Window Functions

_

1. Problem Statement
2. Aggregate Window functions
3. Row_Number()
4. Rank(), Dense_Rank()
5. Ntile()

_

Please note that any topics that are not covered in today's lecture will be covered in the next lecture.

Problem Statement: GlobalTech Solutions

GlobalTech Solutions, a leading technology firm renowned for its innovative solutions and cutting-edge technology, boasts a diverse and talented workforce.

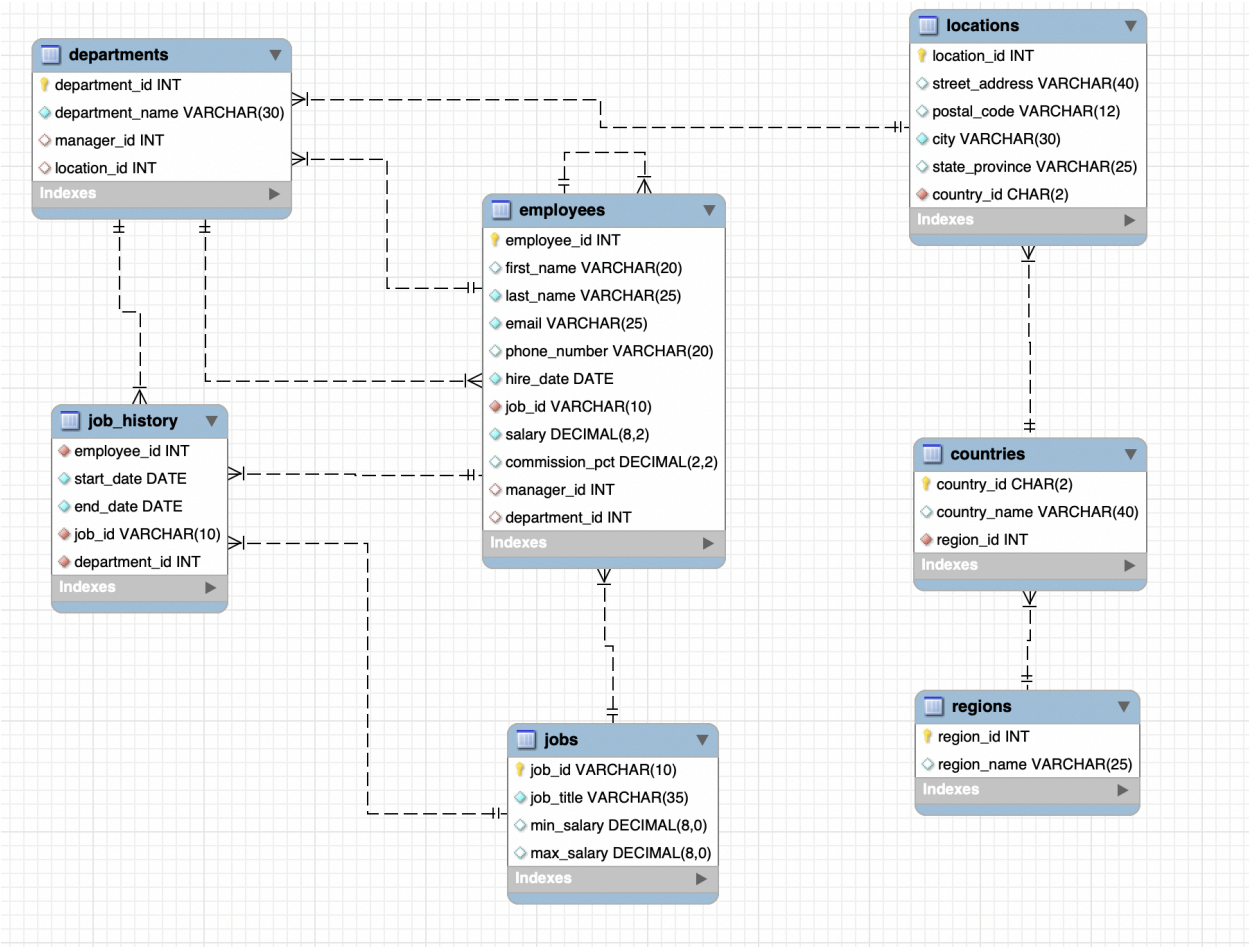
The objective of this project is to conduct an in-depth analysis of employee compensation within the company, using SQL.

This analysis will focus on comparing salaries across various dimensions, including average salaries, salary distributions within departments, and identifying top earners.

The derived insights will help in understanding salary structures, pinpointing any underlying disparities, and assessing salary growth based on job roles within the company.

Dataset: link

Dataset Schema:



_

Disclaimer: In real-world scenarios, learners might not always provide the exact responses anticipated. However, the instructor should guide the conversation strategically to reach the desired learning outcome and introduce the relevant SQL concepts.

The following is a dialogue template for the instructor's reference. This would help motivate/initiate discussion around the topics to be covered in the session :

—— Dialogue Template starts ——

Instructor: Welcome, everyone! Today, we embark on a data analysis journey for employee compensation within our company. By analyzing salary data, we can gain valuable insights and ensure fair compensation practices.

Instructor: Let's begin by understanding the overall salary structure. How can we determine the average salary across the entire company?

Learner: We could potentially calculate the average salary by using an aggregate function that could summarize the average salary for all employees.

Instructor: Precisely! We can use the AVG(Salary) function to calculate the company-wide average salary.

Instructor: Now, with the average in mind, wouldn't it be interesting to identify employees earning more than this average?

Learner: Absolutely! We could potentially filter the data to find those employees, but wouldn't it be more efficient to do it within the query itself?

Instructor: Great thinking! We can leverage filtering functions like WHERE along with the average salary to identify employees exceeding this threshold.

Instructor: This is just the beginning! By mastering various SQL techniques, we can answer more complex questions about employee compensation.

Let’s start our lecture by solving these questions.

—— Dialogue Template ends ——

_

Formulating questions to be explored based on the data provided:

1. Display detailed information on employees earning above the company-wide average salary.
2. Show details of employees whose salaries exceed the average salary of their respective departments.
3. Identify the top 5 highest-earning employees in the company.
4. Fetch details of employees with the highest (or nth highest) salaries in each department.
5. How can we categorize all employees into N different groups (quartiles) based on their salaries?
6. For each employee, compare their salary to the next highest salary in their department, and add a new column showing the difference.
7. Display the Cumulative / Running sum of salaries in each department, ordered by the employees' joining dates.
8. Display the Moving sum/average of salaries in each department, with a window size of N, ordered by the employees' joining dates.
9. Compare each employee's salary with the salaries of the first and last hired employees in their department.

Note: We will cover some of these questions in today's session, while the remaining ones will be addressed in our next class.

#Q: Display detailed information on employees earning above the company-wide average salary.

We can easily solve this using a Subquery.

Query:

```
SELECT *
FROM employees
WHERE salary >
        ( SELECT
          AVG(salary)
          FROM employees )
```

#Q: Show details of employees whose salaries exceed the average salary of their respective departments.

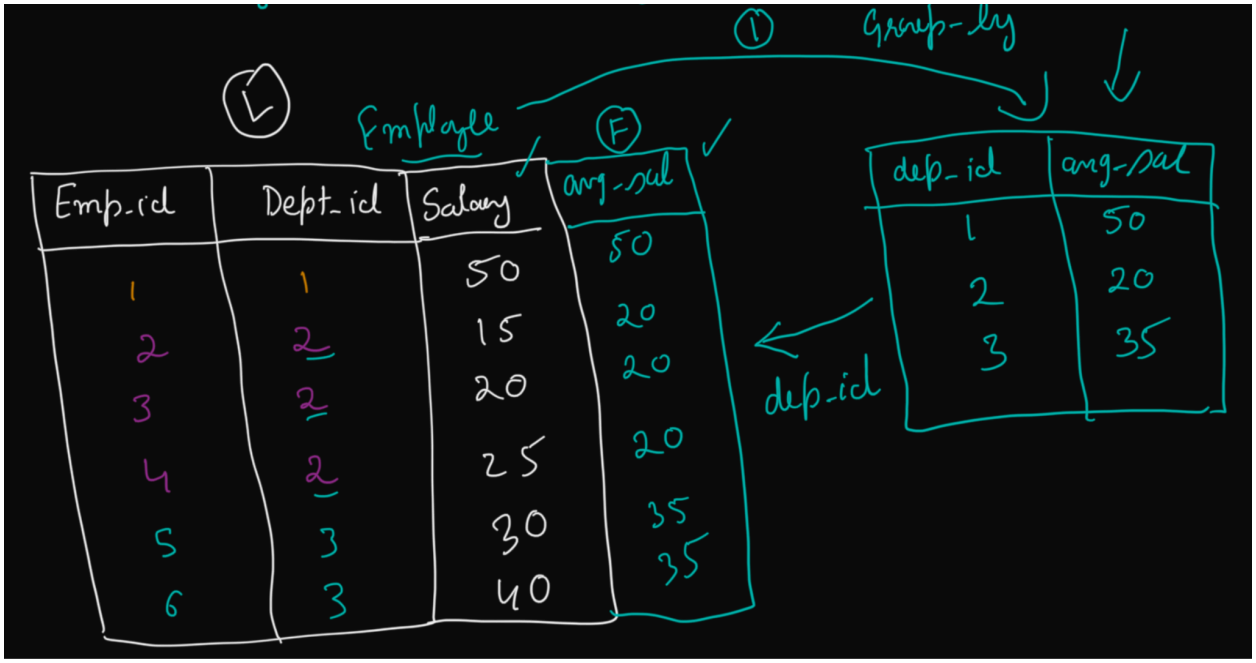
To do this comparison we need an adjacent column `avg_sal` in our table.

What does Group By do to the output? - Groups / Collapses rows.

- All the functions covered, like ROUND(), **return one value in each row** of the results dataset.
- When **GROUP BY is used, the functions operate on multiple values in an aggregated group of records**, summarizing across multiple rows in the underlying dataset, like AVG(), but each value returned is associated with a single row in the results.
- The output is always grouped.

One way to solve this would be -

If you apply a grouping operation to derive a new table, you will obtain a result set with department IDs and their corresponding average salaries.
Now, all that remains is to join both tables on the department_id.



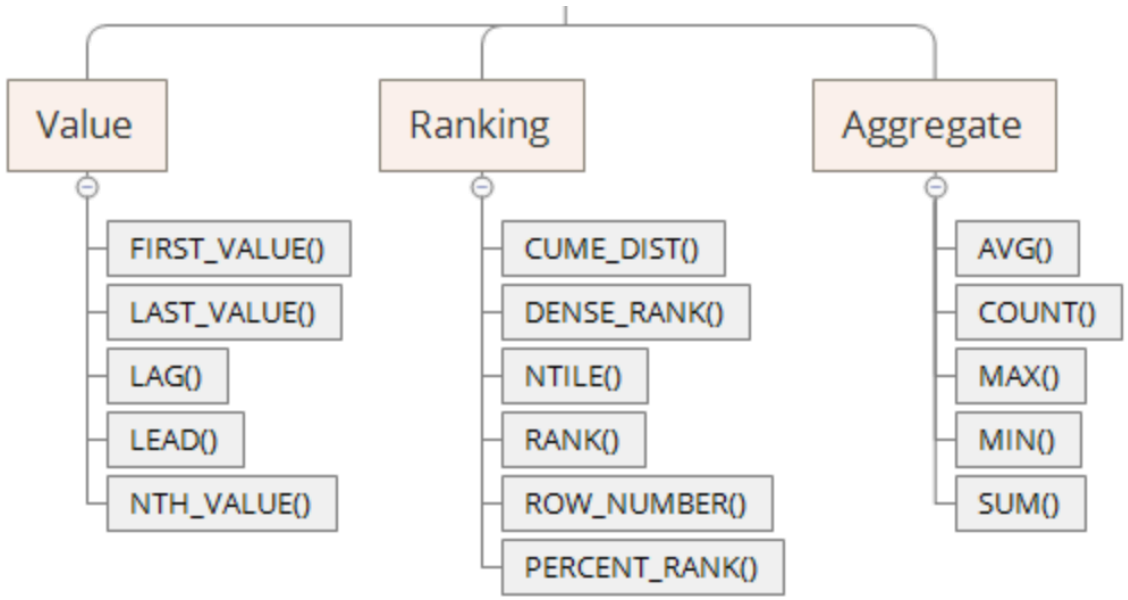
Query:

```
SELECT
 employee_id,
 first_name, last_name,
 e.department_id,
 salary, avg_sal
FROM (
 SELECT
  department_id,
  ROUND(AVG(salary)) AS avg_sal
 FROM hr.employees
 GROUP BY department_id) d
JOIN hr.employees e
ON d.department_id = e.department_id;
```

_

Transitioning -> What if we want to operate on multiple rows but don't want the records to be grouped in the output?

Window Functions



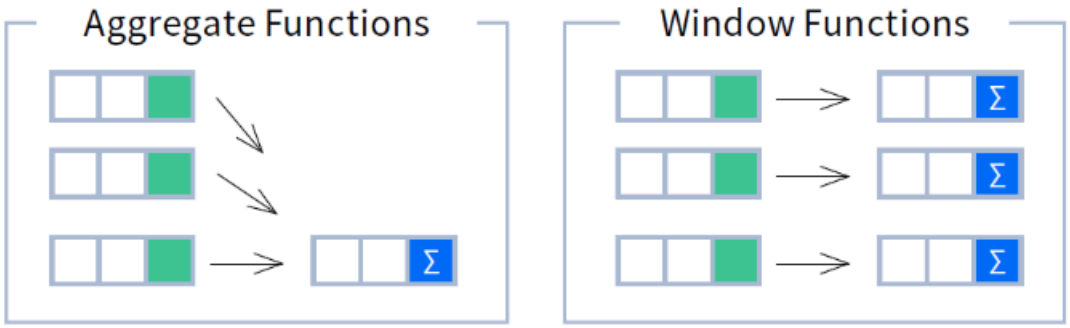
Window function gives the ability to put the values from one row of data into context compared to a group of rows or partitions.

It's important to note that **CUME_DIST**() and **PERCENT_RANK**() will not be part of our curriculum as they are beyond the current scope.

We can answer questions like:

- Where would this row land in the results if the dataset were sorted? - Rank
- How does a value in this row compare to the prior row? - Accessing the preceding / following row.
- How does a current row value compare to the group or partition(in the window function's context) average value?

So, window functions **return group aggregate calculations alongside individual row-level** information for items in that group or partition.



Syntax:

```
SELECT
  window_fn_name (expression)
OVER (
  [PARTITION BY partition_expression, ... ]
  ORDER BY sort_expression [ASC | DESC], ...
) AS col_alias
FROM [table_name]
```

- **OVER() -** It tells the DBMS to apply the function over a window of rows.
- **PARTITION BY -** This clause divides the result set into partitions or groups based on the specified columns.

- **ORDER BY -** This clause orders the rows within each partition based on the specified columns and their sort order.

_

Aggregate Window Functions

Can we use aggregate functions as window functions? Yes!

- This capability is often referred to as "windowed aggregate functions" or "aggregate window functions."
- Window functions operate on a set of rows related to the current row, whereas aggregate functions operate on a set of rows and return a single value.
- You can use aggregate functions like SUM(), AVG(), COUNT(), etc., as window functions by combining them with the OVER() clause.

#Q: Display the average salary of all employees in a new column in the employees table.

Here, we do not want to lose the original data, but we want a new column with the aggregated data.

Query:

```
SELECT
 employee_id,
 department_id,
 salary,
 AVG(salary) OVER() AS avg_salary
FROM hr.employee;
```

#Q: How can we display the average salary of employees in each department by adding a new column next to the employees in their respective departments?

Query:

```
SELECT
 employee_id,
 department_id,
 salary,
 AVG(salary) OVER(PARTITION BY department_id) AS avg_salary
FROM hr.employee;
```

_

Let's answer the questions that we discussed at the beginning of this session -

#Q: Display detailed information on employees earning above the company-wide average salary.

Query:

```
SELECT *
 FROM
 (SELECT
  employee_id,
  department_id,
  salary,
  AVG(salary) OVER() AS avg_salary
 FROM hr.employee) as tbl
WHERE salary > avg_salary;
```

#Q: Show details of employees whose salaries exceed the average salary of their respective departments.

Query:

```
SELECT *
 FROM
 (SELECT
  employee_id,
  department_id,
  salary,
  ROUND(AVG(salary) OVER(PARTITION BY department_id)) AS
avg_salary
  FROM hr.employee) tbl
WHERE salary > avg_salary;
```

_

Quiz - 1

```
Q. Assume a table of 5 rows with 2 unique categories in the category
column and sales as another column.
How many rows do we get if
1. Groupby on the category and sum(sales)
2. Sum(sales) using window functions

   a) Groupby - 5, Window_fn. - 2
   b) Groupby - 3, Window_fn. - 3
   c) Groupby - 2, Window_fn. - 5 - correct
   d) Groupby - 3, Window_fn. - 4
```

_

ROW_NUMBER

#Q: Identify the top 5 highest-earning employees in the company.

You might have solved such questions earlier using ORDER BY and LIMIT clauses.

But what if 2 employees have the exact same salaries?
In that case, we need to come up with a way to rank the data.

We need a function that allows us to **rank rows by a value** - **ROW_NUMBER**().

Syntax:

```
ROW_NUMBER() OVER (<partition_definition> <order_definition>)
```

Here, the highest salary will get row number 1 and so on...

Query:

```
SELECT *
 FROM
 (SELECT
   employee_id,
```

```
  department_id,
  salary,
  ROW_NUMBER() OVER(ORDER BY salary DESC) AS row_num
 FROM hr.employee) x
WHERE
  x.row_num <= 5;
```

You'll notice that the preceding query has a different structure than the queries we have written so far.

- Here, we use a **subquery** or **derived table**. This concept involves embedding one query inside another.
- The inner query calculates the ROW_NUMBER(), and we treat its result as a temporary table, aliased as **x**.
- The outer query then selects from this derived table and applies the filter.

Why can't we just use the WHERE clause in the main query?

When you write SQL queries involving window functions, you might encounter a need to filter based on the results of these functions.

However, if you attempt to filter on the results of a window function directly within the main query, you will encounter errors.

This is due to the order in which SQL processes queries.

- **Order of Execution**:
    - SQL evaluates the WHERE clause before it processes the SELECT clause, where window functions are computed.
    - At the time the WHERE clause is applied, the values calculated by the window function (e.g., ROW_NUMBER()) are not yet available.
- **Window Function Computation**:
    - Window functions like ROW_NUMBER() need to evaluate the entire dataset to assign rankings.
    - This evaluation happens after the initial filtering (done by the WHERE clause) but before the final result set is returned.

Using a subquery allows the window function results to be calculated first, ensuring the WHERE clause can then filter based on these results.

_

Quiz - 2

```
Q. When have not mentioned any PARTITION BY then the entire table is considered as one partition?

   a) True - correct
   b) False
```

_

Transitioning ->

The problem with the output in **ROW_NUMBER()** is that even for the same values, we are getting different numbers or ranks but what if you want the same rank to be assigned to the same values.

RANK and DENSE_RANK

Continuing with the same question...

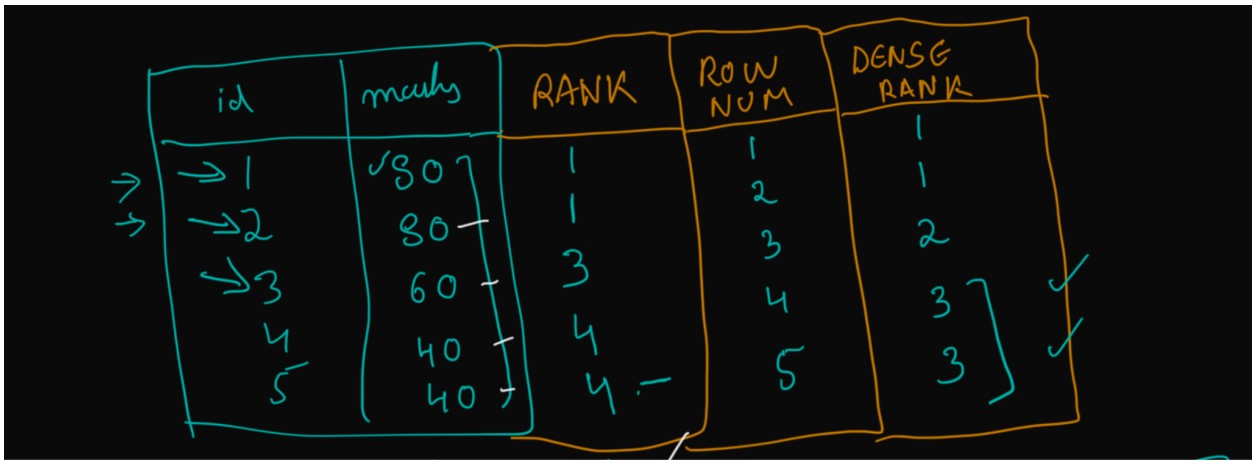
To return the **top 5 earning employees** when there is **more than one employee having the same salary**, we use the **RANK** function**.**

The **RANK** function numbers the results just like **ROW_NUMBER** does, but assigns the same ranking to rows with identical values.

There's another function called **DENSE_RANK** which although similar but is slightly different from RANK. While both functions provide rankings for result sets, **DENSE_RANK does not leave gaps in ranking**.

We'll understand the difference between these 3 functions using the illustration given below.

Illustration ->



_

Replacing ROW_NUMBER with RANK/DENSE_RANK in the original query -

Query:

```
SELECT
 employee_id,
 department_id,
 salary,
```

```
 ROW_NUMBER() OVER(ORDER BY salary DESC) AS row_num,
 RANK() OVER(ORDER BY salary DESC) AS sal_rank,
 DENSE_RANK() OVER(ORDER BY salary DESC) AS sal_dense_rank
FROM hr.employee
ORDER BY
 sal_rank,
 sal_dense_rank,
 row_num;
```

#Q: Fetch details of employees with the highest salaries in each department.

Query:

```
SELECT *
 FROM
 (SELECT
   employee_id,
   department_id,
   salary,
   ROW_NUMBER() OVER(PARTITION BY department_id ORDER BY
salary DESC) AS row_num,
   RANK() OVER(PARTITION BY department_id ORDER BY salary DESC)
AS rank,
  DENSE_RANK() OVER(PARTITION BY department_id ORDER BY salary
DESC) AS dense_rank,
FROM hr.employee) tbl;
```

_

Homework for the learners -

#Q: From each department, display the details of the employee having the 3rd highest salary. Display multiple people if they have the same salaries.

Soln:

```
SELECT *
FROM (
```

```
   SELECT department_id, employee_id, name, salary,
        DENSE_RANK() OVER (PARTITION BY department_id ORDER BY
salary DESC) as sal_rnk
   FROM hr.employee
) tbl
WHERE sal_rnk = 3;
```

_

Quiz - 3

```
Q. Joe wants to rank players by their scores. If there is a tie, he
wants the next rank to jump by the number of ties.
Which window function should he use?

   a) DENSE_RANK
   b) RANK - correct
   c) ROW_NUMBER
```

_

NTILE

Imagine you have a group of people, and you want to divide them into smaller groups, but you want each group to have about the same number of people.

NTILE() helps you do that by splitting them into a specified number of **approximately equal-sized groups or buckets**.

For each row in a group, the NTILE() window function assigns a bucket number representing the group to which the row belongs.

Syntax:
NTILE(num_buckets) OVER (PARTITION BY partition_column ORDER BY order_column)

What are Quartile, Quintile, and Decile?

- Quartiles divide a data set into **4** equal parts.
- Quintiles divides a data set into **5** equal parts.
- Deciles divide a data set into **10** equal parts.

#Q: Divide all the employees into 4 different buckets (i.e. quartiles) based on their salaries.

- **The 1st quartile should contain the employees with the lowest salaries.**
- **The 4th quartile should contain the employees with the highest salaries.**

Query:

```
SELECT
 employee_id,
 department_id,
 salary,
 NTILE(4) OVER(ORDER BY salary) AS sal_group
FROM hr.employee
ORDER BY sal_group;
```

Important points about NTILE()

- If the number of rows in the results set can be divided evenly, the results will be broken up into **n equal-sized groups, labeled 1 to n**.
- If they can't be divided up evenly, some groups will end up with one more row than others.
- The **ORDER BY** salary clause within the NTILE() function ensures that the rows are **ordered by salary before they are divided into groups**.
- Note that the **NTILE() is only using the count of rows to split the groups**, and is not using a field value to determine where to make the splits.
- Therefore, it's possible that **two rows with the same value** specified in the ORDER BY clause will **end up in two different groups.**

_

Quiz - 4

Q. Is it true or false that the NTILE function creates groups solely based on the number of rows and the specified NTILE value?

a) True - correct
b) False

_