---

📌 **Window Functions en SQL**

Las **Window Functions** son funciones avanzadas que permiten realizar cálculos sobre un subconjunto de filas relacionadas con la fila actual, **sin necesidad de agrupar la tabla completa** (como lo hace GROUP BY).

📌 **Ventajas principales:**
✅ Mantienen acceso a todas las filas de la consulta.
✅ Permiten hacer cálculos acumulativos, rankings y comparaciones entre filas.
✅ Se ejecutan después de aplicar WHERE, GROUP BY y HAVING.

---

🔹 **Estructura General**

función() OVER (PARTITION BY columna ORDER BY columna)

📌 **Elementos clave:**

- función() → Cualquier función de ventana (SUM(), AVG(), RANK(), etc.).
- OVER() → Define cómo se aplicará la función.
- **PARTITION BY** (opcional) → Divide las filas en grupos lógicos.
- **ORDER BY** (opcional) → Ordena las filas dentro de cada grupo.

---

🏆 **1. Funciones de Ranking (Ordenar y Enumerar Filas)**

Sirven para asignar números a filas según un criterio de ordenación.

🔹 **ROW_NUMBER()** → Asigna un número de fila único.
🔹 **RANK()** → Similar a ROW_NUMBER(), pero permite empates y deja huecos en los números.
🔹 **DENSE_RANK()** → Como RANK(), pero **no deja huecos** en los números cuando hay empates.

📌 **Ejemplo: Ranking de empleados por salario**

```sql
SELECT nombre, salario,
       ROW_NUMBER() OVER (ORDER BY salario DESC) AS fila,
       RANK() OVER (ORDER BY salario DESC) AS ranking,
       DENSE_RANK() OVER (ORDER BY salario DESC) AS dense_rank_ejemplo
FROM empleados;
```

📌 **Ejemplo de salida:**

| Nombre | Salario | ROW_NUMBER | RANK | DENSE_RANK |
|---|---|---|---|---|
| **Pedro** | 4000 | 1 | 1 | 1 |
| **Luis** | 3500 | 2 | 2 | 2 |
| **Sofía** | 2800 | 3 | 3 | 3 |
| **Marta** | 2700 | 4 | 4 | 4 |
| **Ana** | 2500 | 5 | 5 | 5 |

📌 **Diferencias clave:**

- ROW_NUMBER() **no repite números, aunque haya empates**.
- RANK() **salta números** cuando hay empates.
- DENSE_RANK() **no salta números en empates**.

---

💰 **2. Funciones de Agregación con Ventanas (Cálculo Acumulado)**

Estas funciones permiten hacer cálculos sobre grupos de filas, pero **sin colapsarlas en una sola fila** como lo haría GROUP BY.

🔹 **SUM()** → Calcula un acumulado de valores.
🔹 **AVG()** → Obtiene el promedio dentro de la ventana.
🔹 **MIN() / MAX()** → Encuentra el mínimo o máximo.

📌 **Ejemplo: Salario total por departamento**

```
SELECT nombre, departamento, salario,
       SUM(salario) OVER (PARTITION BY departamento) AS salario_total
FROM empleados;
```

📌 **Salida esperada:**

| Nombre | Departamento | Salario | Salario_Total |
|---|---|---|---|
| **Ana** | Ventas | 2500 | 5300 |
| **Sofía** | Ventas | 2800 | 5300 |
| **Luis** | IT | 3500 | 7500 |
| **Pedro** | IT | 4000 | 7500 |
| **Marta** | Finanzas | 2700 | 2700 |

📌 **Explicación:**

- **PARTITION BY departamento** → Calcula SUM(salario) pero *sin agrupar los datos*.
- **Cada fila sigue apareciendo en la consulta** con el total acumulado de su departamento.

---

🔄 **3. Funciones de Desplazamiento (Comparar Filas)**

Sirven para comparar valores en diferentes filas dentro de la misma ventana.

🔹 **LAG()** → Obtiene el valor de la fila **anterior**.
🔹 **LEAD()** → Obtiene el valor de la fila **siguiente**.

📌 **Ejemplo: Comparar salarios con el anterior y el siguiente**

```
SELECT nombre, salario,
       LAG(salario) OVER (ORDER BY salario) AS anterior,
       LEAD(salario) OVER (ORDER BY salario) AS siguiente
FROM empleados;
```

📌 **Salida esperada:**

| Nombre | Salario | Anterior | Siguiente |
|---|---|---|---|
| **Ana** | 2500 | NULL | 2700 |
| **Marta** | 2700 | 2500 | 2800 |
| **Sofía** | 2800 | 2700 | 3500 |
| **Luis** | 3500 | 2800 | 4000 |
| **Pedro** | 4000 | 3500 | NULL |

📌 **Explicación:**

- LAG(salario) muestra el salario de la fila anterior.
- LEAD(salario) muestra el salario de la fila siguiente.

---

🔥 **Ejercicios para Practicar**

🔹 **Ejercicio 1:** Encuentra el ranking de ventas por cliente.

```sql
SELECT ClienteID, SUM(venta) AS total_ventas,
       RANK() OVER (ORDER BY SUM(venta) DESC) AS ranking
FROM ventas
GROUP BY ClienteID;
```

🔹 **Ejercicio 2:** Muestra el acumulado de ventas por mes.

```sql
SELECT Mes, SUM(venta) AS total,
       SUM(venta) OVER (ORDER BY Mes) AS acumulado
FROM ventas;
```

🔹 **Ejercicio 3:** Obtén el salario anterior y siguiente de cada empleado.

```sql
SELECT nombre, salario,
       LAG(salario) OVER (ORDER BY salario) AS anterior,
       LEAD(salario) OVER (ORDER BY salario) AS siguiente
FROM empleados;
```

---

📌 **Resumen Final**

✅ **Window Functions permiten hacer cálculos sin perder detalles de las filas.**
✅ **PARTITION BY agrupa datos sin colapsarlos.**
✅ **ORDER BY define el orden dentro de la ventana.**
✅ **Ranking, acumulados y comparación entre filas son los usos más comunes.**

📌 **¿Qué parte te gustaría practicar más?** 🚀

Las **Window Functions** en SQL son fundamentales en el análisis de datos porque permiten hacer cálculos avanzados sin perder el detalle de cada fila. Sirven para responder preguntas clave como tendencias, rankings, acumulados y comparaciones entre filas sin necesidad de agrupar los datos.

---

🔹 **¿Para qué se usan en Análisis de Datos?**

📌 **1. Cálculo de Acumulados y Tendencias**

Sirven para ver cómo evolucionan las ventas, ingresos o cualquier métrica a lo largo del tiempo.

👉 **Ejemplo: Ventas Acumuladas Mensuales**

```sql
SELECT Mes, SUM(venta) AS total_ventas,
       SUM(venta) OVER (ORDER BY Mes) AS ventas_acumuladas
FROM ventas;
```

📌 **Esto ayuda a ver si las ventas están creciendo o decreciendo.**

---

📌 **2. Ranking y Clasificación**

Son útiles para encontrar los mejores clientes, productos más vendidos, empleados con mejor desempeño, etc.

👉 **Ejemplo: Ranking de Clientes según sus Compras**

```sql
SELECT ClienteID, SUM(venta) AS total_gastado,
       RANK() OVER (ORDER BY SUM(venta) DESC) AS ranking
FROM ventas
GROUP BY ClienteID;
```

📌 **Esto permite identificar a los clientes más valiosos.**

---

📌 **3. Comparación entre Filas (Análisis Temporal)**

Permiten comparar datos actuales con valores anteriores o siguientes, sin hacer JOIN.

👉 **Ejemplo: Comparar las Ventas de Cada Mes con el Mes Anterior**

```sql
SELECT Mes, SUM(venta) AS total_ventas,
       LAG(SUM(venta)) OVER (ORDER BY Mes) AS ventas_mes_anterior,
       LEAD(SUM(venta)) OVER (ORDER BY Mes) AS ventas_mes_siguiente
FROM ventas
GROUP BY Mes;
```

📌 **Esto ayuda a entender la evolución y detectar caídas o aumentos de ventas.**

---

### 📌 4. Cálculo de Promedios Móviles

Se usan para suavizar fluctuaciones en los datos y encontrar tendencias más claras.

### 👉 Ejemplo: Promedio Móvil de Ventas en los Últimos 3 Meses

```sql
SELECT Mes, SUM(venta) AS total_ventas,
       AVG(SUM(venta)) OVER (ORDER BY Mes ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS promedio_movil
FROM ventas
GROUP BY Mes;
```

### 📌 Esto es útil para análisis de series temporales y pronósticos.

---

### 🔥 ¿Por qué son tan importantes en Análisis de Datos?

✅ **Evitan hacer múltiples JOIN o subqueries** → Código más limpio y eficiente.
✅ **Permiten cálculos avanzados sin perder detalle de cada fila.**
✅ **Facilitan la detección de tendencias, anomalías y patrones en los datos.**
✅ **Son esenciales en reportes y dashboards para análisis dinámico.**

### 📌 ¿En qué tipo de análisis de datos quieres aplicarlas? 🚀