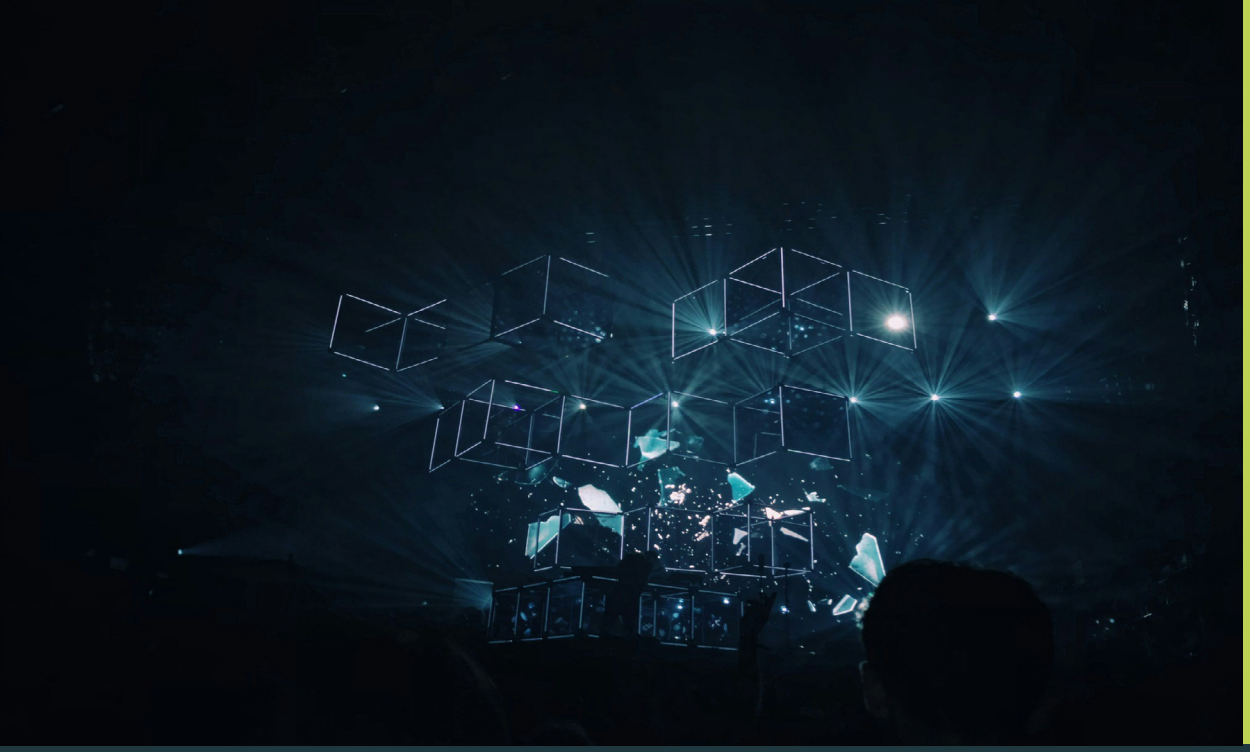

Unidad 6

Lenguaje SQL: DQL unitabla

Contenidos

6. Lenguaje SQL: DQL unitabla

1. Introducción a SQL DQL

El **lenguaje de consulta de datos** (**DQL**) es un subconjunto del **lenguaje de consulta estructurado** (**SQL**) que se utiliza para recuperar datos de una base de datos. DQL es esencial para cualquier persona que trabaje con bases de datos, ya que permite consultar información de manera eficiente y flexible. Las sentencias DQL son fundamentales en el análisis de datos, ya que permiten extraer información específica de las tablas.

Las sentencias **DQL** se utilizan exclusivamente para consultar datos. No se emplean para modificar datos ni para definir esquemas o estructuras de bases de datos. Su único propósito es recuperar información para su análisis o uso posterior.

2. SELECT

La sentencia `SELECT` es la piedra angular del **DQL** (Data Query Language), el sublenguaje de **SQL** utilizado para recuperar datos de una base de datos. Es la instrucción más común cuando trabajamos con bases de datos, ya que nos permite obtener la información que necesitamos de las tablas.

2.1. Sintaxis básica

La estructura fundamental de una sentencia `SELECT` es simple y directa:

```
SELECT columnas
FROM nombre_de_tabla;
```

- `SELECT`: Indica que deseas recuperar información de la base de datos.
- `columnas`: Especificas os nombres de las columnas que quieres recuperar. Si deseas obtener todas las columnas de la tabla, puedes usar el asterisco () como comodín.
- `FROM`: Define la tabla de la que quieres obtener los datos.

2.2. Ejemplos

Consideremos una tabla llamada **empleados** con las siguientes columnas: **id**, **nombre**, **puesto**, y **salario**.

- Para seleccionar todos los datos de la tabla **empleados**:

```
SELECT *
FROM empleados;
```

Este ejemplo devolverá todas las columnas y todas las filas de la tabla. Sin embargo, el uso del `*` implica que se recuperarán todos los campos disponibles, lo que puede no ser óptimo si la tabla crece con el tiempo. Para un control más preciso, se recomienda especificar solo las columnas necesarias:

```
SELECT nombre, puesto
FROM empleados;
```

Este ejemplo recupera solo las columnas **nombre** y **puesto** de la tabla **empleados**.

2.3. Cláusulas adicionales

Aunque la sintaxis básica de `SELECT` es suficiente para consultas simples, se vuelve mucho más poderosa cuando se combina con otras cláusulas, que permiten refinar la búsqueda y procesar los datos de manera más efectiva:

- `WHERE`: Filtra las filas recuperadas según una condición.

```
SELECT nombre, salario
FROM empleados
WHERE salario > 3000;
```

- `ORDER BY`: Ordena los resultados según una o varias columnas, en orden ascendente (**ASC**) o descendente (**DESC**).

```
SELECT nombre, edad
FROM empleados
ORDER BY edad DESC;
```

- `GROUP BY`: Agrupa filas con valores idénticos en una o más columnas, comúnmente usado junto con funciones agregadas (como `SUM`, `AVG`, `COUNT`) para realizar cálculos sobre los grupos.

```
SELECT departamento, COUNT(*) AS num_empleados
FROM empleados
GROUP BY departamento;
```

- `HAVING`: Filtra los grupos creados por `GROUP BY` según una condición específica.

```
SELECT departamento, AVG(salario) AS salario_promedio
FROM empleados
GROUP BY departamento
HAVING AVG(salario) > 4000;
```

2.4. Subconsultas

La sentencia `SELECT` también permite **subconsultas**, que son consultas anidadas dentro de otra consulta. Esto proporciona mayor flexibilidad al filtrar y manipular los datos.

```
  SELECT nombre, salario
  FROM empleados
  WHERE departamento_id = (SELECT id FROM departamentos WHERE nombre_departamento =
'Ventas');
```

En este ejemplo, la subconsulta `(SELECT id FROM departamentos WHERE nombre_departamento = 'Ventas')` obtiene el **id** del departamento de **Ventas**, y ese valor se utiliza en la consulta principal para filtrar los empleados que pertenecen a dicho departamento.

3. WHERE

La cláusula `WHERE` en **SQL** se utiliza para filtrar los resultados de una consulta `SELECT`. Permite especificar condiciones que deben cumplirse para que las filas sean incluidas en el conjunto de resultados, proporcionando una manera eficaz de recuperar solo los datos relevantes.

3.1. Sintaxis básica

```
  sql
  Copiar código
  SELECT columnas
  FROM nombre_de_tabla
  WHERE condición;
```

- `condición`: Es una expresión que debe evaluarse como verdadera o falsa. Esta puede involucrar operadores de comparación, operadores lógicos, funciones y otras expresiones complejas.

3.2. Ejemplos

- Para obtener los empleados cuyo puesto es **'Gerente'**:

```
  SELECT nombre, puesto
  FROM empleados
  WHERE puesto = 'Gerente';
```

Este ejemplo devolverá el **nombre** y el **puesto** de los empleados cuyo puesto es "Gerente".

- Para obtener los empleados cuyo **salario** es mayor a **3000**:

```
  SELECT nombre, salario
  FROM empleados
  WHERE salario > 3000;
```

Este ejemplo devolverá el **nombre** y el **salario** de los empleados cuyo salario sea superior a 3000.

3.3. Operadores en la cláusula WHERE

La cláusula `WHERE` puede incluir una variedad de operadores que permiten realizar comparaciones más complejas:

- **Operadores de comparación**:
  - `=`: Igual a.
  - `>`: Mayor que.
  - `<`: Menor que.
  - `>=`: Mayor o igual que.
  - `<=`: Menor o igual que.
  - `<>` **o** `!=`: Distinto de.

    Ejemplo:

```
SELECT nombre, salario
FROM empleados
WHERE salario >= 4000;
```

- **Operadores lógicos**:
  - `AND`: Combina dos condiciones que deben cumplirse ambas.
  - `OR`: Combina dos condiciones, de las cuales al menos una debe cumplirse.
  - `NOT`: Rechaza una condición.

    Ejemplo:

```
SELECT nombre, puesto, salario
FROM empleados
WHERE salario > 3000 AND puesto = 'Gerente';
```

    Este ejemplo devolverá los empleados con un salario mayor a 3000 y cuyo puesto es **'Gerente'**.

- **Operadores de rango**:
  - `BETWEEN`: Selecciona valores dentro de un rango determinado.

    Ejemplo:

```
SELECT nombre, salario
FROM empleados
WHERE salario BETWEEN 2000 AND 5000;
```

    Este ejemplo selecciona a los empleados cuyo salario está entre 2000 y 5000.

- **Operadores de conjunto**:
  - `IN`: Permite seleccionar valores dentro de un conjunto de opciones.

Ejemplo:

```
SELECT nombre, puesto
FROM empleados
WHERE puesto IN ('Gerente', 'Director');
```

Este ejemplo selecciona empleados cuyo puesto es **'Gerente'** o **'Director'**.

- **Operadores de patrones**:
  - `LIKE`: Se utiliza para buscar un patrón específico en una columna de texto.
  - `%`: Comodín que representa cualquier número de caracteres.
  - `_`: Comodín que representa un solo carácter.

    Ejemplo:

```
SELECT nombre
FROM empleados
WHERE nombre LIKE 'J%';
```

Este ejemplo devolverá todos los empleados cuyos nombres comienzan con la letra **'J'**.

3.4. Uso de `IS NULL`

La cláusula `WHERE` también puede utilizarse para filtrar registros con valores `NULL`.

```
SELECT nombre, puesto
FROM empleados
WHERE puesto IS NULL;
```

Este ejemplo devolverá los empleados cuyo **puesto** no está asignado, es decir, aquellos que tienen un valor **NULL** en la columna **puesto**.

4. Operadores aritméticos

Los **operadores aritméticos** en **SQL** permiten realizar operaciones matemáticas básicas con valores numéricos. Son similares a los operadores utilizados en matemáticas convencionales y se emplean en diversas partes de una consulta **SQL**, como en la cláusula `SELECT` para calcular nuevos valores, en la cláusula `WHERE` para filtrar datos, o en otras cláusulas para realizar cálculos sobre los registros.

4.1. Operadores

- **Suma (+):** Suma dos valores.

```
SELECT precio + 10 AS precio_incrementado FROM productos;
```

- **Resta (-):** Resta un valor de otro.

```
SELECT precio - descuento FROM productos;
```

- **Multiplicación (*):** Multiplica dos valores.

```
SELECT cantidad * precio AS total FROM ventas;
```

- **División (/):** Divide un valor entre otro.

```
SELECT total / cantidad AS precio_unitario FROM ventas;
```

Módulo (%): Devuelve el resto de una división entera.

```
SELECT id % 2 AS es_par FROM productos;
```

4.2. Ejemplos

- **Calcular el precio total de un producto con un 10% de descuento:**

```
SELECT nombre_producto, precio * (1 - 0.10) AS precio_con_descuento FROM productos;
```

- **Calcular el promedio de dos valores:**

```
SELECT (valor1 + valor2) / 2 AS promedio FROM datos;
```

- **Comprobar si un número es divisible por 3:**

```
SELECT numero FROM numeros WHERE numero % 3 = 0;
```

4.3. Uso común de los operadores aritméticos

Los operadores aritméticos en **SQL** son fundamentales para realizar cálculos y transformaciones de datos en tiempo real. Algunos casos comunes incluyen:

- **Cálculos de totales y promedios:** Usar operadores como `+`, , y `/` para sumar, restar, multiplicar o dividir valores numéricos, lo cual es esencial para generar informes financieros o estadísticas.
- **Cálculos de márgenes o descuentos:** El operador es útil para aplicar porcentajes de descuento o calcular márgenes de ganancia sobre precios base.
- **Filtrado de datos con el operador módulo (%):** Este operador permite identificar elementos en función de su divisibilidad, lo cual es útil para filtrados específicos como múltiplos o divisores.

4.4. Consideraciones adicionales

- **Precisión en la división:** Cuando se usan divisiones, es importante tener en cuenta la precisión del resultado, especialmente si los operandos son números enteros. Para obtener decimales, asegúrate de usar tipos de datos numéricos adecuados como `FLOAT` o `DECIMAL`.
- **Uso de paréntesis:** En operaciones más complejas, utiliza **paréntesis** para garantizar que las operaciones se realicen en el orden correcto.

5. Operadores de comparación o búsqueda

Los **operadores de comparación** en **SQL** se utilizan para comparar valores en una base de datos. Normalmente devuelven un valor booleano, es decir, **verdadero** o **falso**, y se emplean principalmente en la cláusula `WHERE` para filtrar los resultados de una consulta.

5.1. Operadores de comparación

- **Igual (=):** Comprueba si dos valores son iguales.

  Ejemplo: (op1 = op2) es falso.

- **Desigual (!= o <>):** Comprueba si dos valores son diferentes.

  Ejemplo: (op1 != op2) es cierto.

- **Mayor que (>):** Comprueba si un valor es mayor que otro.

  Ejemplo: (op1 > op2) es cierto.

- **Menor que (<):** Comprueba si un valor es menor que otro.

  Ejemplo: (op1 < op2) es falso.

- **Mayor o igual que (>=):** Comprueba si un valor es mayor o igual que otro.

  Ejemplo: (op1 >= op2) es cierto.

- **Menor o igual que (<=):** Comprueba si un valor es menor o igual que otro.

  Ejemplo: (op1 <= op2) es falso.

5.2. Operadores de búsqueda

- **BETWEEN:** Comprueba si un valor está dentro de un rango especificado.

  Ejemplo:

```
SELECT * FROM tabla WHERE columna BETWEEN 10 AND 20;
```

  Selecciona filas donde el valor de la columna esté entre 10 y 20.

- **IN:** Comprueba si un valor está en una lista de valores especificados.

  Ejemplo:

```
SELECT * FROM tabla WHERE columna IN (valor1, valor2, valor3);
```

  Selecciona filas donde el valor de la columna está en la lista de valores especificada.

- **LIKE:** Comprueba si un valor coincide con un patrón.

  Ejemplo:

```
SELECT * FROM tabla WHERE columna LIKE 'a%';
```

Selecciona filas donde el valor de la columna comienza con la letra 'a'.

- **IS NULL:** Comprueba si un valor es nulo.

  Ejemplo:

```
SELECT * FROM tabla WHERE columna IS NULL;
```

Selecciona filas donde el valor de la columna es **NULL**.

5.3. Ejemplos

- Para encontrar todos los empleados con un salario mayor a 50.000:

```
SELECT * FROM empleados WHERE salario > 50000
```

- Para encontrar todos los empleados contratados entre el 1 de enero de 2023 y el 31 de diciembre de 2023:

```
SELECT * FROM empleados WHERE fecha_contratacion BETWEEN '2023-01-01' AND '2023-12-
31';
```

- Para encontrar todos los empleados que trabajan en los departamentos de ventas o marketing:

```
SELECT * FROM empleados WHERE departamento IN ('Ventas', 'Marketing');
```

- Para encontrar todos los empleados cuyo nombre empieza con ‘A’:

```
SELECT * FROM empleados WHERE nombre LIKE 'A%';
```

- Para encontrar todos los empleados que no tienen un número de teléfono:

```
SELECT * FROM empleados WHERE telefono IS NULL;
```

5.4. Operadores adicionales

SQL ofrece operadores adicionales que permiten realizar búsquedas más específicas:

- **EXISTS:** Verifica si existe al menos una fila que cumpla con la condición de una subconsulta.

  Ejemplo:

```
SELECT * FROM tabla WHERE EXISTS (SELECT 1 FROM otra_tabla WHERE columna = valor);
```

Este ejemplo selecciona filas de `tabla` si existe al menos una fila en `otra_tabla` que cumpla la condición `columna = valor`.

- **UNIQUE:** Verifica si una columna contiene solo valores únicos.

  Ejemplo:

```
SELECT UNIQUE columna FROM tabla;
```

  Este ejemplo selecciona los valores únicos de la columna `columna` en la tabla `tabla`.

- **ALL:** Compara un valor con todos los valores de un conjunto devuelto por una subconsulta.

  Ejemplo:

```
SELECT * FROM tabla WHERE columna > ALL (SELECT columna FROM otra_tabla);
```

  Este ejemplo selecciona filas de `tabla` donde el valor de `columna` es mayor que todos los valores de `columna` en `otra_tabla`.

- **ANY:** Compara un valor con cualquiera de los valores devueltos por una subconsulta.

  Ejemplo:

```
SELECT * FROM tabla WHERE columna = ANY (SELECT columna FROM otra_tabla);
```

  Este ejemplo selecciona filas de `tabla` donde el valor de `columna` coincide con cualquier valor de `columna` en `otra_tabla`.

5.5. Comodines

Los **comodines** en **SQL** se utilizan para buscar valores con patrones específicos, principalmente con el operador `LIKE`.

- `%`: Representa **cero o más caracteres**.

  Ejemplo:

```
SELECT * FROM tabla WHERE columna LIKE '%promoción%';
```

  Selecciona filas donde el valor de la columna contiene la palabra "promoción".

- `_`: Representa **un solo carácter**.

  Ejemplo:

```
SELECT * FROM tabla WHERE columna LIKE 'C_12';
```

  Selecciona filas donde el valor de la columna comienza con 'C', seguido de cualquier carácter, y termina con '12'.

Estos operadores y comodines permiten realizar consultas más avanzadas y específicas en bases de datos SQL.

6. Operadores lógicos

Los **operadores lógicos** en **SQL** permiten combinar o modificar condiciones en una **cláusula** `WHERE`, evaluando las condiciones y devolviendo un valor **booleano** (verdadero o falso). Son esenciales para la toma de decisiones en consultas, ya que permiten crear expresiones más complejas.

Es importante tener claridad al utilizarlos, ya que pueden complicar las consultas cuando se emplean muchas condiciones. Por eso, se recomienda mantener un código ordenado e indentado correctamente.

6.1. Operadores lógicos comunes

- **AND:** Devuelve **verdadero** solo si todas las condiciones son verdaderas.

  Ejemplo:

```
SELECT * FROM empleados WHERE edad > 30 AND salario > 50000;
```

  En este caso, se seleccionan los empleados cuya edad sea mayor a 30 **y** cuyo salario sea superior a 50,000.

- **OR:** Devuelve **verdadero** si al menos una de las condiciones es verdadera.

  Ejemplo:

```
SELECT * FROM empleados WHERE departamento = 'Ventas' OR departamento = 'Marketing';
```

  Se seleccionan todos los empleados que trabajen en el departamento de **Ventas o** en el de **Marketing**. Con que se cumpla cualquiera de las dos condiciones, la fila se incluirá en el resultado.

- **NOT:** Invierte el resultado lógico de una condición. Si una condición es **verdadera**, `NOT` la convierte en **falsa**, y viceversa.

  Ejemplo:

```
SELECT * FROM empleados WHERE NOT departamento = 'Finanzas'
```

  En este caso, se seleccionan todos los empleados que **no** trabajan en el departamento de **Finanzas**.

6.2. Combinación de operadores lógicos

A menudo, los operadores lógicos se combinan para formar condiciones más complejas. El uso adecuado de paréntesis puede controlar el orden de las evaluaciones y garantizar resultados precisos.

Ejemplo de combinación de `AND`, `OR` y `NOT`:

```
SELECT * FROM empleados
WHERE (edad > 30 AND salario > 50000) OR NOT departamento = 'Finanzas';
```

En este caso, la consulta selecciona a los empleados que cumplen con **ambas condiciones** (edad mayor a 30 y salario mayor a 50,000) **o** aquellos que **no** trabajan en el departamento de **Finanzas**. El uso de paréntesis asegura que las condiciones de **AND** se evalúen antes que **OR**.

6.3. Prioridad de los operadores lógicos

Es crucial entender la **prioridad** de los operadores lógicos al combinar múltiples condiciones. La **prioridad de evaluación** generalmente es:

1. **NOT**
2. **AND**
3. **OR**

Por lo tanto, `NOT` se evalúa primero, seguido de `AND`, y por último `OR`. Para cambiar este orden, se deben utilizar paréntesis.

6.4. Consejos para usar operadores lógicos

- **Claridad:** Utiliza paréntesis para evitar confusión cuando combines múltiples operadores.
- **Legibilidad:** Asegúrate de que la lógica de las condiciones sea fácil de entender, especialmente en consultas complejas.
- **Evita redundancias:** Asegúrate de que las condiciones no se repitan innecesariamente, lo que podría hacer que la consulta sea menos eficiente.

Los operadores lógicos son fundamentales para crear consultas complejas y filtradas en SQL, por lo que es esencial comprender su funcionamiento y cómo combinarlos adecuadamente.

7. Operadores de fechas

SQL ofrece varias funciones especializadas para trabajar con fechas, permitiendo realizar manipulaciones y cálculos de manera eficiente. A continuación, se detallan algunas de las funciones más comunes:

7.1. NOW()

La función **NOW()** devuelve la **fecha y hora actuales** del sistema, con el formato habitual de `YYYY-MM-DD HH:MM:SS`. El formato exacto puede variar dependiendo del **sistema de gestión de bases de datos (SGBD)**.

Consulta:

```
SELECT NOW();
```

Resultado:

```
2024-12-26 14:23:45
```

7.2. DATEDIFF()

La función **DATEDIFF()** calcula la **diferencia entre dos fechas**. El resultado se expresa en días, aunque algunos SGBD permiten calcular la diferencia en otras unidades, como meses o años.

Consulta:

```
SELECT DATEDIFF('2024-12-31', '2024-10-01') AS diferencia_dias;
```

Resultado:

```
91
```

7.3. DATE_FORMAT()

La función **DATE_FORMAT()** permite **formatear** una fecha según el formato especificado, lo que facilita mostrarla de diferentes maneras, como `DD/MM/YYYY`, `MM-DD-YYYY`, o `YYYY-MM-DD`.

Consulta:

```
SELECT DATE_FORMAT(NOW(), '%d/%m/%Y') AS fecha_formateada;
```

Resultado:

```
26/12/2024
```

7.4. Funciones adicionales para fechas

SQL ofrece más funciones útiles para manipular fechas, entre ellas:

- **DATE_ADD()**: Añade un intervalo de tiempo a una fecha específica.

    Consulta:

```
SELECT DATE_ADD('2024-12-01', INTERVAL 10 DAY);
```

    Resultado:

```
2024-12-11
```

- **DATE_SUB()**: Resta un intervalo de tiempo a una fecha específica.

    Consulta:

```
SELECT DATE_SUB('2024-12-01', INTERVAL 10 DAY);
```

    Resultado:

```
2024-11-21
```

- **YEAR(), MONTH(), DAY()**: Estas funciones extraen el **año**, **mes** o **día** de una fecha, respectivamente.

Consulta:

```
  SELECT YEAR(NOW()) AS año_actual, MONTH(NOW()) AS mes_actual, DAY(NOW()) AS dia_ac-
tual;
```

Resultado:

```
  año_actual: 2024
  mes_actual: 12
  dia_actual: 26
```

Estas funciones permiten realizar manipulaciones de fechas de manera flexible, como agregar o restar días, meses o años, o extraer partes específicas de una fecha, lo que facilita el manejo de datos temporales en consultas SQL.

8. Campos calculados

En SQL, es posible crear **campos calculados** a partir de los campos existentes en las tablas, utilizando operadores y funciones dentro de la cláusula `SELECT`. Esto permite realizar cálculos directamente en la consulta sin necesidad de modificar los datos en las tablas.

8.1. Ejemplo de cálculo de campos

Supongamos que tenemos una tabla llamada `empleados` con un campo denominado `sueldoMensual`, y deseamos calcular el sueldo anual de los empleados en 12 y 14 pagas. Podemos hacerlo fácilmente en la consulta `SELECT` creando nuevos campos calculados.

Consulta:

```
  SELECT
      nombreEmpleado,
      apellidoEmpleado,
      sueldoMensual,
      sueldoMensual * 12 AS sueldoAnual_12pagas,
      sueldoMensual * 14 AS sueldoAnual_14pagas
  FROM empleado;
```

En este ejemplo, se generan dos nuevos campos calculados: `sueldoAnual_12pagas` y `sueldoAnual_14pagas`, multiplicando el valor de `sueldoMensual` por 12 y 14, respectivamente. Estos cálculos se realizan directamente en la consulta y no es necesario realizar ningún procesamiento adicional fuera de la base de datos.

8.2. Ventajas de los campos calculados

Los **campos calculados** son una herramienta poderosa que permite realizar análisis y obtener resultados informativos de manera eficiente. Algunas ventajas clave incluyen:

- **Reducción de cálculos externos**: No es necesario calcular los valores fuera de la base de datos, ya que los cálculos se realizan directamente en la consulta.
- **Flexibilidad**: Puedes realizar operaciones matemáticas, concatenar cadenas, o aplicar funciones como `ROUND()`, `AVG()`, entre otras.

- **Optimización**: Permiten optimizar las consultas al incluir cálculos directamente en la cláusula `SELECT`, evitando el procesamiento adicional de datos en aplicaciones externas.

8.3. Otros ejemplos de campos calculados

Además de realizar multiplicaciones o sumas, los campos calculados pueden incluir operaciones más complejas, como la **concatenación de cadenas** o el uso de **funciones agregadas**.

Ejemplo de concatenación:

```
SELECT
    nombreEmpleado,
    apellidoEmpleado,
    CONCAT(nombreEmpleado, ' ', apellidoEmpleado) AS nombreCompleto
FROM empleado;
```

Resultado:

```
nombreCompleto
---------------
Juan Pérez
Maria Gómez
```

Los **campos calculados** son una herramienta esencial en SQL para obtener información derivada de manera eficiente, sin la necesidad de realizar cálculos o manipulaciones adicionales fuera de la consulta.

9. Ordenar resultados de consultas

En SQL, los resultados de una consulta no se devuelven de forma ordenada por defecto. Para organizar los datos según un criterio específico, se utiliza la cláusula `ORDER BY`, que permite ordenar los resultados por una o más columnas. Los resultados pueden ordenarse en orden **ascendente** (`ASC`) o **descendente** (`DESC`).

9.1. Sintaxis básica

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1 ASC/DESC, column2 ASC/DESC, ...;
```

- `ASC`: Ordena los resultados en orden **ascendente** (de menor a mayor).
- `DESC`: Ordena los resultados en orden **descendente** (de mayor a menor).

9.2. Ejemplo

```
  SELECT nombre, apellido, sueldoMensual, sueldoMensual * 12 AS sueldoAnual_12pagas,
sueldoMensual * 14 AS sueldoAnual_14pagas
  FROM empleado
  ORDER BY apellidoEmpleado ASC;
```

En este ejemplo, los resultados se ordenan por la columna `apellidoEmpleado` en orden ascendente (alfabéticamente), lo que permite ver a los empleados organizados por su apellido.

9.3. Consideraciones

- **Orden jerárquico**: Se pueden ordenar los resultados por varias columnas para crear un **orden jerárquico**. Por ejemplo, ordenar primero por salario y luego por apellido.

  Ejemplo:

```
  SELECT nombre, apellido, salario
  FROM empleados
  ORDER BY salario DESC, apellido ASC;
```

- **Orden por defecto**: Si no se especifica `ASC` o `DESC`, el orden predeterminado es `ASC`.
- **Aplicación de filtros y agrupaciones**: La cláusula `ORDER BY` se aplica después de que se hayan realizado los filtros con la cláusula `WHERE` o las agrupaciones con `GROUP BY`.

La cláusula `ORDER BY` es fundamental para organizar los datos y facilitar su análisis, especialmente cuando se necesitan resultados ordenados de acuerdo con ciertos criterios.

10. Funciones agregadas

Las **funciones agregadas** en SQL permiten realizar cálculos sobre un conjunto de valores y devolver un único valor como resultado. Estas funciones son útiles para obtener información resumida de los datos, como la suma, el promedio, el conteo, el valor máximo o el valor mínimo. Son comúnmente utilizadas en análisis de datos para generar estadísticas o resúmenes a partir de grandes volúmenes de información.

10.1. Funciones agregadas comunes

- **COUNT():** Cuenta el número de filas o valores no nulos en una columna.
- **SUM():** Suma los valores de una columna numérica.
- **AVG():** Calcula el promedio de los valores de una columna numérica.
- **MAX():** Obtiene el valor máximo de una columna.
- **MIN():** Obtiene el valor mínimo de una columna.

10.2. Ejemplos

- Para contar el número total de empleados en la tabla `empleados`:

```
SELECT COUNT(*) AS total_empleados FROM empleados;
```

- Para obtener la suma de los salarios de todos los empleados:

```
SELECT SUM(salario) AS suma_salarios FROM empleados;
```

- Para calcular el salario promedio de los empleados:

```
SELECT AVG(salario) AS salario_promedio FROM empleados;
```

- Para obtener el salario más alto entre los empleados:

```
SELECT MAX(salario) AS salario_maximo FROM empleados;
```

- Para obtener el salario más bajo entre los empleados:

```
SELECT MIN(salario) AS salario_minimo FROM empleados;
```

10.3. Aplicaciones de las funciones agregadas

- **Tablas agregadas**: A veces es necesario crear nuevas tablas con información agregada, como resúmenes o totales, a partir de tablas más detalladas. Las funciones agregadas permiten generar estos resúmenes eficientemente.
- **Agrupación**: Las funciones agregadas se suelen combinar con la cláusula `GROUP BY` para realizar cálculos sobre grupos de datos específicos. Por ejemplo, obtener el salario promedio por departamento.

```
SELECT departamento, AVG(salario) AS salario_promedio
FROM empleados
GROUP BY departamento;
```

Las **funciones agregadas** son esenciales para realizar análisis de datos y generar informes resumidos sin necesidad de manipular los datos fuera de la base de datos.

11. GROUP BY

La cláusula `GROUP BY` en SQL se utiliza para agrupar filas que tienen los mismos valores en una o más columnas. Se emplea frecuentemente junto con **funciones agregadas** para realizar cálculos sobre cada grupo. Esta operación permite obtener un resumen o consolidación de datos, como el total de ventas por vendedor o el salario promedio por departamento.

11.1. Sintaxis básica

```
SELECT columna1, columna2, función_agregada(columna3)
FROM nombre_tabla
GROUP BY columna1, columna2;
```

- **columna1**, **columna2**: Las columnas por las que se agrupan las filas.
- **función_agregada**: Funciones como `SUM()`, `AVG()`, `COUNT()`, `MAX()` o `MIN()` que realizan cálculos sobre los datos agrupados.

11.2. Ejemplos

Para ilustrar el uso de `GROUP BY`, consideremos las siguientes tablas de ejemplo:

Tabla 1: `empleados`

id	nombre	departamento	salario
1	Juan	Ventas	3000
2	Ana	Marketing	3500
3	Pedro	Ventas	2800
4	Laura	Finanzas	4000
5	Carlos	Marketing	3200
6	María	Ventas	3100

Tabla 2: `ventas`

id	vendedor	producto	cantidad	precio
1	Juan	A	10	20
2	Ana	B	5	30
3	Pedro	A	15	20
4	Laura	C	8	40
5	Carlos	B	12	30
6	Juan	C	7	40

1. Contar el número de empleados por departamento:

```
SELECT departamento, COUNT(*) AS num_empleados
FROM empleados
GROUP BY departamento;
```

Resultado:

departamento	num_empleados
Ventas	3
Marketing	2
Finanzas	1

2. Calcular el salario promedio por departamento:

```
SELECT departamento, AVG(salario) AS salario_promedio
FROM empleados
GROUP BY departamento;
```

Resultado:

departamento	salario_promedio
Ventas	2966.67
Marketing	3350
Finanzas	4000

3. Obtener la cantidad total de cada producto vendido:

```
SELECT producto, SUM(cantidad) AS total_vendido
FROM ventas
GROUP BY producto;
```

Resultado:

producto	total_vendido
A	25
B	17
C	15

4. Calcular el total de ingresos por vendedor:

```
SELECT vendedor, SUM(cantidad * precio) AS ingresos_totales
FROM ventas
GROUP BY vendedor;
```

Resultado:

vendedor	ingresos_totales
Juan	480
Ana	150
Pedro	300
Laura	320
Carlos	360

11.3. Consideraciones

- **Agrupación por múltiples columnas**: Se pueden agrupar los resultados por una o más columnas. El orden de las columnas en la cláusula `GROUP BY` afecta al resultado final.
- **Uso con funciones agregadas**: Las columnas que aparecen en la cláusula `SELECT` deben ser las columnas por las que se agrupa o deben estar acompañadas de funciones agregadas.
- **Aplicación posterior**: `GROUP BY` se aplica después de la cláusula `WHERE` (si se utiliza), lo que significa que primero se filtran las filas y luego se agrupan.

11.4. Ejemplo adicional:

Si tienes una tabla llamada `ventas` y deseas saber cuántas unidades ha vendido cada vendedor en total, puedes agrupar los datos por **vendedor** y sumar las cantidades de productos vendidos:

```
SELECT vendedor, SUM(cantidad) AS total_vendido
FROM ventas
GROUP BY vendedor;
```

Resultado:

vendedor	total_vendido
Juan	15
María	12
Pedro	23

La cláusula `GROUP BY` es fundamental para realizar **análisis agregados** en SQL, permitiendo simplificar el manejo de grandes cantidades de datos y generar resúmenes útiles para tomar decisiones basadas en información consolidada.

12. HAVING

La cláusula `HAVING` en SQL se utiliza para filtrar los resultados después de que se ha aplicado un `GROUP BY`. A diferencia de la cláusula `WHERE`, que permite filtrar filas antes de que se realice la agrupación, `HAVING` se utiliza para establecer condiciones en los grupos resultantes.

12.1. Sintaxis básica

```
SELECT columna1, columna2, función_agregada(columna3)
FROM nombre_tabla
GROUP BY columna1, columna2
HAVING condición;
```

- **columna1**, **columna2**: Las columnas por las que se agrupan las filas.
- **función_agregada**: Una función agregada como `SUM()`, `AVG()`, `COUNT()`, `MAX()` o `MIN()`.
- **condición**: La condición que deben cumplir los grupos para ser incluidos en el resultado.

12.2. Ejemplo

Supongamos que tienes la tabla `ventas` con la información de ventas realizadas por cada vendedor:

vendedor	producto	cantidad	precio_unitario
Juan	Laptop	5	1000
María	Smartphone	10	600
Juan	Monitor	7	300
María	Laptop	2	1000
Pedro	Teclado	15	20
Juan	Smartphone	3	600
Pedro	Monitor	8	300

Si quieres obtener solo los vendedores que hayan vendido más de 20 unidades en total, puedes utilizar `HAVING` con la función `SUM()`:

```
SELECT vendedor, SUM(cantidad) AS total_vendido
FROM ventas
GROUP BY vendedor
HAVING SUM(cantidad) > 20;
```

Resultado:

vendedor	total_vendido
Pedro	23

En este ejemplo, primero se agrupan las filas por **vendedor** y se calcula la suma de la columna **cantidad** para cada grupo. Luego, la cláusula `HAVING` filtra los grupos que tienen una `SUM(cantidad)` mayor que 20, mostrando solo el vendedor "Pedro" que cumple con esa condición.

12.3. Consideraciones

- `HAVING` se usa generalmente con funciones agregadas como `SUM()`, `AVG()`, `COUNT()`, entre otras, para establecer condiciones de filtrado en los grupos resultantes.
- A diferencia de `WHERE`, `HAVING` se aplica después de `GROUP BY`, lo que permite mayor flexibilidad para filtrar grupos sin necesidad de incluir condiciones dentro de la cláusula `SELECT`.

12.4. Ejemplo adicional:

Si deseas filtrar los grupos por una condición más específica, puedes combinar `HAVING` con múltiples condiciones:

```
SELECT vendedor, SUM(cantidad) AS total_vendido
FROM ventas
GROUP BY vendedor
HAVING SUM(cantidad) > 10 AND SUM(precio_unitario) > 1000;
```

El resultado en este caso sería mostrar aquellos vendedores que cumplan con ambas condiciones de venta total de unidades y valor acumulado superior a los límites especificados.

La cláusula `HAVING` es una herramienta poderosa en SQL para realizar filtros precisos sobre los resultados de los grupos generados con `GROUP BY`, permitiendo así un control detallado sobre los datos analizados.

13. Subconsultas

Las **subconsultas** en SQL, también conocidas como **consultas anidadas**, son consultas que se incluyen dentro de otra consulta principal. Se utilizan comúnmente para realizar cálculos complejos, filtrar datos o seleccionar información que depende de los resultados de otra consulta. Las subconsultas pueden ser utilizadas en diversas cláusulas, como `SELECT`, `FROM`, `WHERE`, entre otras.

13.1. Tipos de subconsultas

- **Subconsulta en la cláusula `WHERE`:** Se utiliza para filtrar filas de la consulta principal en función de los resultados de la subconsulta.
- **Subconsulta en la cláusula `FROM`:** También conocida como “consulta en línea”, se utiliza para crear una tabla temporal a partir de los resultados de la subconsulta, que luego se puede utilizar en la consulta principal.
- **Subconsulta en la cláusula `SELECT`:** Se utiliza para calcular un valor que se mostrará como una columna adicional en los resultados de la consulta principal.

13.2. Ejemplos

- **Subconsulta en la cláusula `WHERE`:**

```
SELECT nombre, salario
FROM empleados
WHERE departamento_id = (SELECT id FROM departamentos WHERE nombre_departamento =
'Ventas');
```

En este ejemplo, la subconsulta `(SELECT id FROM departamentos WHERE nombre_departamento = ‘Ventas’)` obtiene el **ID** del departamento "Ventas", que luego se utiliza en la consulta principal para filtrar a los empleados que pertenecen a ese departamento.

- **Subconsulta en la cláusula `FROM`:**

```
SELECT e.nombre, e.salario, d.nombre_departamento
FROM empleados e, (SELECT id, nombre_departamento FROM departamentos) d
WHERE e.departamento_id = d.id;
```

Aquí, la subconsulta `(SELECT id, nombre_departamento FROM departamentos)` crea una tabla temporal llamada `d` con la información de los departamentos. Esta tabla se utiliza en la consulta principal para obtener el nombre del departamento de cada empleado.

- **Subconsulta en la cláusula `SELECT`:**

```
SELECT nombre, salario, (SELECT AVG(salario) FROM empleados) AS salario_promedio
FROM empleados;
```

En este caso, la subconsulta `(SELECT AVG(salario) FROM empleados)` calcula el **salario promedio** de todos los empleados. Este valor se muestra luego como una columna adicional llamada `salario_promedio` en el conjunto de resultados de la consulta principal.

13.3. Subconsulta en la cláusula `WHERE`

Este es uno de los usos más comunes de las subconsultas. Se emplea para filtrar filas de la consulta principal basándose en los resultados de una subconsulta. Por ejemplo:

```
SELECT nombre, salario
FROM empleados
WHERE departamento_id = (SELECT id FROM departamentos WHERE nombre_departamento =
'Ventas');
```

En este caso, la subconsulta `(SELECT id FROM departamentos WHERE nombre_departamento = 'Ventas')` obtiene el **ID** del departamento "Ventas", y la consulta principal selecciona los empleados que pertenecen a ese departamento.

13.4. Subconsulta en la cláusula `FROM`

En ocasiones, las subconsultas pueden usarse para crear tablas temporales dentro de la cláusula `FROM`. Un ejemplo sería:

```
SELECT nombre, salario
FROM empleados
WHERE salario > (SELECT AVG(salario) FROM empleados);
```

En este ejemplo, la subconsulta `(SELECT AVG(salario) FROM empleados)` calcula el salario promedio de todos los empleados. La consulta principal selecciona a los empleados cuyo salario es superior a ese promedio.

13.5. Subconsulta en la cláusula `SELECT`

Este tipo de subconsulta se utiliza para calcular un valor específico y mostrarlo como una columna adicional en los resultados de la consulta principal. Por ejemplo:

```
SELECT nombre, salario,
       (SELECT AVG(salario) FROM empleados e2 WHERE e2.departamento_id = e1.departa-
mento_id) AS salario_promedio_departamento
FROM empleados e1;
```

En este caso, la subconsulta `(SELECT AVG(salario) FROM empleados e2 WHERE e2.departamento_id = e1.departamento_id)` calcula el **salario promedio** para cada departamento, y la consulta principal muestra el nombre del empleado, su salario y el salario promedio de su departamento.

13.6. Consideraciones importantes

- **Dependencia de resultados:** Las subconsultas permiten crear consultas que dependen de los resultados de otras, lo que las hace muy útiles en escenarios complejos de análisis de datos.
- **Optimización:** Las subconsultas pueden ser menos eficientes que las uniones o combinaciones de tablas directas, especialmente si la subconsulta devuelve grandes conjuntos de datos.
- **Subconsultas correlacionadas:** Son aquellas que hacen referencia a columnas de la consulta principal, lo que las hace más dinámicas pero también más complejas.

Las subconsultas son una herramienta poderosa en SQL para realizar operaciones complejas y obtener resultados más detallados y específicos, permitiendo estructurar consultas más dinámicas y flexibles.

14. Ejercicios

Teniendo en cuenta la siguiente tabla, contesta a los siguientes retos.

id_empleado	nombre	edad	salario	nombre_departamento
1	Juan	28	3000	Desarrollo
2	María	32	3500	Marketing
3	Pedro	45	4000	Desarrollo
4	Ana	29	3200	RRHH
5	Luis	35	3700	Marketing

- Seleccionar todos los empleados: Mostrar el nombre y salario de todos los empleados.
- Filtrar empleados por salario: Mostrar el nombre y el salario de los empleados con un salario mayor a 3500.
- Ordenar empleados por edad: Mostrar el nombre y la edad de los empleados, ordenados por edad de forma descendente.
- Contar empleados en cada departamento: Contar cuántos empleados hay en cada departamento.
- Calcular salario promedio por departamento: Mostrar el salario promedio de los empleados en cada departamento.
- Encontrar el empleado más joven: Mostrar el nombre del empleado más joven de la empresa.
- Filtrar empleados por departamento: Mostrar el nombre de los empleados que pertenecen al departamento de “Desarrollo”.
- Salario máximo en cada departamento: Mostrar el salario más alto en cada departamento.
- Buscar empleados cuyo nombre empieza con “M”
- Obtener empleados con salarios entre 3000 y 4000: Mostrar empleados cuyo salario esté entre 3000 y 4000.

15. Test de conocimientos

1. ¿Qué significa SQL DQL?

a) Data Definition Language
b) Data Manipulation Language
c) Data Query Language
d) Data Control Language

2. ¿Cuál de las siguientes NO es una función agregada en SQL?

a) AVG()
b) MAX()
c) WHERE()
d) COUNT()

3. ¿Para qué se utiliza la cláusula `ORDER BY`?

a) Para filtrar filas en una consulta.
b) Para ordenar los resultados de una consulta.
c) Para agrupar filas en una consulta.
d) Para calcular el promedio de una columna.

4. ¿Qué operador lógico se utiliza para combinar dos condiciones donde ambas deben ser verdaderas?

a) OR
b) AND
c) NOT
d) BETWEEN

5. ¿Qué cláusula se utiliza para filtrar grupos de filas después de aplicar `GROUP BY`?

a) WHERE
b) HAVING
c) ORDER BY
d) LIMIT

6. ¿Qué operador se utiliza para buscar patrones en cadenas de texto?

a) BETWEEN
b) IN
c) LIKE
d) EXISTS

7. ¿Qué función devuelve la fecha y hora actuales del sistema?

a) DATE()
b) NOW()
c) TODAY()
d) TIME()

8. ¿Qué operador aritmético se utiliza para obtener el resto de una división entera?

a) /

b) *

c) %

d) -

9. ¿Qué tipo de consulta se ejecuta dentro de otra consulta?

a) Consulta externa

b) Subconsulta

c) Consulta principal

d) Consulta anidad