12 MySQL Window Functions

Sales Table:

```sql
create database mySQLWF;
use mySQLWF;

-- Create the sales table
CREATE TABLE sales (
    year INT,
    country VARCHAR(50),
    product VARCHAR(50),
    profit INT
);

-- Insert data into the sales table
INSERT INTO sales (year, country, product, profit) VALUES
(2000, 'Finland', 'Computer', 1500),
(2000, 'Finland', 'Phone', 100),
(2001, 'Finland', 'Phone', 10),
(2000, 'India', 'Calculator', 75),
(2000, 'India', 'Calculator', 75),
(2000, 'India', 'Computer', 1200),
(2000, 'USA', 'Calculator', 75),
(2000, 'USA', 'Computer', 1500),
(2001, 'USA', 'Calculator', 50),
(2001, 'USA', 'Computer', 1500),
(2001, 'USA', 'Computer', 1200),
(2001, 'USA', 'TV', 150),
(2001, 'USA', 'TV', 100);
```

SQL Window Functions:

Window functions are powerful SQL features that allow you to perform calculations across a set of rows related to the current row, without collapsing those rows through grouping.

1. Introduction to Window Functions

Window functions perform calculations across a "window" of rows determined by the `OVER` clause, without actually grouping those rows together as a regular aggregate function would.

```sql
-- Basic syntax:
SELECT
    column1,
    column2,
    WINDOW_FUNCTION() OVER ([PARTITION BY column] [ORDER BY column])
FROM table_name;
```

The key parts of a window function are:

- The window function itself (like SUM, AVG, ROW_NUMBER, RANK, DENSE_RANK, NTILE, LEAD, LAG)
- The OVER clause, which defines the "window" of rows
- Optional PARTITION BY, which divides rows into groups
- Optional ORDER BY, which determines the order of rows within each partition

2. Simple Aggregate Window Functions

Let's start with the most basic window functions - aggregate functions used in a window context:

```sql
-- Calculate overall average profit alongside each row
SELECT
    year,
    country,
    product,
    profit,
    AVG(profit) OVER() AS avg_profit
FROM sales;
```

This shows each sale with the average profit across ALL sales. The empty `OVER()` means "consider all rows as one window."

3. PARTITION BY: Grouping Data Without Collapsing Rows

```sql
-- Average profit by country
SELECT
    year,
    country,
    product,
    profit,
    AVG(profit) OVER(PARTITION BY country) AS country_avg_profit
FROM sales;
```

Here, we calculate the average profit separately for each country. Each row shows its own data plus the average for its country.

We can add multiple aggregate calculations:

```sql
-- Multiple aggregates by country
SELECT
    year,
    country,
    product,
    profit,
    AVG(profit) OVER(PARTITION BY country) AS country_avg_profit,
    SUM(profit) OVER(PARTITION BY country) AS country_total_profit,
    COUNT(*) OVER(PARTITION BY country) AS country_sales_count
FROM sales;
```

4. ORDER BY: Creating Running Totals and Moving Averages

When we add ORDER BY to a window function, we create running calculations:

```sql
-- Running total of profit by year
SELECT
    year,
    country,
    product,
    profit,
    SUM(profit) OVER(ORDER BY year) AS running_total
FROM sales;

-- because of the order by clause it is doing the commulative/running total
```

This shows the cumulative sum of profit as we move through years.

We can combine PARTITION BY and ORDER BY:

```sql
-- Running total by country and year
SELECT
    year,
    country,
    product,
    profit,
    SUM(profit) OVER(PARTITION BY country ORDER BY year) AS country_running_total
FROM sales;
```

This calculates a separate running total for each country.

5. Ranking Functions

Window functions include specialized ranking functions:

ROW_NUMBER(): Assigns unique sequential numbers

```sql
-- Assign row numbers to each sale
SELECT
    year,
    country,
    product,
    profit,
    ROW_NUMBER() OVER(ORDER BY profit DESC) AS profit_rank
FROM sales;
```

This numbers each row sequentially based on profit (highest first).

RANK() and DENSE_RANK(): Handle ties differently

```sql
-- Compare different ranking functions
SELECT
    year,
    country,
    product,
    profit,
    ROW_NUMBER() OVER(ORDER BY profit DESC) AS row_num,
    RANK() OVER(ORDER BY profit DESC) AS rank_with_gaps,
    DENSE_RANK() OVER(ORDER BY profit DESC) AS dense_rank
FROM sales;
```

- `ROW_NUMBER()` gives unique numbers (1,2,3,4...)
- `RANK()` assigns the same rank to ties, but skips numbers (1,1,3,4...)
- `DENSE_RANK()` assigns the same rank to ties without skipping (1,1,2,3...)

You can also rank within partitions:

```sql
-- Rank products by profit within each country
SELECT
    year,
    country,
    product,
    profit,
    RANK() OVER(PARTITION BY country ORDER BY profit DESC) AS
profit_rank_in_country
FROM sales;
```

6. NTILE(): Dividing Results into Equal Groups

```sql
-- Divide all sales into 3 profit buckets
SELECT
    year,
    country,
    product,
    profit,
    NTILE(3) OVER(ORDER BY profit) AS profit_tercile
FROM sales;
```

This assigns each row to one of 3 approximately equal-sized buckets (low, medium, high profit).

How can LEAD and LAG help us?

-> `LEAD()` and `LAG()` are **time-series window functions** which let us grab values from subsequent or previous records relative to the position of the "current" record in our data.
-> They can be useful any time we want to compare a value in a given column to the next or previous value in the same column — but side by side, in the same row.
-> This is a very common problem in real-world analytics scenarios.

1. How to use the LEAD() function to access data from the next row in the same result set without using a self-join?

```sql
SELECT
    year,
    country,
    product,
    profit,
    LEAD(profit, 1) OVER (PARTITION BY country ORDER BY year, product) AS
next_profit
FROM
    sales;
```

- **LEAD(profit, 1)**: This specifies that we want to access the `profit` value from the next row (1 row ahead).
- **PARTITION BY country**: This divides the result set into partitions by `country`.
- **ORDER BY year, product**: This specifies the order of the rows within each partition.

year	country	product	profit	next_profit
2000	Finland	Computer	1500	100
2000	Finland	Phone	100	10
2001	Finland	Phone	10	
2000	India	Calculator	75	75
2000	India	Calculator	75	1200
2000	India	Computer	1200	
2000	USA	Calculator	75	1500
2000	USA	Computer	1500	50
2001	USA	Calculator	50	1500
2001	USA	Computer	1500	1200
2001	USA	Computer	1200	150
2001	USA	TV	150	100
2001	USA	TV	100	

7. Lead and Lag Functions: Accessing Previous and Next Rows

These functions let you access values from other rows relative to the current row:

```sql
-- Compare each sale with the next and previous sale
SELECT
    year,
    country,
    product,
    profit,
    LAG(profit) OVER(ORDER BY year, country, product) AS previous_profit,
    LEAD(profit) OVER(ORDER BY year, country, product) AS next_profit
FROM sales;
```

You can calculate differences between rows:

```sql
-- Calculate profit difference from previous sale
SELECT
    year,
    country,
    product,
    profit,
    profit - LAG(profit, 1, 0) OVER(ORDER BY year, country, product) AS
profit_change
FROM sales;
```

The second parameter (1) is the offset, and the third parameter (0) is the default value if there's no previous row.

8. Using Window Functions in Analytical Queries

Calculating percentage of total

```sql
-- Calculate what percentage each sale contributes to country's total profit
SELECT
    year,
    country,
    product,
    profit,
    profit / SUM(profit) OVER(PARTITION BY country) * 100 AS
percent_of_country_profit
FROM sales;
```

Finding products with above-average profits

```sql
-- Find sales with above-average profit
SELECT
    year,
    country,
    product,
    profit,
    AVG(profit) OVER() AS avg_profit
FROM sales
WHERE profit > (SELECT AVG(profit) FROM sales);
```

Limiting by rank using subqueries

```sql
-- Find the top 2 most profitable products in each country
WITH RankedSales AS (
    SELECT
        year,
        country,
        product,
        profit,
        RANK() OVER(PARTITION BY country ORDER BY profit DESC) AS profit_rank
    FROM sales
)
SELECT
    year,
    country,
    product,
    profit
FROM RankedSales
WHERE profit_rank ≤ 2;
```

9. Window Function Limitations

- Window functions can only appear in the SELECT or ORDER BY clauses
- Window functions are processed after regular aggregations, WHERE, and GROUP BY
- Window functions cannot be nested directly, though you can use subqueries or CTEs

10. Summary and Best Practices

Window functions provide powerful analytical capabilities:

- Use empty OVER() for whole-table calculations
- Use PARTITION BY similar to GROUP BY, but without reducing rows
- Add ORDER BY for running totals and cumulative values

- Use ranking functions for position-based analysis
- LEAD/LAG help with time-series and sequential analyses

Window functions greatly improve SQL's analytical capabilities by allowing you to perform complex calculations while preserving the detail rows of your data, avoiding the need for self-joins or subqueries in many cases.