WINDOW FUNCTIONS PART 2

```
CREATE OR REPLACE TABLE CONSUMER_COMPLAINTS
  (
     DATE_RECEIVED STRING ,
     PRODUCT_NAME VARCHAR2(50) ,
    SUB_PRODUCT VARCHAR2(60) ,
 ISSUE_DESC VARCHAR2(100),
     SUB_ISSUE VARCHAR2(100),
     CONSUMER_COMPLAINT_NARRATIVE STRING,
     Company_Public_Response STRING,
     Company VARCHAR(60),
     State_Name CHAR(4),
     Zip_Code VARCHAR(20),
     Tags_Name VARCHAR(50),
     Consumer_Consent_Provided CHAR(50),
     Submitted_Via STRING,
     Date_Sent_to_Company STRING,
     Company_Response_to_Consumer VARCHAR(50),
     Timely_Response CHAR(6),
     CONSUMER_DISPUTED CHAR(6),
     COMPLAINT_ID NUMBER(12,0) NOT NULL PRIMARY KEY
);
```

Load the excel file which has customer_complaints data from : https://github.com/anandjha90/analyticswithanand/blob/SNOWFLAKE/DATASETS/ConsumerComplaints(1).csv

```
SELECT * FROM DEMO_DB.DEMO_SCHEMA.CONSUMER_COMPLAINTS;
```

Apply some window functions on above for practice by going through below mentioned code.

WINDOW FUNCTIONS PART 2

```
CREATE OR REPLACE TABLE EMPLOYEE
(
  EMPID INTEGER NOT NULL PRIMARY KEY,
  EMP_NAME VARCHAR2(20),
  JOB_ROLE STRING,
  SALARY NUMBER(10,2)
);

INSERT INTO EMPLOYEE
VALUES('101','ANAND JHA','Analyst',50000);

INSERT INTO EMPLOYEE
VALUES(102,'ALex', 'Data Enginner',60000);

INSERT INTO EMPLOYEE
VALUES(103,'Ravi', 'Data Scientist',48000);

INSERT INTO EMPLOYEE
VALUES(104,'Peter', 'Analyst',98000);

INSERT INTO EMPLOYEE
VALUES(105,'Pulkit', 'Data Scientist',72000);

INSERT INTO EMPLOYEE
VALUES(106,'Robert','Analyst',100000);

INSERT INTO EMPLOYEE
VALUES(107,'Rishabh','Data Engineer',67000);
```

WINDOW FUNCTIONS PART 2

```
INSERT INTO EMPLOYEE
VALUES(108,'Subhash','Manager',148000);

INSERT INTO EMPLOYEE
VALUES(109,'Michaeal','Manager',213000);

INSERT INTO EMPLOYEE
VALUES(110,'Dhruv','Data Scientist',89000);

INSERT INTO EMPLOYEE
VALUES(111,'Amit Sharma','Analyst',72000);


SELECT * FROM EMPLOYEE;

update employee set job_role='Data Engineer'
where empid=102;

update employee set salary= 50000
where empid=104;
```


----------WINDOW FUNCTIONS----------

```
-- SYNTAX : window_function_name(<exprsn>) OVER (<partition_by_clause> <order_clause>)
```

WINDOW FUNCTIONS PART 2

```
--- display total salary based on job profile
SELECT JOB_ROLE,SUM(SALARY) FROM EMPLOYEE
GROUP BY 1
ORDER BY 2 DESC;

-- display total salary along with all the records ()every row value
SELECT * , SUM(SALARY) OVER() AS TOT_SALARY,
ROUND(SALARY / TOT_SALARY * 100,2) AS PERC_CONTRIBUTION
FROM EMPLOYEE
ORDER BY PERC_CONTRIBUTION DESC;

-- display the MAX salary per job category for all the rows
SELECT *,
MAX(SALARY) OVER(PARTITION BY JOB_ROLE) AS MAX_JOB_SAL,
MIN(SALARY) OVER(PARTITION BY JOB_ROLE) AS MIN_JOB_SAL,
ROUND(SALARY / MAX_JOB_SAL * 100,2) AS MAX_PERC_CONTRIBUTION_JOB_ROLE,
ROUND(SALARY / MIN_JOB_SAL * 100,2) AS MIN_PERC_CONTRIBUTION_JOB_ROLE
FROM EMPLOYEE;
--ORDER BY PERC_CONTRIBUTION_JOB_ROLE DESC;

select *,max(salary) over(partition by JOB_ROLE) as MAX_SAL ,
min(salary) over(partition by JOB_ROLE) as MIN_SAL,
SUM(salary) over(partition by JOB_ROLE) as TOT_SAL
from Employee;



--ARRANGING ROWS WITHIN EACH PARTITION BASED ON SALARY IN DESC ORDDER
```

WINDOW FUNCTIONS PART 2

```
select *,max(salary) over(partition by JOB_ROLE ORDER BY SALARY DESC) as MAX_SAL ,

min(salary) over(partition by JOB_ROLE ORDER BY SALARY ASC) as MIN_SAL,

SUM(salary) over(partition by JOB_ROLE) as TOT_SAL

from Employee

ORDER BY JOB_ROLE;


-- ROW_NUMBER() It assigns a unique sequential number to each row of the table ...

--SELECT * FROM

SELECT * FROM

(SELECT *,ROW_NUMBER() OVER(PARTITION BY JOB_ROLE ORDER BY SALARY DESC) AS
PART_ROW_NUM

FROM EMPLOYEE ) WHERE PART_ROW_NUM <=2;

--)

--WHERE PART_ROW_NUM <=2;


/* The RANK() window function, as the name suggests, ranks the rows within their partition based on
the given condition.

  In the case of ROW_NUMBER(), we have a sequential number.

  On the other hand, in the case of RANK(), we have the same rank for rows with the same value.

But there is a problem here. Although rows with the same value are assigned the same rank, the
subsequent rank skips the missing rank.

This wouldn't give us the desired results if we had to return "top N distinct" values from a table.

Therefore we have a different function to resolve this issue. */


SELECT *,

ROW_NUMBER() OVER(PARTITION BY JOB_ROLE ORDER BY SALARY DESC) AS ROW_NUM ,

RANK() OVER(PARTITION BY JOB_ROLE ORDER BY SALARY DESC) AS RANK_ROW

FROM EMPLOYEE;


/* The DENSE_RANK() function is similar to the RANK() except for one difference, it doesn't skip any
ranks when ranking rows
```

Here, all the ranks are distinct and sequentially increasing within each partition.

As compared to the RANK() function, it has not skipped any rank within a partition. */

```
SELECT * FROM
(
SELECT *,
ROW_NUMBER() OVER(PARTITION BY JOB_ROLE ORDER BY SALARY) AS ROW_NUM ,
RANK() OVER(PARTITION BY JOB_ROLE ORDER BY SALARY) AS RANK_ROW,
DENSE_RANK() OVER(PARTITION BY JOB_ROLE ORDER BY SALARY) AS DENSE_RANK_ROW
FROM EMPLOYEE
)
WHERE DENSE_RANK_ROW <=2;

SELECT *,
ROW_NUMBER() OVER(PARTITION BY JOB_ROLE ORDER BY SALARY) AS ROW_NUM ,
RANK() OVER(PARTITION BY JOB_ROLE ORDER BY SALARY) AS RANK_ROW,
DENSE_RANK() OVER(PARTITION BY JOB_ROLE ORDER BY SALARY) AS DENSE_RANK_ROW ,
NTH_VALUE(SALARY,2) OVER(PARTITION BY JOB_ROLE ORDER BY SALARY ) AS
SECOND_HIGHEST_SAL_JOB_ROLE
FROM EMPLOYEE ;
```

WINDOW FUNCTIONS PART 2

----------WINDOW FUNCTIONS----------

-- SYNTAX : window_function_name(<exprsn>) OVER (<partition_by_clause> <order_clause>)

--- display total salary based on job profile

```
SELECT JOB_ROLE,SUM(SALARY) FROM EMPLOYEE
GROUP BY 1
ORDER BY 2 DESC;
```

-- display total salary along with all the records ()every row value

```
SELECT * , SUM(SALARY) OVER() AS TOT_SALARY,
ROUND(SALARY / TOT_SALARY * 100,2) AS PERC_CONTRIBUTION
FROM EMPLOYEE
ORDER BY PERC_CONTRIBUTION DESC;
```

-- display the MAX salary per job category for all the rows

```
SELECT *,
MAX(SALARY) OVER(PARTITION BY JOB_ROLE) AS MAX_JOB_SAL,
```

WINDOW FUNCTIONS PART 2

```
MIN(SALARY) OVER(PARTITION BY JOB_ROLE) AS MIN_JOB_SAL,

ROUND(SALARY / MAX_JOB_SAL * 100,2) AS MAX_PERC_CONTRIBUTION_JOB_ROLE,

ROUND(SALARY / MIN_JOB_SAL * 100,2) AS MIN_PERC_CONTRIBUTION_JOB_ROLE

FROM EMPLOYEE;

--ORDER BY PERC_CONTRIBUTION_JOB_ROLE DESC;


select *,max(salary) over(partition by JOB_ROLE) as MAX_SAL ,

min(salary) over(partition by JOB_ROLE) as MIN_SAL,

SUM(salary) over(partition by JOB_ROLE) as TOT_SAL

from Employee;




--ARRANGING ROWS WITHIN EACH PARTITION BASED ON SALARY IN DESC ORDDER

select *,max(salary) over(partition by JOB_ROLE ORDER BY SALARY DESC) as MAX_SAL ,

min(salary) over(partition by JOB_ROLE ORDER BY SALARY ASC) as MIN_SAL,

SUM(salary) over(partition by JOB_ROLE) as TOT_SAL

from Employee

ORDER BY JOB_ROLE;


-- ROW_NUMBER() It assigns a unique sequential number to each row of the table ...

--SELECT * FROM

SELECT * FROM

(SELECT *,ROW_NUMBER() OVER(PARTITION BY JOB_ROLE ORDER BY SALARY DESC) AS
PART_ROW_NUM

FROM EMPLOYEE ) WHERE PART_ROW_NUM <=2;

--)

--WHERE PART_ROW_NUM <=2;
```

WINDOW FUNCTIONS PART 2

/* The RANK() window function, as the name suggests, ranks the rows within their partition based on the given condition.

In the case of ROW_NUMBER(), we have a sequential number.

On the other hand, in the case of RANK(), we have the same rank for rows with the same value.

But there is a problem here. Although rows with the same value are assigned the same rank, the subsequent rank skips the missing rank.

This wouldn't give us the desired results if we had to return "top N distinct" values from a table.

Therefore we have a different function to resolve this issue. */

```
SELECT *,
ROW_NUMBER() OVER(PARTITION BY JOB_ROLE ORDER BY SALARY DESC) AS ROW_NUM ,
RANK() OVER(PARTITION BY JOB_ROLE ORDER BY SALARY DESC) AS RANK_ROW
FROM EMPLOYEE;
```

/* The DENSE_RANK() function is similar to the RANK() except for one difference, it doesn't skip any ranks when ranking rows

Here, all the ranks are distinct and sequentially increasing within each partition.

As compared to the RANK() function, it has not skipped any rank within a partition. */

```
SELECT * FROM
(
SELECT *,
ROW_NUMBER() OVER(PARTITION BY JOB_ROLE ORDER BY SALARY) AS ROW_NUM ,
RANK() OVER(PARTITION BY JOB_ROLE ORDER BY SALARY) AS RANK_ROW,
DENSE_RANK() OVER(PARTITION BY JOB_ROLE ORDER BY SALARY) AS DENSE_RANK_ROW
FROM EMPLOYEE
)
WHERE DENSE_RANK_ROW <=2;
```

```
SELECT *,
```

WINDOW FUNCTIONS PART 2

```
ROW_NUMBER() OVER(PARTITION BY JOB_ROLE ORDER BY SALARY) AS ROW_NUM ,

RANK() OVER(PARTITION BY JOB_ROLE ORDER BY SALARY) AS RANK_ROW,

DENSE_RANK() OVER(PARTITION BY JOB_ROLE ORDER BY SALARY) AS DENSE_RANK_ROW ,

NTH_VALUE(SALARY,2) OVER(PARTITION BY JOB_ROLE ORDER BY SALARY ) AS
SECOND_HIGHEST_SAL_JOB_ROLE

FROM EMPLOYEE ;
```

```
CREATE OR REPLACE TABLE EMPLOYEE
(
   EMPID INTEGER NOT NULL PRIMARY KEY,
   EMP_NAME VARCHAR2(20),
   JOB_ROLE STRING,
   SALARY NUMBER(10,2)
);

INSERT INTO EMPLOYEE
VALUES('101','ANAND JHA','Analyst',50000);

INSERT INTO EMPLOYEE
VALUES(102,'ALex', 'Data Enginner',60000);

INSERT INTO EMPLOYEE
VALUES(103,'Ravi', 'Data Scientist',48000);

INSERT INTO EMPLOYEE
VALUES(104,'Peter', 'Analyst',98000);

INSERT INTO EMPLOYEE
VALUES(105,'Pulkit', 'Data Scientist',72000);

INSERT INTO EMPLOYEE
VALUES(106,'Robert','Analyst',100000);

INSERT INTO EMPLOYEE
VALUES(107,'Rishabh','Data Engineer',67000);

INSERT INTO EMPLOYEE
VALUES(108,'Subhash','Manager',148000);
```

WINDOW FUNCTIONS PART 2

```
INSERT INTO EMPLOYEE
VALUES(109,'Michaeal','Manager',213000);

INSERT INTO EMPLOYEE
VALUES(110,'Dhruv','Data Scientist',89000);

INSERT INTO EMPLOYEE
VALUES(111,'Amit Sharma','Analyst',72000);


SELECT * FROM EMPLOYEE;

update employee set job_role='Data Engineer'
where empid=102;

update employee set salary= 50000
where empid=104;

----------------------------------------------------------------WINDOW FUNCTIONS-----------------------------
------------------------------

-- SYNTAX : window_function_name(<exprsn>) OVER (<partition_by_clause>
<order_clause>)

--- display total salary based on job profile
SELECT JOB_ROLE,SUM(SALARY) FROM EMPLOYEE
GROUP BY 1
ORDER BY 2 DESC;

-- display total salary along with all the records ()every row value
SELECT * , SUM(SALARY) OVER() AS TOT_SALARY,
ROUND(SALARY / TOT_SALARY * 100,2) AS PERC_CONTRIBUTION
FROM EMPLOYEE
ORDER BY PERC_CONTRIBUTION DESC;

-- display the MAX salary per job category for all the rows
SELECT *,
MAX(SALARY) OVER(PARTITION BY JOB_ROLE) AS MAX_JOB_SAL,
MIN(SALARY) OVER(PARTITION BY JOB_ROLE) AS MIN_JOB_SAL,
ROUND(SALARY / MAX_JOB_SAL * 100,2) AS
MAX_PERC_CONTRIBUTION_JOB_ROLE,
ROUND(SALARY / MIN_JOB_SAL * 100,2) AS MIN_PERC_CONTRIBUTION_JOB_ROLE
FROM EMPLOYEE;
--ORDER BY PERC_CONTRIBUTION_JOB_ROLE DESC;

select *,max(salary) over(partition by JOB_ROLE) as MAX_SAL ,
min(salary) over(partition by JOB_ROLE) as MIN_SAL,
SUM(salary) over(partition by JOB_ROLE) as TOT_SAL
from Employee;
```

WINDOW FUNCTIONS PART 2

```
--ARRANGING ROWS WITHIN EACH PARTITION BASED ON SALARY IN DESC
ORDDER
select *,max(salary) over(partition by JOB_ROLE ORDER BY SALARY DESC) as
MAX_SAL ,
min(salary) over(partition by JOB_ROLE ORDER BY SALARY DESC) as MIN_SAL,
SUM(salary) over(partition by JOB_ROLE ORDER BY SALARY DESC) as TOT_SAL
from Employee;

-- ROW_NUMBER() It assigns a unique sequential number to each row of the table ...
SELECT * FROM
(
SELECT *,ROW_NUMBER() OVER(PARTITION BY JOB_ROLE ORDER BY SALARY
DESC) AS PART_ROW_NUM
FROM EMPLOYEE
)
WHERE PART_ROW_NUM <=2;

/* The RANK() window function, as the name suggests, ranks the rows within their partition
based on the given condition.
   In the case of ROW_NUMBER(), we have a sequential number.
   On the other hand, in the case of RANK(), we have the same rank for rows with the same
value.
But there is a problem here. Although rows with the same value are assigned the same rank,
the subsequent rank skips the missing rank.
This wouldn't give us the desired results if we had to return "top N distinct" values from a
table.
Therefore we have a different function to resolve this issue. */

SELECT *,ROW_NUMBER() OVER(PARTITION BY JOB_ROLE ORDER BY SALARY
DESC) AS ROW_NUM ,
RANK() OVER(PARTITION BY JOB_ROLE ORDER BY SALARY DESC) AS RANK_ROW
FROM EMPLOYEE;

/* The DENSE_RANK() function is similar to the RANK() except for one difference, it doesn't
skip any ranks when ranking rows
Here, all the ranks are distinct and sequentially increasing within each partition.
As compared to the RANK() function, it has not skipped any rank within a partition. */

SELECT * FROM
(
SELECT *,ROW_NUMBER() OVER(PARTITION BY JOB_ROLE ORDER BY SALARY) AS
ROW_NUM ,
RANK() OVER(PARTITION BY JOB_ROLE ORDER BY SALARY) AS RANK_ROW,
DENSE_RANK() OVER(PARTITION BY JOB_ROLE ORDER BY SALARY) AS
DENSE_RANK_ROW
FROM EMPLOYEE
)
WHERE DENSE_RANK_ROW <=2;
```

```
CREATE OR REPLACE TABLE AJ_sales (
  customer_id VARCHAR(1),
  order_date DATE,
  product_id INTEGER
);

INSERT INTO AJ_sales
  (customer_id, order_date, product_id)
VALUES
  ('A', '2021-01-01', '1'),
  ('A', '2021-01-01', '2'),
  ('A', '2021-01-07', '2'),
  ('A', '2021-01-10', '3'),
  ('A', '2021-01-11', '3'),
  ('A', '2021-01-11', '3'),
  ('B', '2021-01-01', '2'),
  ('B', '2021-01-02', '2'),
  ('B', '2021-01-04', '1'),
  ('B', '2021-01-11', '1'),
  ('B', '2021-01-16', '3'),
  ('B', '2021-02-01', '3'),
  ('C', '2021-01-01', '3'),
  ('C', '2021-01-01', '3'),
  ('C', '2021-01-07', '3');
```

```
SELECT * FROM AJ_sales;


CREATE OR REPLACE TABLE AJ_menu (
  product_id INTEGER,
  product_name VARCHAR(5),
  price INTEGER
);

INSERT INTO AJ_menu
  (product_id, product_name, price)
VALUES
  ('1', 'sushi', '10'),
  ('2', 'curry', '15'),
  ('3', 'ramen', '12');



CREATE OR REPLACE TABLE AJ_members (
  customer_id VARCHAR(1),
  join_date DATE
);

INSERT INTO AJ_members
  (customer_id, join_date)
VALUES
  ('A', '2021-01-07'),
```

('B', '2021-01-09');

--What was the first item from the menu purchased by each customer?

```
SELECT * FROM
(SELECT
customer_id,product_name,order_date,
ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY order_date) AS rank
FROM AJ_SALES AS s
JOIN AJ_MENU AS m ON s.product_id = m.product_id) WHERE rank = 1;
```

--What is the most purchased item on the menu and how many times was it purchased by all customers?

```
SELECT product_name,
COUNT(product_name) AS total_purchase_quantity
--row_number() OVER(ORDER BY) AS rank
FROM AJ_sales AS s
JOIN AJ_MENU AS m ON s.product_id = m.product_id
GROUP BY 1
ORDER BY 2 DESC;
```

--Which item was the most popular for each customer?

```
SELECT
  customer_id,
  product_name,
  COUNT(product_name) AS total_purchase_quantity
```

```
FROM
  AJ_SALES AS s
  INNER JOIN AJ_MENU AS m ON s.product_id = m.product_id
GROUP BY 1,2
ORDER BY 3 DESC;
```

---or--------

```
SELECT
      customer_id,
      product_name,
      COUNT(product_name) AS total_purchase_quantity,
      rank() OVER (PARTITION BY customer_id ORDER BY COUNT(product_name) desc)
AS rank
    FROM
      AJ_SALES AS s
      JOIN AJ_MENU AS m ON s.product_id = m.product_id
    GROUP BY 1,2;
```

WINDOW FUNCTIONS PART 2

```
DROP TABLE IF EXISTS student_score;


CREATE TABLE student_score (
  student_id INT PRIMARY KEY,
  student_name VARCHAR(30),
  dep_name VARCHAR(40),
  score INT
);


INSERT INTO student_score VALUES (11, 'Ibrahim', 'Computer Science', 80);
INSERT INTO student_score VALUES (7, 'Taiwo', 'Microbiology', 76);
INSERT INTO student_score VALUES (9, 'Nurain', 'Biochemistry', 80);
INSERT INTO student_score VALUES (8, 'Joel', 'Computer Science', 90);
INSERT INTO student_score VALUES (10, 'Mustapha', 'Industrial Chemistry', 78);
INSERT INTO student_score VALUES (5, 'Muritadoh', 'Biochemistry', 85);
INSERT INTO student_score VALUES (2, 'Yusuf', 'Biochemistry', 70);
INSERT INTO student_score VALUES (3, 'Habeebah', 'Microbiology', 80);
INSERT INTO student_score VALUES (1, 'Tomiwa', 'Microbiology', 65);
INSERT INTO student_score VALUES (4, 'Gbadebo', 'Computer Science', 80);
INSERT INTO student_score VALUES (12, 'Tolu', 'Computer Science', 67);


--Say for instance you want to compare the minimum score and maximum score from all the records in the table we created earlier.
--You can do that using a window function as shown below.
-- Remember that not specifying a partition clause in the OVER clause will cause all the windows to span through the entire dataset.


SELECT *,
MAX(score) OVER() AS maximum_score,
```

```
MIN(score) OVER() AS minimum_score

FROM student_score;


--Also, note that the above query can be also achieved using subqueries like this:


SELECT *,

        (SELECT MAX(score) FROM student_score) AS maximum_score,

        (SELECT MIN(score) FROM student_score) AS minimum_score

FROM student_score;


--For example, say you want to compare the maximum score and average score in each department with the individual score.

--You can do this by specifying the PARTITION BY clause in the OVER statement and also use it with the aggregate function you want ----to use to achieve your desired result.


SELECT *,

MAX(score)OVER(PARTITION BY dep_name) AS dep_maximum_score,

ROUND(AVG(score)OVER(PARTITION BY dep_name), 2) AS dep_average_score

FROM student_score;



SELECT *,

ROW_NUMBER() OVER(ORDER BY student_name) AS name_serial_number,

RANK()OVER(PARTITION BY dep_name ORDER BY score DESC) AS dept_wise_rank,

DENSE_RANK()OVER(PARTITION BY dep_name ORDER BY score DESC)     AS dept_wise_dense_rank,

LAG(score) OVER(PARTITION BY dep_name ORDER BY score)AS LAG_DPT_WISE,

LEAD(score) OVER(PARTITION BY dep_name ORDER BY score)AS LEAD_DPT_WISE

FROM student_score;
```

WINDOW FUNCTIONS PART 2

--Since you want to sum up all rows before the current row and the current row itself, you can use the UNBOUNDED PRECEDING keyword. ---Remember that this gets all rows before the current row and also uses the current row itself.

```
SELECT *,

SUM(score)OVER(ORDER BY student_id ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS cummulative_sum

FROM student_score;
```

--The first row in the dataset does not have any row before it. But since we also specify the CURRENT ROW keyword as the last frame, then the SQL engine finds its sum which equals 65.

--Then moving to the second row. It has 1 row before it. So the SQL engine sums the score of the first row 65 with the current row which is 70. That is why the result is 135...and so on down the table.