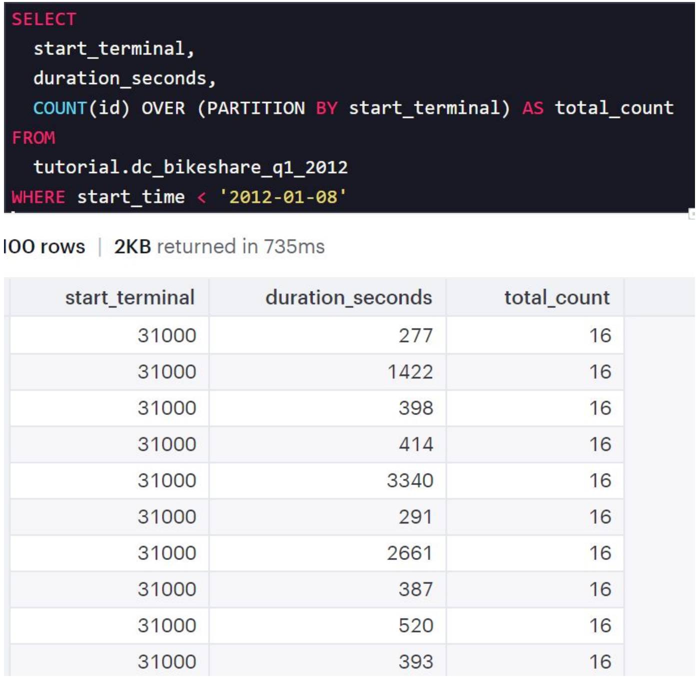
Display the total number of rides per terminal at each row. Partition the data at the start_terminal. Consider the data where start_time is before ‘2012-01-08’.

Solution

```
SELECT
        start_terminal,
        duration_seconds,
        COUNT(id) OVER (PARTITION BY start_terminal) AS total_count
FROM
        tutorial.dc_bikeshare_q1_2012
WHERE start_time < '2012-01-08'
```

Solution

Output



```
SELECT
  start_terminal,
  duration_seconds,
  COUNT(id) OVER (PARTITION BY start_terminal) AS total_count
FROM
  tutorial.dc_bikeshare_q1_2012
WHERE start_time < '2012-01-08'
```

100 rows | **2KB** returned in 735ms

start_terminal	duration_seconds	total_count
31000	277	16
31000	1422	16
31000	398	16
31000	414	16
31000	3340	16
31000	291	16
31000	2661	16
31000	387	16
31000	520	16
31000	393	16

Understanding the Frame Clause

Earlier in this topic, we introduced two concepts:

- ROWS
- Frame Clause

In all the examples discussed above, we take all the rows within a window for analysis. However, there might be some situations where we might be required to select some rows within a window—for example, cumulative sum, the rolling average for a week.

Frame clause and rows allow us to limit the number of selected rows within the window.

Understanding the Frame Clause

Here’s what the generic syntax looks like:

ROWS BETWEEN <starting_row> **AND** <ending_row>

In the <starting_row> and <ending row>, we have the following options at our disposal:

- UNBOUNDED PRECEDING — all rows before the current row in the partition, i.e. the first row of the partition
- [some #] PRECEDING — # of rows before the current row
- CURRENT ROW — the current row
- [some #] FOLLOWING — # of rows after the current row
- UNBOUNDED FOLLOWING — all rows after the current row in the partition, i.e. the last row of the partition

Understanding the Frame Clause

Here are some examples of ways to write it:

- ROWS BETWEEN 3 PREVIOUS AND CURRENT ROW indicates going back three rows to the current row.
- ROWS BETWEEN UNBOUNDED PRECEDING AND 1 FOLLOWING — this signifies that you should look from the first row of the partition to one row following the current row.
- ROWS BETWEEN 5 AND 1 PRIOR — this means to go back five rows and search up to 1 row before the current row.
- ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING — this means that you should look from the first row of the partition to the last row of the partition.

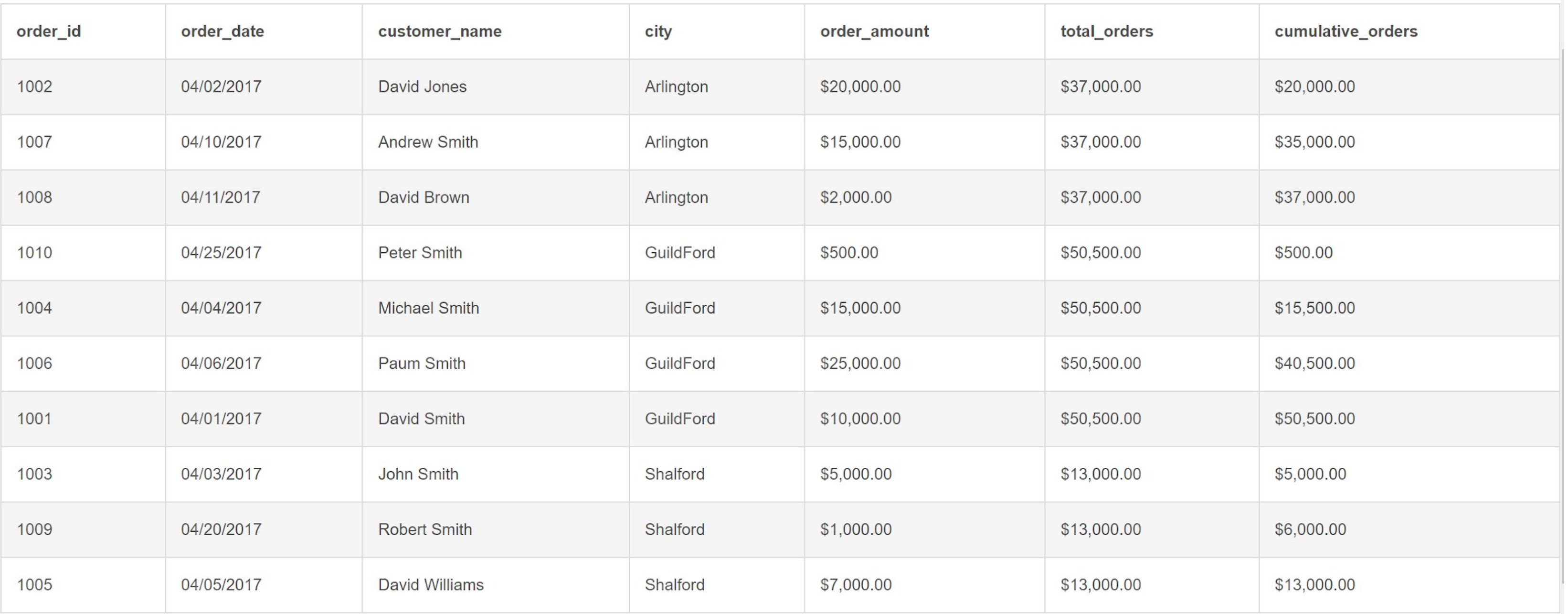
Understanding the Frame Clause - Example

In this example, we will display the total_orders and cumulative_orders for a city for each row.

Query

```
SELECT order_id, order_date, customer_name, city, order_amount

SUM(order_amount) OVER(PARTITION BY city) as total_orders,

SUM(order_amount) OVER(PARTITION BY city ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) as cumulative_orders

FROM orders
```

Understanding the Frame Clause - Example



order_id	order_date	customer_name	city	order_amount	total_orders	cumulative_orders
1002	04/02/2017	David Jones	Arlington	$20,000.00	$37,000.00	$20,000.00
1007	04/10/2017	Andrew Smith	Arlington	$15,000.00	$37,000.00	$35,000.00
1008	04/11/2017	David Brown	Arlington	$2,000.00	$37,000.00	$37,000.00
1010	04/25/2017	Peter Smith	GuildFord	$500.00	$50,500.00	$500.00
1004	04/04/2017	Michael Smith	GuildFord	$15,000.00	$50,500.00	$15,500.00
1006	04/06/2017	Paum Smith	GuildFord	$25,000.00	$50,500.00	$40,500.00
1001	04/01/2017	David Smith	GuildFord	$10,000.00	$50,500.00	$50,500.00
1003	04/03/2017	John Smith	Shalford	$5,000.00	$13,000.00	$5,000.00
1009	04/20/2017	Robert Smith	Shalford	$1,000.00	$13,000.00	$6,000.00
1005	04/05/2017	David Williams	Shalford	$7,000.00	$13,000.00	$13,000.00

Practice Question

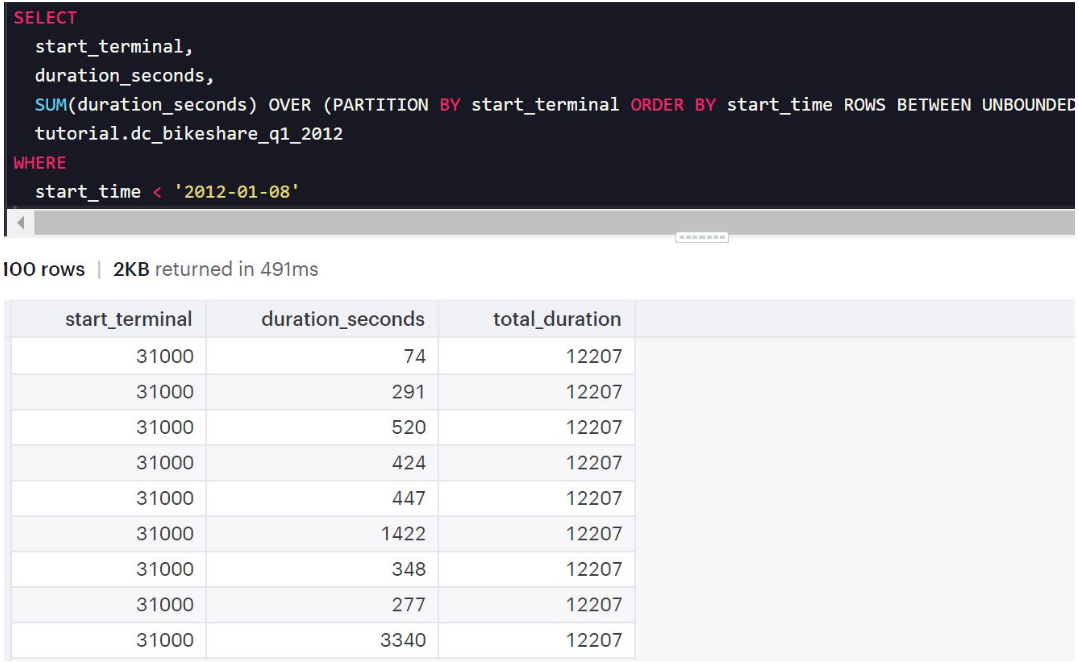
Write a query to find the total (sum) of all the durations_seconds for all the trips for each terminal. Display the data against each row in the dataset. Partition the data at the start_terminal and sort the rows in the window by start_time of the trip. Consider the data where start_time is before ‘2012-01-08’.

Solution

```
SELECT

    start_terminal,

    duration_seconds,

    SUM(duration_seconds) OVER (PARTITION BY start_terminal ORDER BY start_time ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS total_duration FROM

    tutorial.dc_bikeshare_q1_2012

WHERE

    start_time < '2012-01-08'
```

Solution

Output



```
SELECT
  start_terminal,
  duration_seconds,
  SUM(duration_seconds) OVER (PARTITION BY start_terminal ORDER BY start_time ROWS BETWEEN UNBOUNDED
  tutorial.dc_bikeshare_q1_2012
WHERE
  start_time < '2012-01-08'
```

100 rows | 2KB returned in 491ms

start_terminal	duration_seconds	total_duration
31000	74	12207
31000	291	12207
31000	520	12207
31000	424	12207
31000	447	12207
31000	1422	12207
31000	348	12207
31000	277	12207
31000	3340	12207

Practice Question

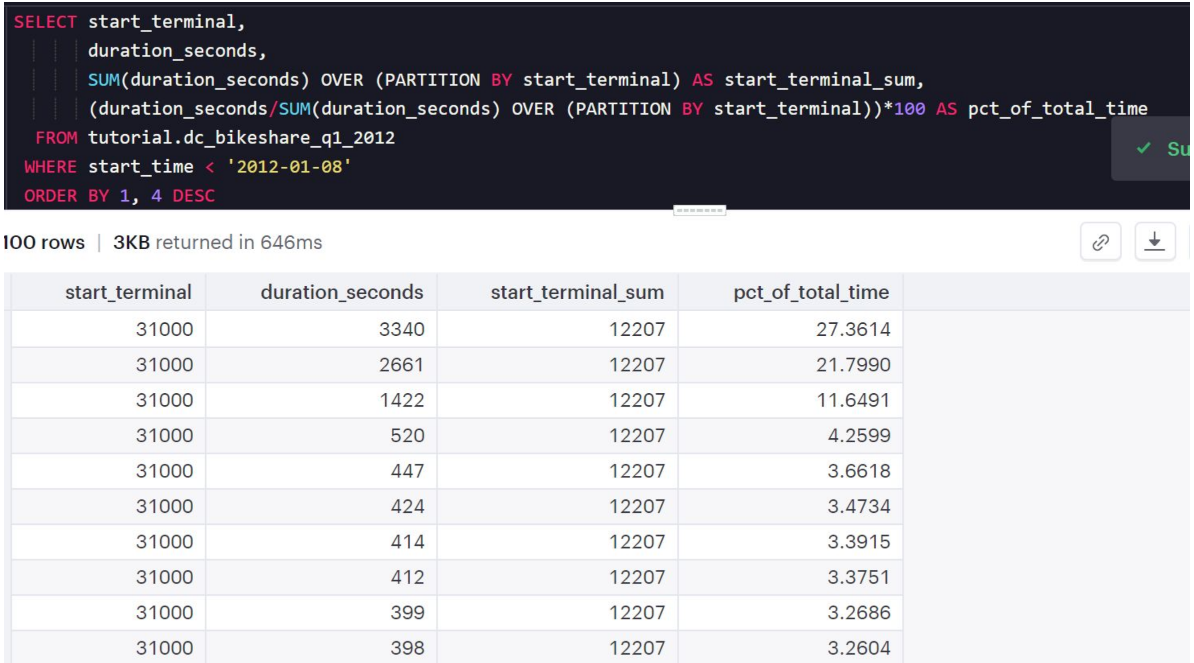
Write a query modification of the above example query that shows the duration of each ride as a percentage of the total time accrued by riders from each start_terminal. Order the data by start_terminal and pct_of_tal_time in descending order.

Solution

```
SELECT

          start_terminal,

          duration_seconds,

          SUM(duration_seconds) OVER (PARTITION BY start_terminal) AS start_terminal_sum,
(duration_seconds/SUM(duration_seconds) OVER (PARTITION BY start_terminal))*100 AS pct_of_total_time

FROM

          tutorial.dc_bikeshare_q1_2012

WHERE start_time < '2012-01-08’

ORDER BY 1, 4 DESC
```

Solution

Output



```
SELECT start_terminal,
       duration_seconds,
       SUM(duration_seconds) OVER (PARTITION BY start_terminal) AS start_terminal_sum,
       (duration_seconds/SUM(duration_seconds) OVER (PARTITION BY start_terminal))*100 AS pct_of_total_time
  FROM tutorial.dc_bikeshare_q1_2012
 WHERE start_time < '2012-01-08'
 ORDER BY 1, 4 DESC
```

100 rows | 3KB returned in 646ms

start_terminal	duration_seconds	start_terminal_sum	pct_of_total_time
31000	3340	12207	27.3614
31000	2661	12207	21.7990
31000	1422	12207	11.6491
31000	520	12207	4.2599
31000	447	12207	3.6618
31000	424	12207	3.4734
31000	414	12207	3.3915
31000	412	12207	3.3751
31000	399	12207	3.2686
31000	398	12207	3.2604

In the next class we will study:

- Window functions - II

Conclusion