MySQL 8 Window Functions - Basic Student Notes

Database Setup

```sql
```

```sql
-- Create Database
CREATE DATABASE company_db;
USE company_db;

-- Create Departments Table
CREATE TABLE departments (
    dept_id INT PRIMARY KEY AUTO_INCREMENT,
    dept_name VARCHAR(50) NOT NULL,
    location VARCHAR(50) NOT NULL
);

-- Create Employees Table
CREATE TABLE employees (
    emp_id INT PRIMARY KEY AUTO_INCREMENT,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    dept_id INT,
    salary DECIMAL(8,2) NOT NULL,
    hire_date DATE NOT NULL,
    FOREIGN KEY (dept_id) REFERENCES departments(dept_id)
);

-- Insert Sample Data
INSERT INTO departments (dept_name, location) VALUES
('HR', 'New York'),
('IT', 'San Francisco'),
('Finance', 'Chicago'),
('Marketing', 'Los Angeles'),
('Sales', 'Miami');

INSERT INTO employees (first_name, last_name, dept_id, salary, hire_date) VALUES
('John', 'Smith', 2, 75000, '2020-01-15'),
('Sarah', 'Johnson', 2, 85000, '2019-03-20'),
('Mike', 'Brown', 1, 55000, '2021-06-10'),
('Lisa', 'Davis', 3, 70000, '2020-08-15'),
('Tom', 'Wilson', 2, 95000, '2018-05-12'),
('Emma', 'Garcia', 1, 50000, '2022-01-20'),
('David', 'Martinez', 3, 80000, '2019-11-08'),
('Anna', 'Lee', 4, 60000, '2021-04-25'),
('James', 'Taylor', 5, 65000, '2020-12-03'),
('Maria', 'Lopez', 4, 58000, '2021-09-17'),
('Robert', 'Anderson', 5, 72000, '2019-07-30'),
('Jennifer', 'Thomas', 1, 48000, '2022-03-15');
```

Window Function Basics

Syntax:

```
sql

function_name([arguments]) OVER (
    [PARTITION BY column_name]
    [ORDER BY column_name [ASC|DESC]]
)
```

Key Points:

- Window functions don't reduce the number of rows (unlike GROUP BY)
- PARTITION BY divides data into groups
- ORDER BY specifies the order for calculations
- OVER clause is mandatory

---

1. ROW_NUMBER()

Assigns unique sequential numbers to rows.

```
sql

-- Basic row numbering
SELECT
    emp_id,
    first_name,
    last_name,
    salary,
    ROW_NUMBER() OVER (ORDER BY salary DESC) as row_num
FROM employees;

-- Row numbering within each department
SELECT
    e.first_name,
    e.last_name,
    d.dept_name,
    e.salary,
    ROW_NUMBER() OVER (PARTITION BY d.dept_name ORDER BY e.salary DESC) as dept_row_num
FROM employees e
JOIN departments d ON e.dept_id = d.dept_id;
```

---

2. RANK() and DENSE_RANK()

RANK() - Same values get same rank, skips next ranks **DENSE_RANK()** - Same values get same rank, no gaps

sql

```
-- Salary ranking with RANK and DENSE_RANK
SELECT
    first_name,
    last_name,
    salary,
    RANK() OVER (ORDER BY salary DESC) as salary_rank,
    DENSE_RANK() OVER (ORDER BY salary DESC) as salary_dense_rank
FROM employees
ORDER BY salary DESC;

-- Department-wise ranking
SELECT
    e.first_name,
    e.last_name,
    d.dept_name,
    e.salary,
    RANK() OVER (PARTITION BY e.dept_id ORDER BY e.salary DESC) as dept_rank
FROM employees e
JOIN departments d ON e.dept_id = d.dept_id
ORDER BY d.dept_name, dept_rank;
```

---

3. NTILE(n)

Divides rows into 'n' approximately equal groups.

sql

```sql
-- Divide employees into 4 salary groups
SELECT
    first_name,
    last_name,
    salary,
    NTILE(4) OVER (ORDER BY salary DESC) as salary_quartile,
    CASE
        WHEN NTILE(4) OVER (ORDER BY salary DESC) = 1 THEN 'High Earners'
        WHEN NTILE(4) OVER (ORDER BY salary DESC) = 2 THEN 'Above Average'
        WHEN NTILE(4) OVER (ORDER BY salary DESC) = 3 THEN 'Below Average'
        ELSE 'Low Earners'
    END as salary_group
FROM employees
ORDER BY salary DESC;
```

---

4. LAG() and LEAD()

Access data from previous (LAG) or next (LEAD) rows.

sql

```sql
-- Compare salary with previous employee (ordered by salary)
SELECT
    first_name,
    last_name,
    salary,
    LAG(salary) OVER (ORDER BY salary DESC) as prev_salary,
    salary - LAG(salary) OVER (ORDER BY salary DESC) as salary_diff,
    LEAD(salary) OVER (ORDER BY salary DESC) as next_salary
FROM employees
ORDER BY salary DESC;

-- Department-wise salary comparison
SELECT
    e.first_name,
    e.last_name,
    d.dept_name,
    e.salary,
    LAG(e.salary) OVER (PARTITION BY e.dept_id ORDER BY e.salary DESC) as prev_dept_salary
FROM employees e
JOIN departments d ON e.dept_id = d.dept_id
ORDER BY d.dept_name, e.salary DESC;
```

---

5. FIRST_VALUE() and LAST_VALUE()

Get first or last value in the window.

```sql
sql

-- Show highest and lowest salary in each department
SELECT
    e.first_name,
    e.last_name,
    d.dept_name,
    e.salary,
    FIRST_VALUE(e.salary) OVER (
        PARTITION BY e.dept_id
        ORDER BY e.salary DESC
        ROWS UNBOUNDED PRECEDING
    ) as highest_dept_salary,
    FIRST_VALUE(e.salary) OVER (
        PARTITION BY e.dept_id
        ORDER BY e.salary ASC
        ROWS UNBOUNDED PRECEDING
    ) as lowest_dept_salary
FROM employees e
JOIN departments d ON e.dept_id = d.dept_id
ORDER BY d.dept_name, e.salary DESC;
```

---

6. Aggregate Functions as Window Functions

```sql
sql
```

```
-- Running totals and counts
SELECT
    e.first_name,
    e.last_name,
    d.dept_name,
    e.salary,
    -- Running sum of salaries (ordered by hire_date)
    SUM(e.salary) OVER (ORDER BY e.hire_date ROWS UNBOUNDED PRECEDING) as running_total_salary,
    -- Count of employees hired so far
    COUNT(*) OVER (ORDER BY e.hire_date ROWS UNBOUNDED PRECEDING) as employees_count,
    -- Department average salary
    AVG(e.salary) OVER (PARTITION BY e.dept_id) as dept_avg_salary,
    -- Difference from department average
    e.salary - AVG(e.salary) OVER (PARTITION BY e.dept_id) as diff_from_avg
FROM employees e
JOIN departments d ON e.dept_id = d.dept_id
ORDER BY e.hire_date;
```

7. PERCENT_RANK() and CUME_DIST()

sql

```
-- Salary percentiles and cumulative distribution
SELECT
    first_name,
    last_name,
    salary,
    ROUND(PERCENT_RANK() OVER (ORDER BY salary) * 100, 1) as salary_percentile,
    ROUND(CUME_DIST() OVER (ORDER BY salary) * 100, 1) as cumulative_percent
FROM employees
ORDER BY salary DESC;
```

Practice Examples

Example 1: Department Analysis

sql

```sql
-- Complete department analysis
SELECT
    d.dept_name,
    e.first_name,
    e.last_name,
    e.salary,
    -- Rank within department
    RANK() OVER (PARTITION BY d.dept_name ORDER BY e.salary DESC) as dept_rank,
    -- Department employee count
    COUNT(*) OVER (PARTITION BY d.dept_name) as dept_emp_count,
    -- Department total salary
    SUM(e.salary) OVER (PARTITION BY d.dept_name) as dept_total_salary,
    -- Department average salary
    ROUND(AVG(e.salary) OVER (PARTITION BY d.dept_name), 2) as dept_avg_salary
FROM employees e
JOIN departments d ON e.dept_id = d.dept_id
ORDER BY d.dept_name, dept_rank;
```

Example 2: Salary Analysis

sql

```sql
-- Comprehensive salary analysis
SELECT
    first_name,
    last_name,
    salary,
    -- Overall ranking
    RANK() OVER (ORDER BY salary DESC) as overall_rank,
    -- Salary quartile
    NTILE(4) OVER (ORDER BY salary DESC) as salary_quartile,
    -- Percentile rank
    ROUND(PERCENT_RANK() OVER (ORDER BY salary) * 100, 1) as percentile,
    -- Difference from highest salary
    FIRST_VALUE(salary) OVER (ORDER BY salary DESC ROWS UNBOUNDED PRECEDING) - salary as diff_from_highest,
    -- Difference from company average
    ROUND(salary - AVG(salary) OVER (), 2) as diff_from_company_avg
FROM employees
ORDER BY salary DESC;
```

Example 3: Hiring Timeline Analysis

sql

```
-- Hiring pattern analysis
SELECT
    first_name,
    last_name,
    hire_date,
    salary,
    -- Hiring sequence number
    ROW_NUMBER() OVER (ORDER BY hire_date) as hire_sequence,
    -- Days since previous hire
    DATEDIFF(hire_date, LAG(hire_date) OVER (ORDER BY hire_date)) as days_since_prev_hire,
    -- Running count of employees
    COUNT(*) OVER (ORDER BY hire_date ROWS UNBOUNDED PRECEDING) as total_employees,
    -- Running average salary
    ROUND(AVG(salary) OVER (ORDER BY hire_date ROWS UNBOUNDED PRECEDING), 2) as running_avg_salary
FROM employees
ORDER BY hire_date;
```

Quick Reference

Function	Purpose	Example
ROW_NUMBER()	Sequential numbering	1, 2, 3, 4...
RANK()	Ranking with gaps	1, 2, 2, 4...
DENSE_RANK()	Ranking without gaps	1, 2, 2, 3...
NTILE(n)	Divide into n groups	1, 1, 2, 2, 3, 3...
LAG()	Previous row value	Access preceding row
LEAD()	Next row value	Access following row
FIRST_VALUE()	First value in window	Get first value
LAST_VALUE()	Last value in window	Get last value
SUM() OVER	Running/Partitioned sum	Cumulative totals
AVG() OVER	Running/Partitioned average	Moving averages
COUNT() OVER	Running/Partitioned count	Cumulative counts

Key Points to Remember

1. **OVER clause is mandatory** for window functions
2. **PARTITION BY** is like GROUP BY but doesn't collapse rows
3. **ORDER BY** in OVER clause determines calculation order
4. **Window functions are calculated after WHERE** but before ORDER BY

5. **Use ROWS UNBOUNDED PRECEDING** for running totals
6. **Multiple window functions** can be used in same query

Common Frame Clauses

```sql
-- All preceding rows
ROWS UNBOUNDED PRECEDING

-- Current + 2 preceding rows
ROWS 2 PRECEDING

-- Between 1 preceding and 1 following
ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING

-- All rows in partition
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
```