**Windows Functions**

1. What is SQL Aggregate Function

-- An aggregate function performs calculation on a set of values to return a single value.

-- Aggregate functions are usually used with the GROUP BY clause of the SELECT statement.

-- HAVING clause is used to write conditions on the value that is returned by the aggregate function.

-- Aggregate functions return the same value each time that it is called with a specific set of input values on same dataset.

-- For example,

**-- Fetch total salary distribution for each department**

SELECT

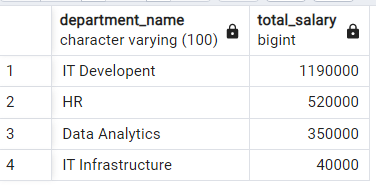
dept.department\_name, SUM(emp.salary) as total\_salary

FROM

dev\_schema.employee as emp INNER JOIN dev\_schema.department as dept

ON emp.fk\_department\_id = dept.department\_id

GROUP BY dept.department\_name ORDER BY total\_salary DESC;



-- The above example is to get sum of salaries by each department. It uses a simple aggregate function which is sum(salary) output by single column grouping.

-- We can add multiple columns in the select statement as long as those columns are included in the group by clause as shown below.

**-- Fetch total salary distribution for each department along with the department id**

SELECT

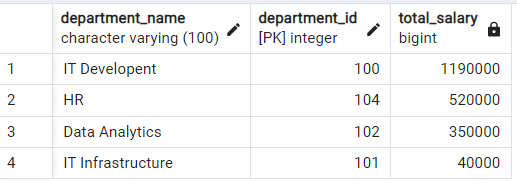
dept.department\_name, dept.department\_id, SUM(emp.salary) as total\_salary

FROM

dev\_schema.employee as emp INNER JOIN dev\_schema.department as dept

ON emp.fk\_department\_id = dept.department\_id

GROUP BY dept.department\_name, dept.department\_id ORDER BY total\_salary DESC;



-- The above query uses two field for grouping the data and for each group the query return a single row as the result.

-- If we try to select other columns that are not in the group by clause, the aggregate function query will give error.

**-- Fetch total salary distribution for each department along with the department id**

SELECT

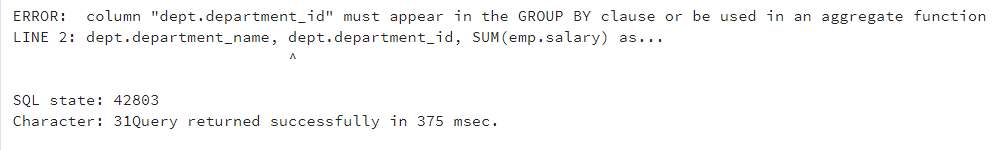
dept.department\_name, dept.department\_id, SUM(emp.salary) as total\_salary

FROM

dev\_schema.employee as emp INNER JOIN dev\_schema.department as dept

ON emp.fk\_department\_id = dept.department\_id

GROUP BY dept.department\_name ORDER BY total\_salary DESC;



-- Here the Aggregate functions worked on a set of rows to return a single result value. If we need only a high-level summary on the data the aggregate functions are useful.

1. What is SQL Window Function

-- The SQL Window Function calculates an aggregate value based on a group of table records called window frame and return multiple rows for each group.

-- FROM clause in the Query filters the data and those virtual table data are considered in a window function for operations.

-- Multiple window functions can slice up the data in different ways by using OVER clauses.

-- We can specify the window frame partition by using PARTITION BY clauses.

-- When we have the OVER clause with an empty parameter, it will operate on the whole set of rows.

-- Below example does calculations sum, average, min, max and percentage using the whole set of rows:

**-- Using empty over()**

SELECT

salary,

SUM(salary) over() as total\_salary,

AVG(salary) over() as avg\_salary,

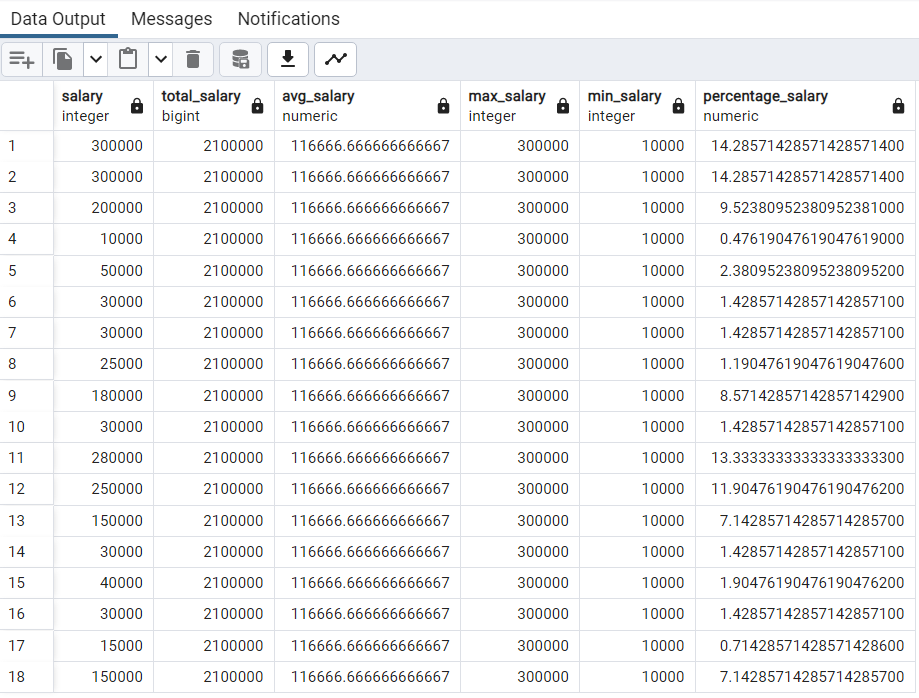
MAX(salary) over() as max\_salary,

MIN(salary) over() as min\_salary,

(salary::numeric/sum(salary) over()) \* 100 as percentage\_salary

FROM

dev\_schema.employee ORDER BY total\_salary DESC;



-- When we expand the above example by adding PARTITION in the OVER clause, we reduce the window frame for the column value group specified in the PARTITION column.

-- Below example does calculations sum, average, min, max and percentage for a set of window frame for each department partition.

-- We can add running total of sales in the above example by adding ORDER BY clause in the window frame.

-- It orders the rows within a window frame partition.

-- Using partition clause with over()

SELECT

dept.department\_name,

emp.salary,

SUM(emp.salary) over(PARTITION BY dept.department\_id ORDER BY emp.salary DESC) as total\_salary\_as\_per\_department,

AVG(emp.salary) over(PARTITION BY dept.department\_id) as avg\_salary\_as\_per\_department,

MAX(emp.salary) over(PARTITION BY dept.department\_id) as max\_salary\_as\_per\_department,

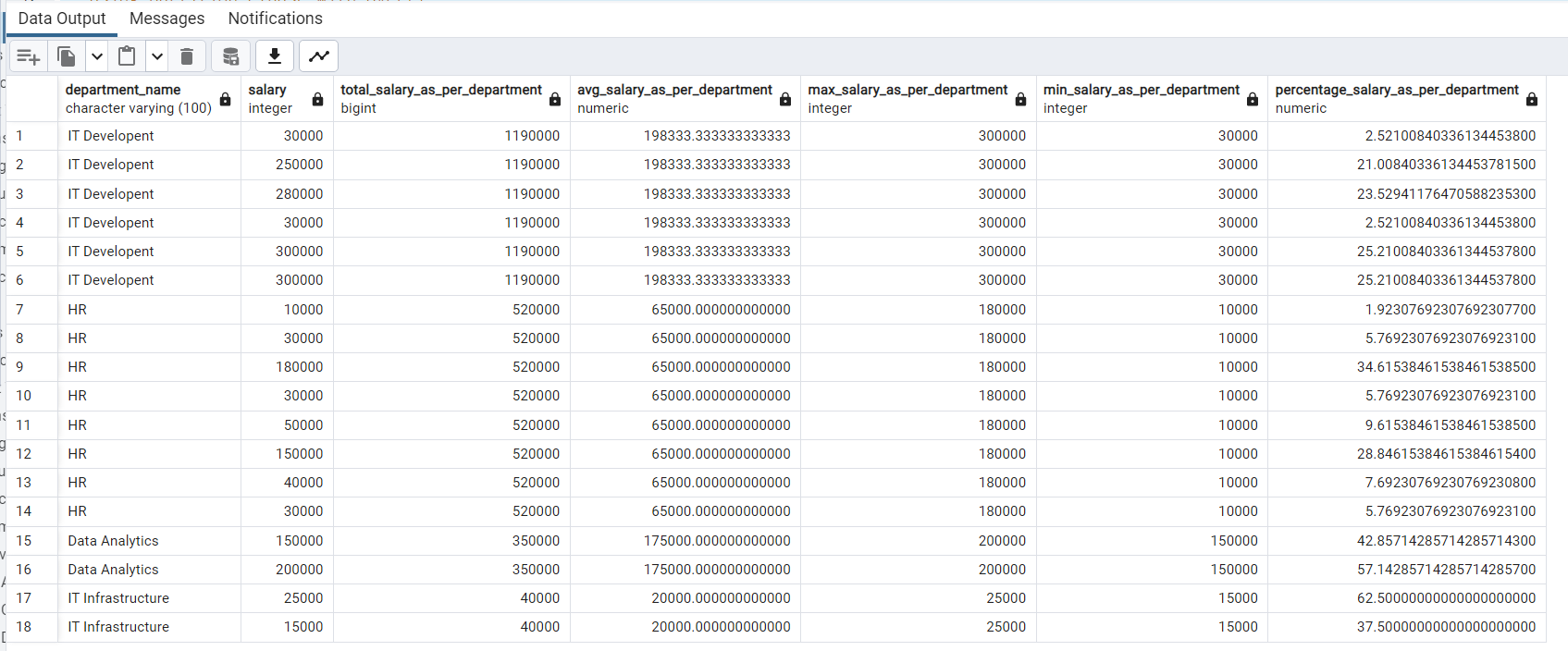
MIN(emp.salary) over(PARTITION BY dept.department\_id) as min\_salary\_as\_per\_department,

(emp.salary::numeric/sum(emp.salary) over(PARTITION BY dept.department\_id)) \* 100 as percentage\_salary\_as\_per\_department

FROM

dev\_schema.employee as emp INNER JOIN dev\_schema.department as dept

ON emp.fk\_department\_id = dept.department\_id;



**-- Write a sql query to fetch employee id, employee name, employee salary, employee department name, employee job title, And max salary as per the department.**

SELECT emp.employee\_id, emp.first\_name, emp.last\_name, emp.salary, dept.department\_name, job.job\_title,

MAX(emp.salary) OVER (PARTITION BY dept.department\_id ORDER BY emp.salary DESC) AS max\_salary\_per\_department

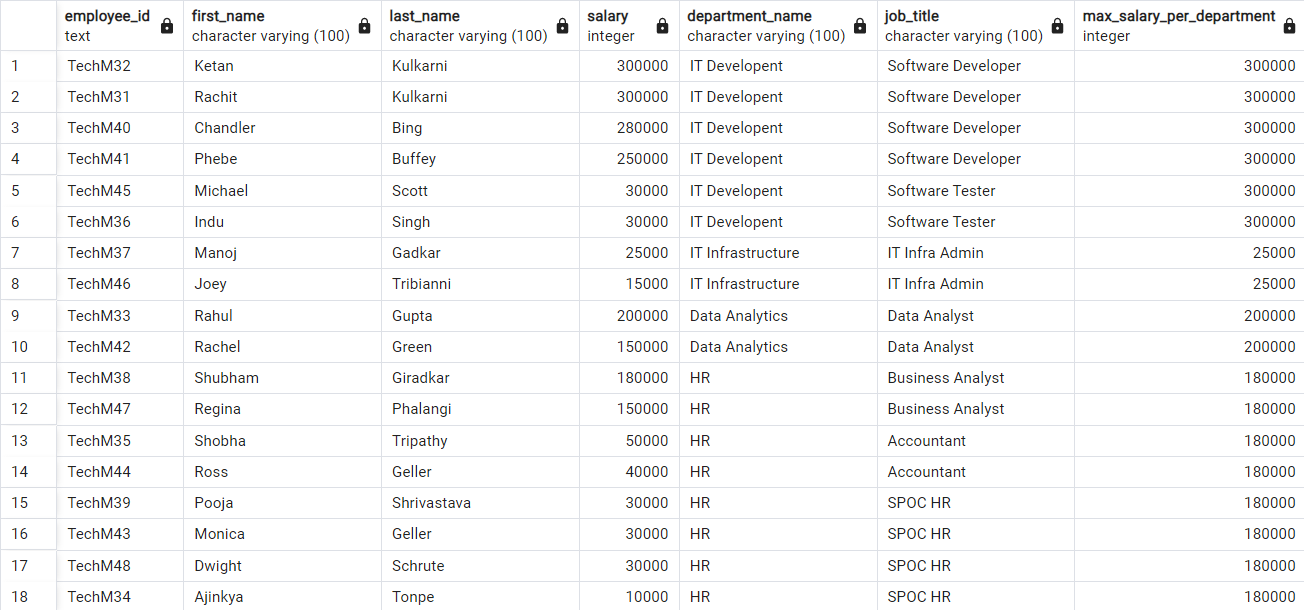
FROM

dev\_schema.employee as emp INNER JOIN dev\_schema.department as dept

ON emp.fk\_department\_id = dept.department\_id

INNER JOIN dev\_schema.job as job

ON emp.fk\_job\_id = job.job\_id;



**-- Write a sql query to fetch employee id, employee name, employee salary, employee department name, employee job title, And max salary as per the department along with the count of employees per department**

SELECT emp.employee\_id, emp.first\_name, emp.last\_name, emp.salary, dept.department\_name, job.job\_title,

MAX(emp.salary) OVER (PARTITION BY dept.department\_id ORDER BY emp.salary DESC) AS max\_salary\_per\_department,

COUNT(\*) OVER (PARTITION BY dept.department\_id) as total\_no\_of\_emp\_per\_dept

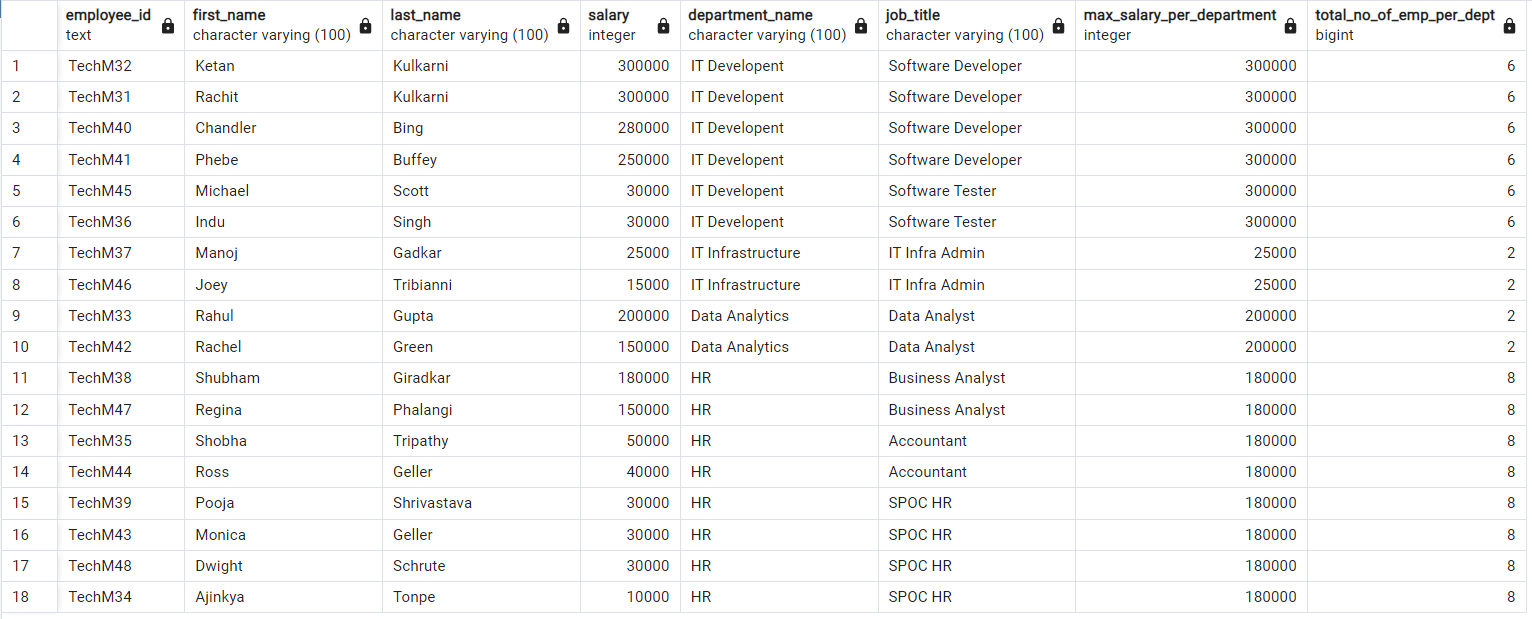
FROM

dev\_schema.employee as emp INNER JOIN dev\_schema.department as dept

ON emp.fk\_department\_id = dept.department\_id

INNER JOIN dev\_schema.job as job

ON emp.fk\_job\_id = job.job\_id;



**1] Rank():**

-- In PostgreSQL, the RANK () function is used to assign a rank to each row of the query result set within the specified partition.

-- The rank of the first row within each partition is 1.

**Syntax:**

RANK() OVER (

[PARTITION BY partition\_expression, ... ]

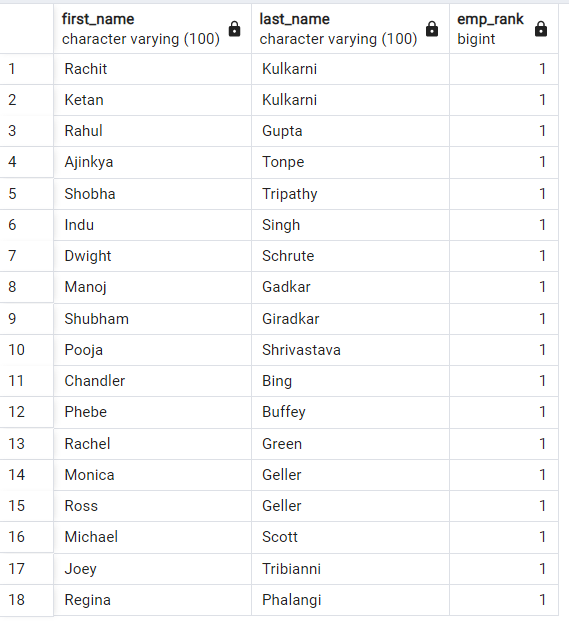
ORDER BY sort\_expression [ASC | DESC], ...

)

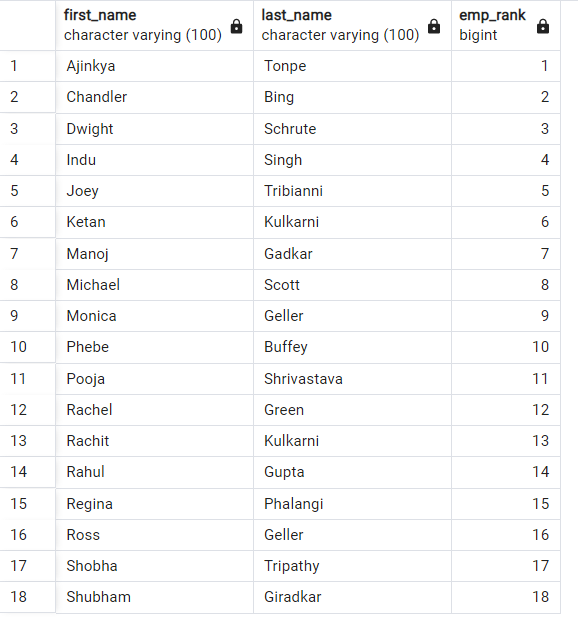
-- Let’s analyse the above syntax:

* First, the PARTITION BY clause creates partition in the query result where the RANK() function is applied.
* The ORDER BY clause sets the order in which the query results are displayed.

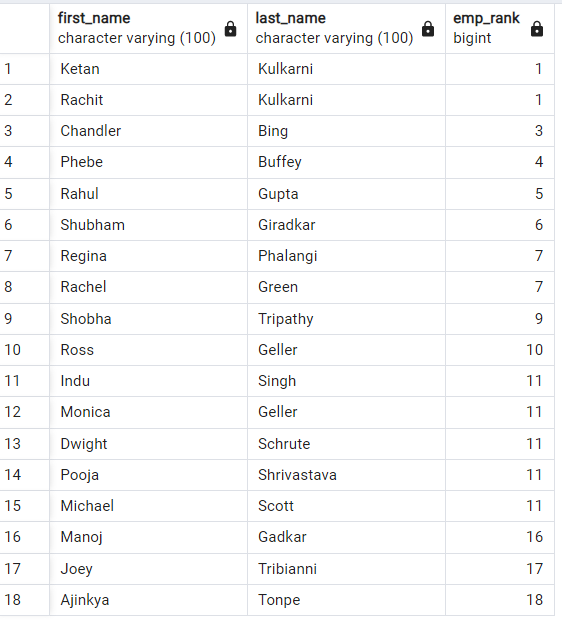
-- SELECT first\_name, last\_name, RANK() OVER () emp\_rank FROM dev\_schema.employee;



-- SELECT first\_name, last\_name, RANK() OVER (ORDER BY first\_name) emp\_rank FROM dev\_schema.employee;

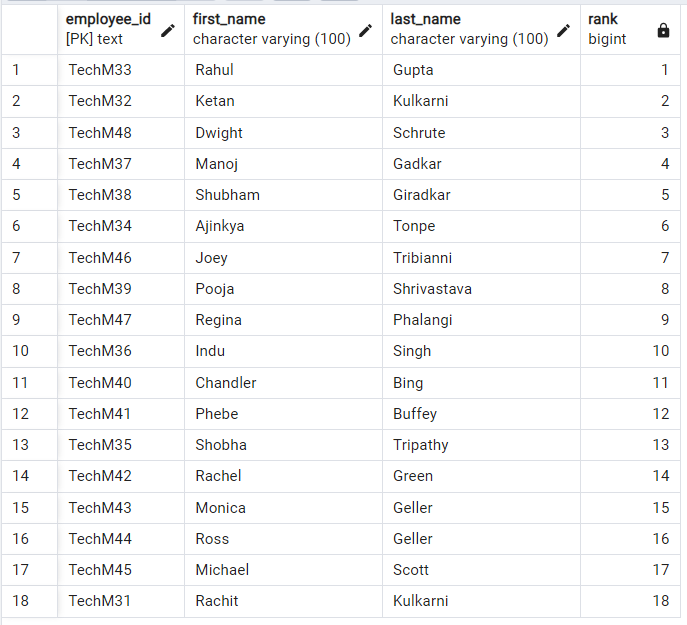


-- SELECT first\_name, last\_name, RANK() OVER (ORDER BY Salary DESC) emp\_rank FROM dev\_schema.employee;



**-- Assign the rank to each employee based on their joining date in ascending order**

SELECT employee\_id, first\_name, last\_name, RANK() OVER (ORDER BY created\_at ASC) FROM dev\_schema.employee;



**2] DESNSE\_RANK():**

-- The PostgreSQL dense\_rank() function returns the rank within the partition in which the current row is located, starting at 1, with no gap.

-- That is, the same value has the same rank, but the rank of the next different value increases in order. For example, if there are 2 first places, then the rank of third place is 2. This is different from the rank() function.

-- Here is the syntax of the PostgreSQL dense\_rank() function:

dense\_rank()

OVER (

[PARTITION BY partition\_column\_list]

[ORDER BY order\_column\_list]

)

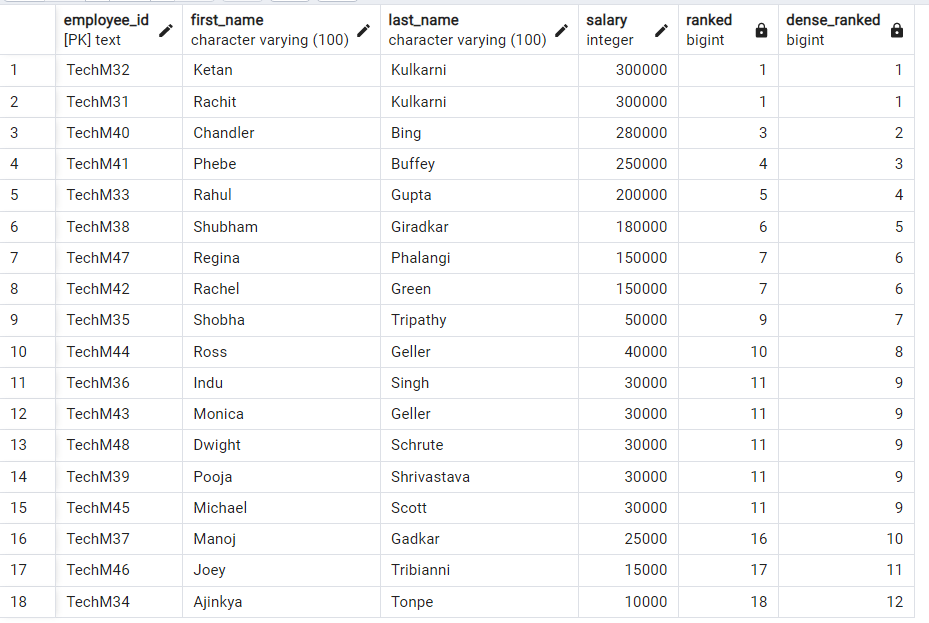
**-- Assign the rank to each employee based on their salary in descending order**

SELECT employee\_id, first\_name, last\_name, salary,

RANK() OVER (ORDER BY salary DESC) AS ranked,

DENSE\_RANK() OVER (ORDER BY salary DESC) dense\_ranked

FROM dev\_schema.employee;



**-- Write a SQL query to find the employees with the second highest salary**

SELECT employee\_id, first\_name, last\_name, salary FROM

(

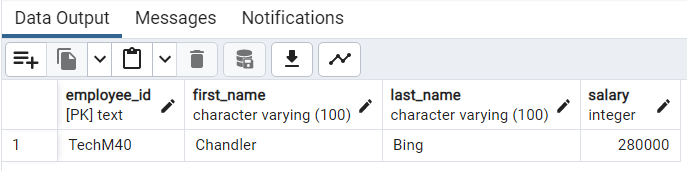
SELECT employee\_id, first\_name, last\_name, salary,

DENSE\_RANK() OVER (ORDER BY salary DESC) AS rk

FROM dev\_schema.employee

) AS subq

WHERE rk = 2;



**3] LAG():**

-- In PostgreSQL, the LAG() function is used to access a row that comes exactly before the current row at a specific physical offset.

-- The LAG() comes in handy while comparing the values of the current row with the previous row.

-- **Syntax:**

LAG(expression [, offset [, default\_value]])

OVER (

[PARTITION BY partition\_expression, ... ]

ORDER BY sort\_expression [ASC | DESC], ...

)

-- The following statement uses the PostgreSQL lag() function to add a next\_quarter\_revenue column for comparing the revenue of the current quarter and the next quarter:

SELECT \*,

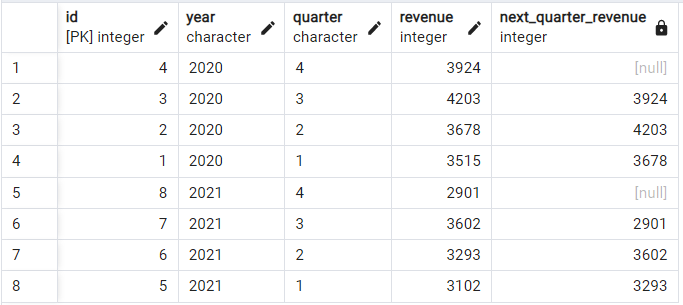
lag(revenue, 1) OVER (

PARTITION BY year

ORDER BY quarter DESC

) next\_quarter\_revenue

FROM tax\_revenue;



-- In the OVER clause,

* The PARTITION BY year partitions all rows by year.
* The ORDER BY quarter DESC sort the rows within each partition in descending order by quarter.
* The lag(revenue, 1) returns the revenue (revenue) from the previous row (1) for within the partition where the current row is located.

-- So the value of the next\_quarter\_revenue column is the revenue of the next quarter of the current row.

-- And the value of the next\_quarter\_revenue column in the first row of each partition is null.

-- You can specify a default value for nulls ​​in the next\_quarter\_revenue column. The following statement is used 0 as a default value:

SELECT \*,

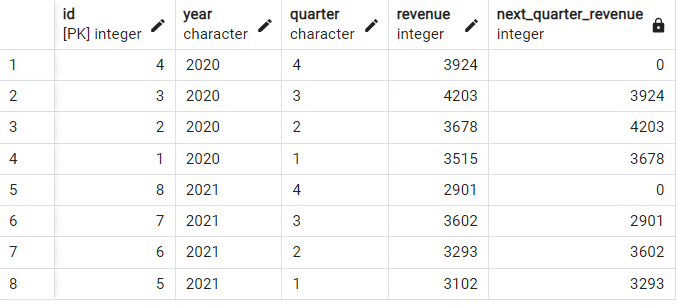
lag(revenue, 1, 0) OVER (

PARTITION BY year

ORDER BY quarter DESC

) next\_quarter\_revenue

FROM tax\_revenue;



**4] LEAD():**

-- In PostgreSQL, the LEAD() function is used to access a row that follows the current row, at a specific physical offset and is generally used for comparing the value of the current row with the value of the next row following the current row.

-- **Syntax:**

LEAD(expression [, offset [, default\_value]])

OVER (

[PARTITION BY partition\_expression, ... ]

ORDER BY sort\_expression [ASC | DESC], ...

)

-- The following statement uses a PostgreSQL lead() function to add a last\_quarter\_revenue column to compare current quarter and previous quarter earnings:

SELECT \*,

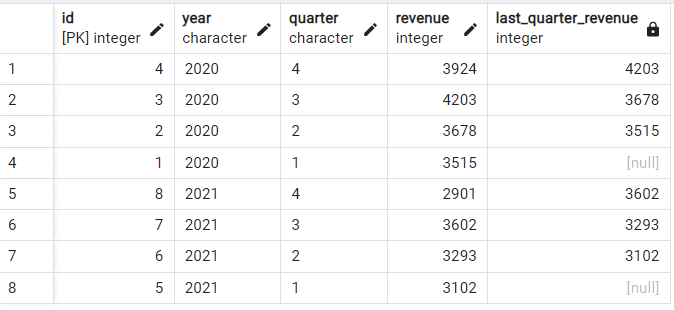
lead(revenue, 1) OVER (

PARTITION BY year

ORDER BY quarter DESC

) last\_quarter\_revenue

FROM tax\_revenue;



**5] ROW\_NUMBER():**

-- In PostgreSQL, the ROW\_NUMBER() function is used to assign a unique integer to every row that is returned by a query.

**Syntax:**

ROW\_NUMBER() OVER(

[PARTITION BY column\_1, column\_2, …]

[ORDER BY column\_3, column\_4, …]

)

* The set of rows on which the ROW\_NUMBER() function operates is called a window.
* The PARTITION BY clause is used to divide the query set results.
* The ORDER BY clause inside the OVER clause is used to set the order in which the query result will be displayed.

SELECT \*,

ROW\_NUMBER() OVER (ORDER BY salary)

FROM dev\_schema.employee;

