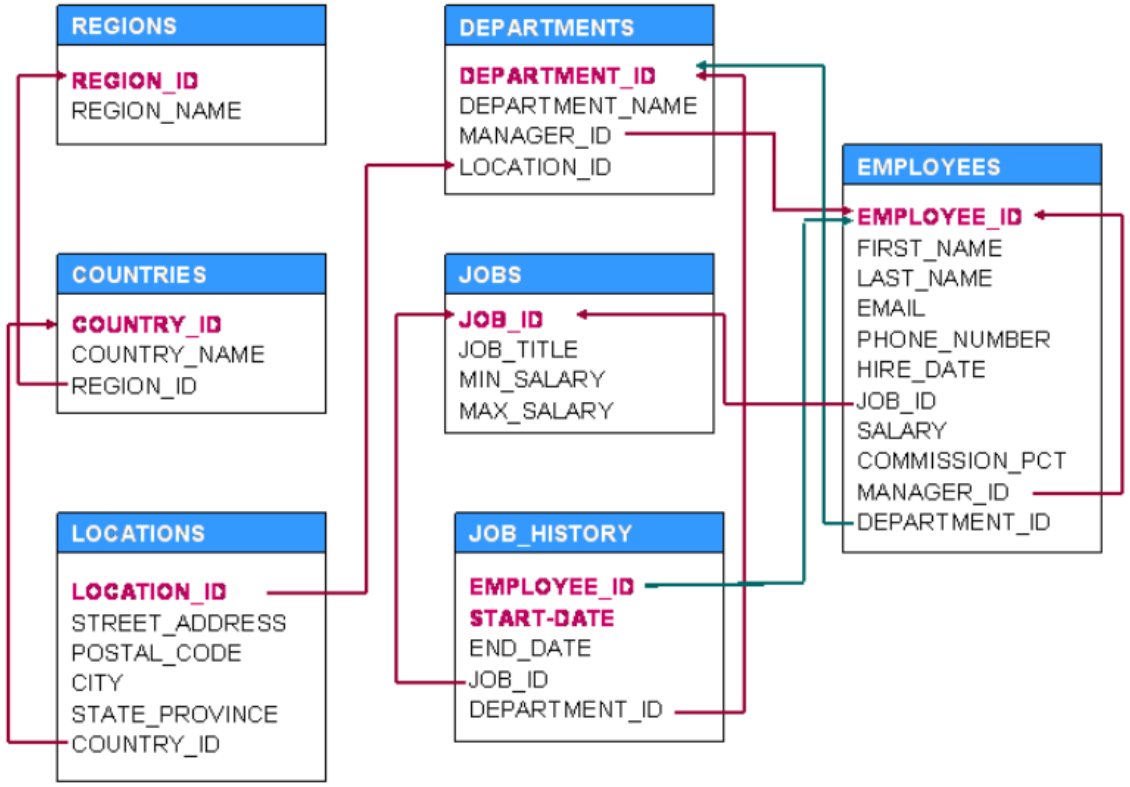
Advanced Analytics with SQL

CS 341 Database Systems

Recall: Subqueries

Subquery Example

Retrieve employees who are located in location # 1700



Comparison

Comparison operators
(=,>, >=, <, <=, <>)

Find the employee(s) who have the highest salary

```
SELECT
    employee_id, first_name, last_name,
salary
FROM
    employees
WHERE
    salary = (SELECT
            MAX(salary)
        FROM
            employees)
ORDER BY first_name , last_name;
```

Correlated Subqueries

- A subquery that uses values from the outer query.
- The inner query depends on the row that is currently being examined in the outer query.

```
SELECT
    employee_id, first_name, last_name, salary,
department_id
FROM
    employees e
WHERE
    salary > (SELECT
            AVG(salary)
        FROM
            employees
        WHERE
            department_id = e.department_id)
ORDER BY department_id, first_name, last_name;
```

ANY - ALL

Find all employees whose salaries are greater than the lowest salary of **every** department

X > ALL (subquery) – evaluates to true of x is greater than every value returned by the subquery.

X > ANY (subquery) – evaluates to true of x is greater than any value returned by the subquery.

```
SELECT
    employee_id, first_name, last_name,
salary
FROM
    employees
WHERE
    salary >= ALL (SELECT
            MIN(salary)
        FROM
            employees
        GROUP BY department_id)
ORDER BY first_name , last_name;
```

Refactoring

- **Refactoring** is the process of restructuring existing code or queries without changing their external behavior or output. The goal of refactoring is to improve readability, maintainability (updates/debugging code), performance, or structure.

Refactoring in SQL involves:

- Replacing nested subqueries with CTEs or derived tables for better clarity.
- Using window functions instead of complex aggregations.
- Simplifying joins and filtering logic.

WITH clause

```
WITH cte_name (column1, column2, ...) AS (
    -- Subquery or SQL statement here
)
SELECT *
FROM cte_name;
```

```
WITH sid_103 AS (
    SELECT sid FROM Reserves WHERE bid=103
)
SELECT * FROM sid_103;
```

```
WITH sid_103 AS (
    SELECT sid FROM Reserves WHERE bid=103
)
SELECT DISTINCT sname FROM Sailors S WHERE
sid IN (SELECT * FROM sid_103);
```

** You can use the CTE as another table available to this specific query*

- Also known as CTE – Common Table Expression used to define a temporary table or result set that can be referenced in another query in SELECT, INSERT, UPDATE or DELETE.
- Works like Assignment operator in relational algebra allowing to break complex SQL queries into smaller and manageable parts.

Lab11 – Question 23

Find department names whose total salary is greater than the average organization salary. Note that the average organization salary is the average of all department totals.

```
WITH DepartmentTotalSalaries AS (
  SELECT department_name, SUM(salary) AS total_salary
  FROM employees e join departments d ON e.department_id=d.department_id
  GROUP BY department_name
),
AverageOrganizationSalary AS (
  SELECT AVG(total_salary) AS avg_org_salary
  FROM DepartmentTotalSalaries
)

SELECT department_name
FROM DepartmentTotalSalaries
WHERE total_salary > (SELECT avg_org_salary FROM AverageOrganizationSalary);
```

Window Functions

How is this different?

Joins

Bring together tables/information for analysis

Subqueries

Nested queries (query within a query) for a deep dive into data

Window Functions

Allow calculations across a specified range of rows.

Window Functions

- *A powerful tool that allows you to perform calculations across a set of rows that are related to the current row within the same query result.*
- **Group by:** Collapses data into a single output for each group
- **Window functions:** Maintain the individual rows in addition to new insights.

Use Cases

- **Ranking:** Assigning ranks to rows based on specific criteria.
- **Running Totals:** Calculating cumulative sums or averages.
- **Moving Averages:** Smoothing out fluctuations in data over a specified range.
- **Lag and Lead Analysis:** Comparing values from different rows within the same dataset.

View all Employees

```
1  SELECT
2      EMPLOYEE_ID, DEPARTMENT_ID, SALARY
3  FROM EMPLOYEES ORDER BY DEPARTMENT_ID;
```

Query Result

SQL | All Rows Fetched: 108 in 0.009 seconds

	EMPLOYEE_ID	DEPARTMENT_ID	SALARY
1	200	10	4400
2	201	20	13000
3	202	20	6000
4	114	30	11000
5	119	30	2500
6	115	30	3100
7	116	30	2900
8	117	30	2800
9	118	30	2600
10	203	40	6500

Group by with Aggregate Function

```
7  SELECT
8      DEPARTMENT_ID, ROUND(AVG(SALARY),2)
9  FROM EMPLOYEES
10 GROUP BY DEPARTMENT_ID
11 ORDER BY DEPARTMENT_ID;
```

Query Result

SQL | All Rows Fetched: 12 in 0.003 seconds

	DEPARTMENT_ID	ROUND(AVG(SALARY),2)
1	10	4400
2	20	9500
3	30	4150
4	40	6500
5	50	3475.56
6	60	5760
7	70	10000
8	80	8955.88
9	90	29500
10	100	8601.33
11	110	10154
12	(null)	7000

How does the group by work?

	EMPLOYEE_ID	DEPARTMENT_ID	SALARY
1	200	10	4400

Department 10

Department 20

2	201	20	13000
3	202	20	6000

4	114	30	11000
5	119	30	2500
6	115	30	3100
7	116	30	2900
8	117	30	2800
9	118	30	2600

Department 30

Over a Partition

```
14  SELECT
15      EMPLOYEE_ID, DEPARTMENT_ID, SALARY,
16      AVG(SALARY) OVER (PARTITION BY DEPARTMENT_ID) AS DEPT_AVG
17  FROM EMPLOYEES;
18
```

Query Result

SQL | Fetched 50 rows in 0.003 seconds

	EMPLOYEE_ID	DEPARTMENT_ID	SALARY	DEPT_AVG
1	200	10	4400	4400
2	201	20	13000	9500
3	202	20	6000	9500
4	114	30	11000	4150
5	119	30	2500	4150
6	115	30	3100	4150
7	116	30	2900	4150
8	117	30	2800	4150
9	118	30	2600	4150
10	203	40	6500	6500
11	198	50	2600	3475.555555555555555555555555555555555555556
12	199	50	2600	3475.555555555555555555555555555555555555556
13	120	50	8000	3475.555555555555555555555555555555555555556

ROUND

```
20 SELECT
21     EMPLOYEE_ID, DEPARTMENT_ID, SALARY,
22     ROUND(AVG(SALARY) OVER (PARTITION BY DEPARTMENT_ID),2) AS DEPT_AVG
23 FROM EMPLOYEES;
24
```

Query Result

SQL | Fetched 50 rows in 0.004 seconds

	EMPLOYEE_ID	DEPARTMENT_ID	SALARY	DEPT_AVG
1	200	10	4400	4400
2	201	20	13000	9500
3	202	20	6000	9500
4	114	30	11000	4150
5	119	30	2500	4150
6	115	30	3100	4150
7	116	30	2900	4150
8	117	30	2800	4150
9	118	30	2600	4150
10	203	40	6500	6500
11	198	50	2600	3475.56
12	199	50	2600	3475.56
13	120	50	8000	3475.56

Department Name

```
26 SELECT
27     EMPLOYEE_ID, DEPARTMENT_NAME, SALARY,
28     ROUND(AVG(SALARY) OVER (PARTITION BY DEPARTMENT_ID),2) AS DEPT_AVG
29 FROM EMPLOYEES JOIN DEPARTMENTS USING (DEPARTMENT_ID);
30
```

Query Result

SQL | Fetched 50 rows in 0.006 seconds

	EMPLOYEE_ID	DEPARTMENT_NAME	SALARY	DEPT_AVG
1	200	Administration	4400	4400
2	201	Marketing	13000	9500
3	202	Marketing	6000	9500
4	114	Purchasing	11000	4150
5	119	Purchasing	2500	4150
6	115	Purchasing	3100	4150
7	116	Purchasing	2900	4150
8	117	Purchasing	2800	4150
9	118	Purchasing	2600	4150
10	203	Human Resources	6500	6500
11	198	Shipping	2600	3475.56
12	199	Shipping	2600	3475.56

Visualize the Windows

```
6  SELECT
7      EMPLOYEE_ID, DEPARTMENT_NAME, SALARY
8  FROM EMPLOYEES JOIN DEPARTMENTS USING (DEPARTMENT_ID)
9  ORDER BY DEPARTMENT_NAME;
```

	EMPLOYEE_ID	DEPARTMENT_NAME	SALARY
1	205	Accounting	12008
2	206	Accounting	8300

4	101	Executive	17000
5	102	Executive	17000
6	800	Executive	60000
7	100	Executive	24000

8	108	Finance	12008
9	110	Finance	8200
10	109	Finance	9000
11	112	Finance	7800
12	113	Finance	6900
13	111	Finance	7700

Syntax: OVER ( )

- **OVER() clause** creates a window.
- When there in nothing written within the bracket, it creates a large window with all the rows of the table

```
38 SELECT
39     EMPLOYEE_ID, DEPARTMENT_ID, SALARY,
40     ROUND(AVG(SALARY) OVER (),2) AS ORG_AVG
41 FROM EMPLOYEES;
42
```

Query Result

SQL | Fetched 50 rows in 0.003 seconds

	EMPLOYEE_ID	DEPARTMENT_ID	SALARY	ORG_AVG
1	198	50	2600	6957.56
2	199	50	2600	6957.56
3	200	10	4400	6957.56
4	201	20	13000	6957.56
5	202	20	6000	6957.56
6	203	40	6500	6957.56
7	204	70	10000	6957.56
8	205	110	12008	6957.56

Syntax: Partition By

- **OVER(Partition By _____) :** The groups of rows are formed on the basis of the partition specified.

```
44  SELECT
45      EMPLOYEE_ID, DEPARTMENT_ID, SALARY,
46      ROUND(AVG(SALARY) OVER (PARTITION BY DEPARTMENT_ID),2) AS DEPT_AVG
47  FROM EMPLOYEES;
48
49
```

Query Result

SQL | Fetched 50 rows in 0.002 seconds

	EMPLOYEE_ID	DEPARTMENT_ID	SALARY	DEPT_AVG
1	200	10	4400	4400
2	201	20	13000	9500
3	202	20	6000	9500
4	114	30	11000	4150
5	119	30	2500	4150
6	115	30	3100	4150
7	116	30	2900	4150

More Aggregates alongside original rows

```
50 SELECT
51     EMPLOYEE_ID, DEPARTMENT_NAME, SALARY,
52     ROUND(AVG(SALARY) OVER (PARTITION BY DEPARTMENT_NAME),2) AS DEPT_AVG,
53     MIN(SALARY) OVER (PARTITION BY DEPARTMENT_NAME) AS DEPT_MIN,
54     MAX(SALARY) OVER (PARTITION BY DEPARTMENT_NAME) AS DEPT_MAX
55 FROM EMPLOYEES JOIN DEPARTMENTS USING (DEPARTMENT_ID);
56
```

Query Result

SQL | Fetched 50 rows in 0.005 seconds

	EMPLOYEE_ID	DEPARTMENT_NAME	SALARY	DEPT_AVG	DEPT_MIN	DEPT_MAX
1	205	Accounting	12008	10154	8300	12008
2	206	Accounting	8300	10154	8300	12008
3	200	Administration	4400	4400	4400	4400
4	101	Executive	17000	29500	17000	60000
5	102	Executive	17000	29500	17000	60000
6	800	Executive	60000	29500	17000	60000
7	100	Executive	24000	29500	17000	60000

Department-wise total salary along with each employee information

```
58  SELECT
59      EMPLOYEE_ID, FIRST_NAME, LAST_NAME, DEPARTMENT_ID, SALARY,
60      SUM(SALARY) OVER (PARTITION BY DEPARTMENT_ID)
61      AS CUMULATIVESALARY
62  FROM EMPLOYEES
63
```

Script Output | Query Result

SQL | Fetched 50 rows in 0.004 seconds

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	DEPARTMENT_ID	SALARY	CUMULATIVESALARY
1	200	Jennifer	Whalen	10	4400	4400
2	201	Michael	Hartstein	20	13000	19000
3	202	Pat	Fay	20	6000	19000
4	114	Den	Raphaely	30	11000	24900
5	119	Karen	Colmenares	30	2500	24900
6	115	Alexander	Khoo	30	3100	24900
7	116	Shelli	Baida	30	2900	24900
8	117	Sigal	Tobias	30	2800	24900
9	118	Guy	Himuro	30	2600	24900
10	203	Susan	Mavris	40	6500	6500

Department-wise cumulative salary along with each employee information (running total)

```
58  SELECT
59      EMPLOYEE_ID, FIRST_NAME, LAST_NAME, DEPARTMENT_ID, SALARY,
60      SUM(SALARY) OVER (PARTITION BY DEPARTMENT_ID ORDER BY SALARY DESC)
61      AS CumulativeSalary
62  FROM EMPLOYEES
63
```

Query Result

SQL | Fetched 50 rows in 0.003 seconds

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	DEPARTMENT_ID	SALARY	CUMULATIVESALARY
1	200	Jennifer	Whalen	10	4400	4400
2	201	Michael	Hartstein	20	13000	13000
3	202	Pat	Fay	20	6000	19000
4	114	Den	Raphaely	30	11000	11000
5	115	Alexander	Khoo	30	3100	14100
6	116	Shelli	Baida	30	2900	17000
7	117	Sigal	Tobias	30	2800	19800

Filter where Cumulative Sum >3000

Use of subqueries to filter – where clause within initial query will not work

```
67 WITH CumulativeSalaries AS
68 ( SELECT
69     EMPLOYEE_ID, FIRST_NAME, LAST_NAME, DEPARTMENT_ID, SALARY,
70     SUM(SALARY) OVER (PARTITION BY DEPARTMENT_ID ORDER BY SALARY DESC) AS CumulativeSalar
71 FROM EMPLOYEES)
72 SELECT * FROM CumulativeSalaries WHERE CumulativeSalary > 30000
73 ORDER BY CumulativeSalary;
```

Query Result | Script Output | Query Result 1 | Query Result 2

SQL | Fetched 50 rows in 0.004 seconds

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	DEPARTMENT_ID	SALARY	CUMULATIVESALARY
1	123	Shanta	Vollman	50	6500	30600
2	124	Kevin	Mourgos	50	5800	36400
3	112	Jose Manuel	Urman	100	7800	37008
4	147	Alberto	Errazuriz	80	12000	39500
5	184	Nandita	Sarchand	50	4200	40600
6	185	Alexis	Bull	50	4100	44700
7	111	Ismael	Sciarra	100	7700	44708

Window Functions with OVER clause

- **RANK():** used to rank items within a partition
- **OVER (ORDER BY salary DESC) –** orders rows within each window

```
76 SELECT
77     EMPLOYEE_ID, DEPARTMENT_NAME, SALARY,
78     RANK() OVER(ORDER BY SALARY DESC) AS OVERALL_RANK
79 FROM EMPLOYEES JOIN DEPARTMENTS USING (DEPARTMENT_ID);
```

Query Result | Script Output | Query Result 1 | Query Result 2

SQL | Fetched 50 rows in 0.004 seconds

	EMPLOYEE_ID	DEPARTMENT_NAME	SALARY	OVERALL_RANK
1	800	Executive	60000	1
2	100	Executive	24000	2
3	102	Executive	17000	3
4	101	Executive	17000	3
5	145	Sales	14000	5
6	146	Sales	13500	6
7	201	Marketing	13000	7
8	108	Finance	12008	8
9	205	Accounting	12008	8
10	147	Sales	12000	10
11	168	Sales	11500	11
12	114	Purchasing	11000	12
13	148	Sales	11000	12
14	174	Sales	11000	12

RANK () with partition

```
82 SELECT
83     EMPLOYEE_ID, DEPARTMENT_NAME, SALARY,
84     RANK() OVER(ORDER BY SALARY DESC) AS OVERALL_RANK,
85     RANK() OVER(PARTITION BY DEPARTMENT_NAME ORDER BY SALARY DESC) AS DEPT_RANK
86 FROM EMPLOYEES JOIN DEPARTMENTS USING (DEPARTMENT_ID);
```

Query Result | Script Output | Query Result 1 | Query Result 2

SQL | Fetched 50 rows in 0.003 seconds

	EMPLOYEE_ID	DEPARTMENT_NAME	SALARY	OVERALL_RANK	DEPT_RANK
1	800	Executive	60000	1	1
2	100	Executive	24000	2	2
3	101	Executive	17000	3	3
4	102	Executive	17000	3	3
5	145	Sales	14000	5	1
6	146	Sales	13500	6	2
7	201	Marketing	13000	7	1
8	108	Finance	12008	8	1
9	205	Accounting	12008	8	1
10	147	Sales	12000	10	3
11	168	Sales	11500	11	4
12	148	Sales	11000	12	5
13	174	Sales	11000	12	5
14	114	Purchasing	11000	12	1

DENSE_RANK()

Does not skip a rank for duplicate values in the data.

```
90 SELECT EMPLOYEE_ID, DEPARTMENT_NAME, SALARY,
91     RANK() OVER(ORDER BY SALARY DESC) AS OVERALL_RANK,
92     DENSE_RANK() OVER(ORDER BY SALARY DESC) AS OVERALL_RANK
93     --RANK() OVER(PARTITION BY DEPARTMENT_NAME ORDER BY SALARY DESC) AS DEPT_RANK
94 FROM EMPLOYEES JOIN DEPARTMENTS USING (DEPARTMENT_ID);
```

Query Result

SQL | Fetched 50 rows in 0.004 seconds

	EMPLOYEE_ID	DEPARTMENT_NAME	SALARY	OVERALL_RANK	OVERALL_RANK_1
1	800	Executive	60000	1	1
2	100	Executive	24000	2	2
3	102	Executive	17000	3	3
4	101	Executive	17000	3	3
5	145	Sales	14000	5	4
6	146	Sales	13500	6	5
7	201	Marketing	13000	7	6
8	108	Finance	12008	8	7
9	205	Accounting	12008	8	7
10	147	Sales	12000	10	8
11	168	Sales	11500	11	9
12	114	Purchasing	11000	12	10
13	148	Sales	11000	12	10
14	174	Sales	11000	12	10
15	162	Sales	10500	15	11

Find the highest salaried person in each department using the RANK window function

```
97  WITH RANKEDEMPLOYEES AS
98      (SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME, DEPARTMENT_ID, SALARY,
99      RANK() OVER (PARTITION BY DEPARTMENT_ID ORDER BY SALARY DESC) AS SALARYRANK
100     FROM EMPLOYEES)
101 SELECT * FROM RANKEDEMPLOYEES WHERE SALARYRANK = 1;
```

Query Result

SQL | Fetched 50 rows in 0.003 seconds

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	DEPARTMENT_ID	SALARY	SALARYRANK
1	200	Jennifer	Whalen	10	4400	1
2	201	Michael	Hartstein	20	13000	1
3	202	Pat	Fay	20	6000	2
4	114	Den	Raphaely	30	11000	1
5	115	Alexander	Khoo	30	3100	2
6	116	Shelli	Baida	30	2900	3
7	117	Sigal	Tobias	30	2800	4
8	118	Guy	Himuro	30	2600	5
9	119	Karen	Colmenares	30	2500	6

Find the highest salaried person in each department using the RANK window function

```
97  WITH RANKEDEMPLOYEES AS
98      (SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME, DEPARTMENT_ID, SALARY,
99      RANK() OVER (PARTITION BY DEPARTMENT_ID ORDER BY SALARY DESC) AS SALARYRANK
100     FROM EMPLOYEES)
101 SELECT * FROM RANKEDEMPLOYEES WHERE SALARYRANK = 1;
```

Query Result

SQL | All Rows Fetched: 12 in 0.004 seconds

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	DEPARTMENT_ID	SALARY	SALARYRANK
1	200	Jennifer	Whalen	10	4400	1
2	201	Michael	Hartstein	20	13000	1
3	114	Den	Raphaely	30	11000	1
4	203	Susan	Mavris	40	6500	1
5	121	Adam	Fripp	50	8200	1
6	103	Alexander	Hunold	60	9000	1
7	204	Hermann	Baer	70	10000	1
8	145	John	Russell	80	14000	1
9	800	Muhammad	Shehzad	90	60000	1
10	108	Nancy	Greenberg	100	12008	1
11	205	Shelley	Higgins	110	12008	1
12	178	Kimberely	Grant	(null)	7000	1

ROW_NUMBER()

```
104 SELECT
105     EMPLOYEE_ID, FIRST_NAME, LAST_NAME, DEPARTMENT_ID, SALARY,
106     ROW_NUMBER() OVER (PARTITION BY DEPARTMENT_ID ORDER BY SALARY DESC) AS MyRowNum
107 FROM EMPLOYEES;
108
109
```

Query Result

SQL | Fetched 50 rows in 0.003 seconds

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	DEPARTMENT_ID	SALARY	MYROWNUM
1	200	Jennifer	Whalen	10	4400	1
2	201	Michael	Hartstein	20	13000	1
3	202	Pat	Fay	20	6000	2
4	114	Den	Raphaely	30	11000	1
5	115	Alexander	Khoo	30	3100	2
6	116	Shelli	Baida	30	2900	3
7	117	Sigal	Tobias	30	2800	4

Working Similar to Ranking

```
110 WITH RANKEDEMPLOYEES AS
111 (SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME, DEPARTMENT_ID, SALARY,
112 ROW_NUMBER() OVER (PARTITION BY DEPARTMENT_ID ORDER BY SALARY DESC) AS MYROWNUM
113 FROM EMPLOYEES)
114 SELECT *FROM RANKEDEMPLOYEES WHERE MYROWNUM <= 3;
115
```

Query Result

SQL | All Rows Fetched: 26 in 0.005 seconds

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	DEPARTMENT_ID	SALARY	MYROWNUM
1	200	Jennifer	Whalen	10	4400	1
2	201	Michael	Hartstein	20	13000	1
3	202	Pat	Fay	20	6000	2
4	114	Den	Raphaely	30	11000	1
5	115	Alexander	Khoo	30	3100	2
6	116	Shelli	Baida	30	2900	3
7	203	Susan	Mavris	40	6500	1
8	121	Adam	Fripp	50	8200	1
9	120	Matthew	Weiss	50	8000	2
10	122	Payam	Kaufling	50	7900	3

LAG ()

- Write a query to compare salary of each employee with the previous employee in the same department.
- By default, lag checks previous row.
- Lag (salary, 2 ,0) → attribute, 2 record previous to the selected, default value

```
117 SELECT
118     EMPLOYEE_ID, DEPARTMENT_ID, SALARY,
119     LAG(SALARY) OVER (PARTITION BY DEPARTMENT_ID ORDER BY EMPLOYEE_ID)
120     AS PREV_EMP_SALARY
121 FROM EMPLOYEES E;
122
```

Query Result

SQL | Fetched 50 rows in 0.002 seconds

	EMPLOYEE_ID	DEPARTMENT_ID	SALARY	PREV_EMP_SALARY
1	200	10	4400	(null)
2	201	20	13000	(null)
3	202	20	6000	13000
4	114	30	11000	(null)
5	115	30	3100	11000
6	116	30	2900	3100
7	117	30	2800	2900
8	118	30	2600	2800
9	119	30	2500	2600

LEAD ()

Similar to lag, now for rows following or next to the specific record

```
124 SELECT
125     EMPLOYEE_ID, DEPARTMENT_ID, SALARY,
126     LAG(SALARY) OVER (PARTITION BY DEPARTMENT_ID ORDER BY EMPLOYEE_ID)
127     AS PREV_EMP_SALARY,
128     LEAD(SALARY) OVER (PARTITION BY DEPARTMENT_ID ORDER BY EMPLOYEE_ID)
129     AS NEXT_EMP_SALARY
130 FROM EMPLOYEES E;
```

Query Result

SQL | Fetched 50 rows in 0.005 seconds

	EMPLOYEE_ID	DEPARTMENT_ID	SALARY	PREV_EMP_SALARY	NEXT_EMP_SALARY
1	200	10	4400	(null)	(null)
2	201	20	13000	(null)	6000
3	202	20	6000	13000	(null)
4	114	30	11000	(null)	3100
5	115	30	3100	11000	2900
6	116	30	2900	3100	2800
7	117	30	2800	2900	2600
8	118	30	2600	2800	2500
9	119	30	2500	2600	(null)
10	203	40	6500	(null)	(null)
11	120	50	8000	(null)	8200

Practice Questions

Test it out on SQL Developer

Write SQL queries using Window Functions

1. Write a query to display if the salary of an employee is higher, lower or equal to the previous employee in the same department.
2. Get the Top 3 employees with highest salaries in each Job title.
3. Assign row number to the employees table based on the increasing value of commission. Ignore the employees having a null valued commission. Output employee_id, department_id, commission and row number.