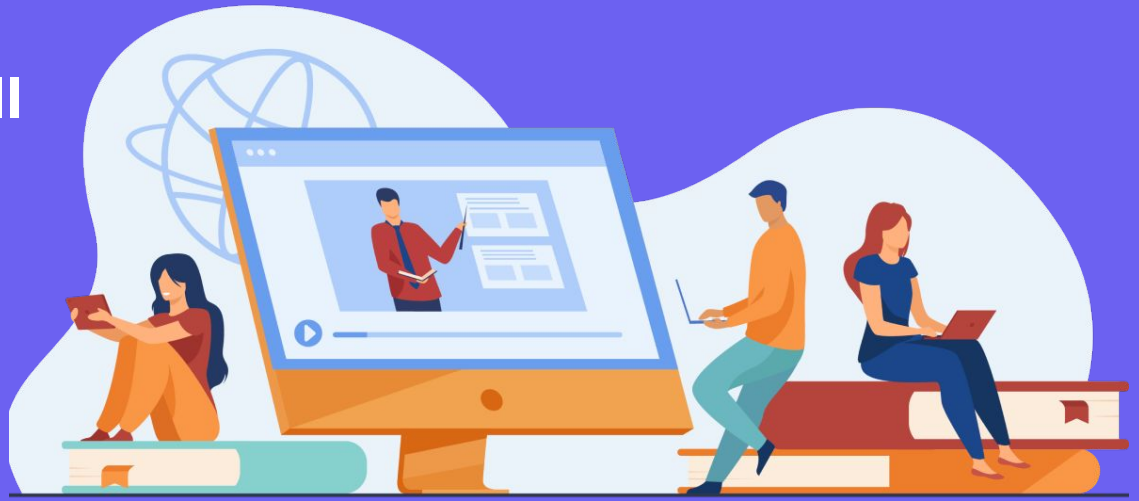
Windows Function II



Relevel
by Unacademy

Introduction to Ranking functions

In the same way Window aggregate functions aggregate the value of a specified field, RANKING functions rank the value of a specified field and categorise it based on its rank.

The most common application of RANKING functions is to locate the top (N) records based on a given value. For example, the top ten highest-paid employees, the top ten ranked students, the top 50 largest orderst, etc.

We will learn about following ranking function:

- RANK()
- DENSE_RANK()
- ROW_NUMBER()
- NTILE()

Understanding Ranking functions with an example
1. RANK() Window Function

We will use the below mentioned table ‘orders’ for understanding ranking 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

RANK() Window function

The RANK() function assigns a unique rank to each record based on a specified value, such as salary or order amount.

If two records have the same value, the RANK() function will give both records the same rank by skipping the next rank. If two identical values are found at rank 2, it will assign the same rank 2 to both records, skip rank 3, and assign rank 4 to the next record.

RANK() Window function

In this example, we will rank each order by the order amount:

Query

```
SELECT
        order_id,
        order_date,
        customer_name,
        city,
        RANK() OVER(ORDER BY order_amount DESC) AS rank
FROM
        orders
```

RANK() Window function

Output

	order_id	order_date	customer_name	city	order_amount	Rank
1	1006	2017-04-06	Paum Smith	GuildFord	25000.00	1
2	1002	2017-04-02	David Jones	Arington	20000.00	2
3	1004	2017-04-04	Michael Smith	GuildFord	15000.00	3
4	1007	2017-04-10	Andrew Smith	Arington	15000.00	3
5	1001	2017-04-01	David Smith	GuildFord	10000.00	5
6	1005	2017-04-05	David Williams	Shalford	7000.00	6
7	1003	2017-04-03	John Smith	Shalford	5000.00	7
8	1008	2017-04-11	David Brown	Arington	2000.00	8
9	1009	2017-04-20	Robert Smith	Shalford	1000.00	9
10	1010	2017-04-25	Peter Smith	GuildFord	500.00	10

Same rank for two identical records (Rank = 3) and then skipping next rank (4) and assigning rank 5 to the next record.

From the above image, you can see that the same rank (3) is assigned to two identical records (each having an order amount of 15,000) and it then skips the next rank (4) and assigns rank 5 to the next record.

Instructions for practice questions

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

Practice Question

Write a query to rank all the rides per terminal on the ascending order of the duration taken per ride.

Solution

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

Solution

Output

start_terminal	duration_seconds	rank
31000	74	1
31000	291	2
31000	520	3
31000	424	4
31000	447	4
31000	1422	6
31000	348	7
31000	277	8
31000	3340	9

Practice Question

Write a query that shows the 5 longest rides from each starting terminal, ordered by terminal, and longest to shortest rides within each terminal. There are limit to rides that occurred before Jan. 8, 2012.

Solution

```
SELECT *
  FROM (
        SELECT start_terminal,
               start_time,
               duration_seconds AS trip_time,
               RANK() OVER (PARTITION BY start_terminal ORDER BY duration_seconds DESC) AS rank
          FROM tutorial.dc_bikeshare_q1_2012
         WHERE start_time < '2012-01-08'
               ) sub
 WHERE sub.rank <= 5
```

Solution

Output

start_terminal	start_time	trip_time	rank
31000	2012-01-05 17:25:00	3340	1
31000	2012-01-06 17:29:00	2661	2
31000	2012-01-03 12:32:00	1422	3
31000	2012-01-02 19:15:00	520	4
31000	2012-01-03 07:22:00	447	5
31001	2012-01-01 15:09:00	3624	1
31001	2012-01-01 15:09:00	3598	2
31001	2012-01-01 13:35:00	2876	3
31001	2012-01-01 13:36:00	2804	4

2. DENSE_RANK() Window Function

We will use the below mentioned table ‘orders’ for understanding DENSE_RANK 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

DENSE_RANK() Window function

The DENSE RANK() function is the same as the RANK() function, except it, does not skip any ranks.

If two identical records are found, DENSE RANK() will assign the same rank to both but will not skip to the next rank.

DENSE_RANK() Window function

In this example, we will rank each order by the order amount:

Query

```
SELECT
        order_id,
        order_date,
        customer_name,city,
        order_amount,
        DENSE_RANK() OVER(ORDER BY order_amount DESC) AS Rank
FROM
        Orders
```

DENSE_RANK() Window function

Output

	order_id	order_date	customer_name	city	order_amount	Rank
1	1006	2017-04-06	Paum Smith	GuildFord	25000.00	1
2	1002	2017-04-02	David Jones	Arington	20000.00	2
3	1004	2017-04-04	Michael Smith	GuildFord	15000.00	3
4	1007	2017-04-10	Andrew Smith	Arington	15000.00	3
5	1001	2017-04-01	David Smith	GuildFord	10000.00	4
6	1005	2017-04-05	David Williams	Shalford	7000.00	5
7	1003	2017-04-03	John Smith	Shalford	5000.00	6
8	1008	2017-04-11	David Brown	Arington	2000.00	7
9	1009	2017-04-20	Robert Smith	Shalford	1000.00	8
10	1010	2017-04-25	Peter Smith	GuildFord	500.00	9

As you can clearly see above, the same rank is given to two identical records (each having the same order amount) and then the next rank number is given to the next record without skipping a rank value.

3. ROW_NUMBER() Window Function

We will use the below mentioned table ‘orders’ for understanding ROW_NUMBER 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

ROW_NUMBER() Window function

The name is self-explanatory. These functions assign a unique row number to each record. The row number will be reset for each partition if PARTITION BY is specified. Let's see how ROW_NUMBER() works without PARTITION BY and then with PARTITION BY.

ROW_NUMBER() Without Partition By

In this example, we will give row number to each order by the order_id:

Query

```
SELECT
        order_id,
        order_date,
        customer_name,
        city,
        order_amount,
        ROW_NUMBER() OVER(ORDER BY order_id) AS row_number
FROM
        Orders
```

ROW_NUMBER() Without Partition By

Output

	order_id	order_date	customer_name	city	order_amount	row_number
1	1001	2017-04-01	David Smith	GuildFord	10000.00	1
2	1002	2017-04-02	David Jones	Arington	20000.00	2
3	1003	2017-04-03	John Smith	Shalford	5000.00	3
4	1004	2017-04-04	Michael Smith	GuildFord	15000.00	4
5	1005	2017-04-05	David Williams	Shalford	7000.00	5
6	1006	2017-04-06	Paum Smith	GuildFord	25000.00	6
7	1007	2017-04-10	Andrew Smith	Arington	15000.00	7
8	1008	2017-04-11	David Brown	Arington	2000.00	8
9	1009	2017-04-20	Robert Smith	Shalford	1000.00	9
10	1010	2017-04-25	Peter Smith	GuildFord	500.00	10

Practice Question

Write a query to assign a row number to the entire dataset ordered by the start_time. Consider the data where the start_time is before ‘2012-01-08’.

Solution

```
SELECT
        start_terminal,
        start_time,
        duration_seconds,
        ROW_NUMBER() OVER (ORDER BY start_time) AS row_number
FROM
        tutorial.dc_bikeshare_q1_2012
WHERE
        start_time < '2012-01-08'
```

Solution

Output

start_terminal	start_time	duration_seconds	row_number
31245	2012-01-01 00:04:00	475	1
31400	2012-01-01 00:10:00	1162	2
31400	2012-01-01 00:10:00	1145	3
31101	2012-01-01 00:15:00	485	4
31102	2012-01-01 00:15:00	471	5
31017	2012-01-01 00:17:00	358	6
31236	2012-01-01 00:18:00	1754	7
31101	2012-01-01 00:22:00	259	8
31014	2012-01-01 00:24:00	516	9

ROW_NUMBER() WITH Partition By

In this example, we will give row number to each order partitioned by city:

Query

```
SELECT
        order_id,
        order_date,
        customer_name,
        city,
        order_amount,
        ROW_NUMBER() OVER(PARTITION BY city ORDER BY order_amount DESC) AS row_number
FROM
        Orders
```

ROW_NUMBER() With Partition By

Output

	order_id	order_date	customer_name	city	order_amount	row_number
1	1002	2017-04-02	David Jones	Arington	20000.00	1
2	1007	2017-04-10	Andrew Smith	Arington	15000.00	2
3	1008	2017-04-11	David Brown	Arington	2000.00	3
4	1006	2017-04-06	Paum Smith	GuildFord	25000.00	1
5	1004	2017-04-04	Michael Smith	GuildFord	15000.00	2
6	1001	2017-04-01	David Smith	GuildFord	10000.00	3
7	1010	2017-04-25	Peter Smith	GuildFord	500.00	4
8	1005	2017-04-05	David Williams	Shalford	7000.00	1
9	1003	2017-04-03	John Smith	Shalford	5000.00	2
10	1009	2017-04-20	Robert Smith	Shalford	1000.00	3

Note that we have done the partition on city. This means that the row number is reset for each city and so it restarts at 1 again. However, the order of the rows is determined by the order amount so that for any given city the largest order amount will be the first row and so the assigned row number is 1.

Difference Between RANK, DENSE_RANK, and ROW_NUMBER

- **RANK:** It assigns the rank number to each row in a partition. It skips the number for similar values.

- **DENSE_RANK:** It assigns the rank number to each row in a partition. It does not skip the number for similar values.

- **ROW_NUMBER**: It assigns the sequential rank number to each unique record.

3. NTILE() Window Function

We will use the below mentioned table ‘orders’ for understanding ROW_NUMBER 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

NTILE() Window function

The window function NTILE() is extremely useful. It assists you in determining which percentile (or quartile, or other subdivision) a given row belongs to.

If you have 100 rows and want to create four quartiles based on a specific value field, you can easily do so and see how many rows fall into each quartile.

NTILE() Window function

Let's see an example. In the query below, we have specified that we want to create four quartiles based on the order amount. We then want to see how many orders fall into each quartile.

Query

```
SELECT
        order_id,
        order_date,
        customer_name,
        city,
        order_amount,
        NTILE(4) OVER(ORDER BY order_amount) AS quartile
FROM
        orders
```

NTILE() Window function

	order_id	order_date	customer_name	city	order_amount	quartile
1	1010	2017-04-25	Peter Smith	GuildFord	500.00	1
2	1009	2017-04-20	Robert Smith	Shalford	1000.00	1
3	1008	2017-04-11	David Brown	Arington	2000.00	1
4	1003	2017-04-03	John Smith	Shalford	5000.00	2
5	1005	2017-04-05	David Williams	Shalford	7000.00	2
6	1001	2017-04-01	David Smith	GuildFord	10000.00	2
7	1004	2017-04-04	Michael Smith	GuildFord	15000.00	3
8	1007	2017-04-10	Andrew Smith	Arington	15000.00	3
9	1002	2017-04-02	David Jones	Arington	20000.00	4
10	1006	2017-04-06	Paum Smith	GuildFord	25000.00	4

Quartile 1:
Order Amount <= 4,999

Quartile 2:
Order Amount >= 5,000 AND <= 10,000

Quartile 3:
Order Amount >10,000 AND <= 15,000

Quartile 4:
Order Amount above 15,000

NTILE creates tiles based on the following formula:

No. of rows in each tile = number of rows in result set / number of tiles specified.Here is our example. We have a total of 10 rows and 4 tiles are specified in the query, so the number of rows in each tile will be 2.5 (10/4). As the number of rows should be a whole number and not a decimal, the SQL engine will assign 3 rows for the first two groups and 2 rows for the remaining two groups.

Practice Question

Write a query that shows only the duration of the trip and the percentile into which that duration falls (across the entire dataset—not partitioned by terminal).

Solution

```
SELECT
        duration_seconds,
        NTILE(100) OVER (ORDER BY duration_seconds) AS percentile
FROM
        tutorial.dc_bikeshare_q1_2012
WHERE
        start_time < '2012-01-08'
ORDER BY 1
```

Solution

Output

duration_seconds	percentile
2	1
2	1
2	1
2	1
3	1
3	1
3	1
3	1
3	1

Intro to Value Window functions

Value window functions are used to find the first, last, previous and next values. The functions that can be used are:

- LAG(): to find the previous value
- LEAD(): to find the next value
- FIRST_VALUE(): to find the first value
- LAST_VALUE(): to find the last value

Understanding Value functions with an example
1. LAG() Window Function

We will use the below mentioned table ‘orders’ for understanding value 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

LAG() Window function

The LAG function allows to access data from the previous row in the same result set without the use of any SQL joins.

Syntax

```
LAG(return_value ,offset)

OVER (

   [PARTITION BY partition_expression, ... ]

   ORDER BY sort_expression [ASC | DESC], ...

)
```

return_value: The return value of the previous row based on a specified offset. The return value must evaluate to a single value and cannot be another window function.

offset: The number of rows back from the current row from which to access data. offset can be an expression, subquery, or column that evaluates to a positive integer.

The default value of offset is 1 if you don't specify it explicitly.

LAG() Window function

In below example, we will use the LAG function to find the previous order date:

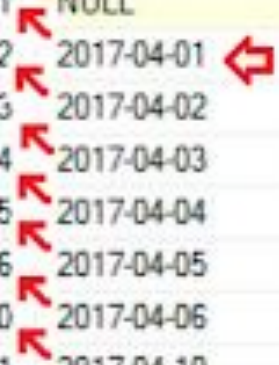
Query

```
SELECT
        order_id,
        customer_name,
        city,
        order_amount,order_date,
        LAG(order_date,1) OVER(ORDER BY order_date) AS prev_order_date --1 indicates check for previous row of the current row
FROM
        Orders
```

LAG() Window function

Output

order_id	customer_name	city	order_amount	order_date	prev_order_date
1001	David Smith	GuildFord	10000.00	2017-04-01	NULL
1002	David Jones	Arlington	20000.00	2017-04-02	2017-04-01
1003	John Smith	Shalford	5000.00	2017-04-03	2017-04-02
1004	Michael Smith	GuildFord	15000.00	2017-04-04	2017-04-03
1005	David Williams	Shalford	7000.00	2017-04-05	2017-04-04
1006	Paum Smith	GuildFord	25000.00	2017-04-06	2017-04-05
1007	Andrew Smith	Arlington	15000.00	2017-04-10	2017-04-06
1008	David Brown	Arlington	2000.00	2017-04-11	2017-04-10



LAG of order id 1002 is 1001, so here LAG(orderdate,1) will give you previous order date which is 2017-04-01

Arrow is pointing to LAG value of each record

Practice Question

Write a query to find the difference in the duration between two rides. Partition the data at start_terminal and order it by duration_seconds in ascending order. Consider the data where start_time is before ‘2012-01-08’.

Solution

```
SELECT
        start_terminal,
        duration_seconds,
        duration_seconds -LAG(duration_seconds, 1) OVER (PARTITION BY start_terminal ORDER BY duration_seconds)  AS
difference
FROM
        tutorial.dc_bikeshare_q1_2012
WHERE
        start_time < '2012-01-08’
ORDER BY
        start_terminal, duration_seconds
```

Solution

Output

start_terminal	duration_seconds	difference
31000	74	
31000	277	203
31000	291	14
31000	348	57
31000	387	39
31000	393	6
31000	398	5
31000	399	1
31000	412	13

Practice Question

Similar to previous question, write a query to find the difference in the duration between the two rides. Partition the data at start_terminal and order it by duration_seconds in ascending order. Consider the data where the start_time is before '2012-01-08'. Ensure that we only select data where the difference in the duration between two rides is not null.

Solution

```
SELECT *
  FROM (
    SELECT start_terminal,
           duration_seconds,
           duration_seconds -LAG(duration_seconds, 1) OVER
            (PARTITION BY start_terminal ORDER BY duration_seconds)
            AS difference
      FROM tutorial.dc_bikeshare_q1_2012
     WHERE start_time < '2012-01-08'
     ORDER BY start_terminal, duration_seconds
      ) sub
 WHERE sub.difference IS NOT NULL
```

Solution

Output

start_terminal	duration_seconds	difference
31000	74	
31000	277	203
31000	291	14
31000	348	57
31000	387	39
31000	393	6
31000	398	5
31000	399	1
31000	412	13

2. LEAD() Window Function

We will use the below mentioned table ‘orders’ for understanding LEAD window function:

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

LEAD() Window function

The LEAD function allows to access data from the next row in the same result set without the use of any SQL joins.

Syntax

```
LEAD(return_value ,offset)
OVER (
   [PARTITION BY partition_expression, ... ]
   ORDER BY sort_expression [ASC | DESC], ...
)
```

return_value: The return value of the next row based on a specified offset. The return value must evaluate to a single value and cannot be another window function.

offset: The number of rows forward from the current row from which to access data. offset can be an expression, subquery, or column that evaluates to a positive integer.

The default value of offset is 1 if you don't specify it explicitly.

LEAD() Window function

In below example, we will use LEAD function to find next order date:

Query

```
SELECT
        order_id,
        customer_name,
        city,
        order_amount,order_date,
        LEAD(order_date,1) OVER(ORDER BY order_date) AS next_order_date --1 indicates check for next row of the current row
FROM
        Orders
```

LEAD() Window function

Output

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

LEAD value of 1002 order is 1003, so here LEAD(orderdate,1) will be 2017-04-03 which is order date of 1003 (next record of order 1002)

Arrow is pointing to LEAD value of each record

3. FIRST_VALUE() AND LAST_VALUE() Window Function

We will use the below mentioned table ‘orders’ for understanding LEAD window function:

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

FIRST_VALUE() AND LAST_VALUE()

These functions help you to identify the first and last record within a partition or entire table if **PARTITION BY** is not specified.

ORDER BY clause is mandatory for FIRST_VALUE() and LAST_VALUE() functions.

FIRST_VALUE() AND LAST_VALUE()

Let’s find the first and last order of each city.

Query

```
SELECT
        order_id,
        order_date,
        customer_name,
        city,
        order_amount,
        FIRST_VALUE(order_date) OVER(PARTITION BY city ORDER BY city) first_order_date,
        LAST_VALUE(order_date) OVER(PARTITION BY city ORDER BY city) last_order_date
FROM
        Orders
```

FIRST_VALUE() AND LAST_VALUE()

Output

	order_id	order_date	customer_name	city	order_amount	first_order_date	last_order_date
1	1002	2017-04-02	David Jones	Arington	20000.00	2017-04-02	2017-04-11
2	1007	2017-04-10	Andrew Smith	Arington	15000.00	2017-04-02	2017-04-11
3	1008	2017-04-11	David Brown	Arington	2000.00	2017-04-02	2017-04-11
4	1001	2017-04-01	David Smith	GuildFord	10000.00	2017-04-01	2017-04-25
5	1006	2017-04-06	Paum Smith	GuildFord	25000.00	2017-04-01	2017-04-25
6	1004	2017-04-04	Michael Smith	GuildFord	15000.00	2017-04-01	2017-04-25
7	1010	2017-04-25	Peter Smith	GuildFord	500.00	2017-04-01	2017-04-25
8	1005	2017-04-05	David Williams	Shalford	7000.00	2017-04-05	2017-04-20
9	1003	2017-04-03	John Smith	Shalford	5000.00	2017-04-05	2017-04-20
10	1009	2017-04-20	Robert Smith	Shalford	1000.00	2017-04-05	2017-04-20

From the above image, we can clearly see that first order received on 2017-04-02 and last order received on 2017-04-11 for Arlington city and it works the same for other cities as well.

In the next class we will study:

Date Time Function

Thank You