UF2176 - DEFINICIÓN Y MANIPULACIÓN DE DATOS

Unidad 2: El lenguaje de manipulación de la base de datos

By: Sergi Faura Alsina

SOC Servei d'Ocupació de Catalunya · Generalitat de Catalunya · Ministerio de Educación, Formación Profesional y Deportes · Ministerio de Trabajo y Economía Social · SEPE

Índice

2. El lenguaje de manipulación de la base de datos

El lenguaje de manipulación de bases de datos es una herramienta clave en la administración y el uso eficiente de sistemas de bases de datos relacionales. Su objetivo principal es permitir la interacción directa con los datos, tanto en términos de su definición, manipulación y control, como en la optimización de las consultas realizadas sobre ellos.

Este lenguaje está compuesto por varias subcategorías que permiten gestionar las distintas facetas de una base de datos. En primer lugar, el **Lenguaje de Definición de Datos (DDL)** se encarga de la creación y modificación de la estructura de la base de datos, incluyendo tablas, vistas e índices. Además, permite establecer restricciones de integridad, que garantizan la correcta relación entre los datos y su consistencia.

Por otro lado, el **Lenguaje de Manipulación de Datos (DML)** está orientado a trabajar directamente con los datos almacenados. A través de DML, es posible realizar consultas complejas para seleccionar, insertar, modificar y eliminar datos. Esto incluye operaciones de agregación, subconsultas, y la posibilidad de combinar datos de diferentes tablas mediante uniones e intersecciones.

Las **cláusulas para la agrupación y ordenación de consultas** también forman parte del lenguaje de manipulación de bases de datos. Estas permiten organizar los resultados obtenidos de manera que sea más fácil interpretarlos y utilizarlos en el contexto adecuado. Junto con estas, las capacidades aritméticas, lógicas y de comparación amplían las posibilidades del lenguaje, permitiendo que las consultas sean más precisas y ajustadas a las necesidades específicas.

Otra parte fundamental de este lenguaje son las **funciones agregadas** y el tratamiento de valores nulos, que ayudan a sintetizar la información y gestionar datos incompletos sin perder precisión. Además, se incluyen herramientas para la construcción de consultas anidadas, operaciones de unión, intersección y diferencia entre tablas, y otras cláusulas avanzadas que permiten una mayor flexibilidad.

En el ámbito de la seguridad y el control de transacciones, el **Lenguaje de Control de Datos (DCL)** garantiza la integridad de los datos en entornos transaccionales, permitiendo gestionar las propiedades de las transacciones como atomicidad, consistencia, aislamiento y durabilidad. También se encarga de definir y controlar los privilegios de acceso, asegurando que los datos solo estén disponibles para usuarios autorizados.

Por último, la optimización de consultas es un elemento esencial en el manejo eficiente de bases de datos. La capacidad de procesar consultas de manera rápida y eficiente depende de los mecanismos de **optimización de consultas**, que se basan en reglas, costos y otros factores que determinan el rendimiento del sistema.

En este documento, nos sumergiremos en cada uno de estos componentes del lenguaje de manipulación de bases de datos, proporcionando una comprensión profunda y práctica de cómo se aplican en un entorno real para maximizar la eficiencia y seguridad de los datos.

2.1. El lenguaje de definición de datos (DDL):

El Lenguaje de Definición de Datos (DDL) es una parte esencial del SQL que se utiliza para definir, modificar y eliminar la estructura de los objetos en una base de datos. Estos objetos incluyen tablas, vistas, índices y otras entidades que son fundamentales para la organización y el acceso eficiente a los datos. A continuación, desarrollaremos cada subpunto del DDL.

2.1.1. Tipos de datos del lenguaje.

Los tipos de datos son la base sobre la que se construyen las tablas, ya que definen qué tipo de información se puede almacenar en cada columna. Entre los principales tipos de datos se encuentran:

- **Numéricos**: Estos tipos de datos permiten almacenar valores numéricos. Los más comunes incluyen:
    - **INT**: Para números enteros.
    - **DECIMAL y NUMERIC**: Para números con decimales y precisión fija.
    - **FLOAT y REAL**: Para números con decimales de precisión variable.
- **Cadenas de texto**: Permiten almacenar secuencias de caracteres.
    - **VARCHAR**: Cadenas de longitud variable.
    - **CHAR**: Cadenas de longitud fija.
    - **TEXT**: Cadenas muy largas.
- **Fecha y hora**: Para almacenar información temporal.
    - **DATE**: Para fechas.
    - **TIME**: Para horas.
    - **DATETIME y TIMESTAMP**: Para fechas y horas combinadas.
- **Booleanos**: Tipo de dato lógico con dos posibles valores: `TRUE` o `FALSE`.

2.1.2. Creación, modificación y borrado de tablas.

Las tablas son los objetos fundamentales en las bases de datos relacionales. Definen la estructura de los datos almacenados y se componen de columnas, cada una con un tipo de dato y restricciones.

- **Crear tablas**: El comando `CREATE TABLE` define una nueva tabla especificando sus columnas, tipos de datos y restricciones (como claves primarias o foráneas).
- **Modificar tablas**: Con `ALTER TABLE`, se pueden realizar cambios en la estructura de una tabla ya existente, como añadir, eliminar o modificar columnas.
- **Borrar tablas**: El comando `DROP TABLE` elimina permanentemente una tabla, mientras que `TRUNCATE` borra todos sus registros sin alterar su estructura.

2.1.3. Creación, modificación y borrado de vistas.

Las vistas son tablas virtuales que presentan datos de una o varias tablas a través de una consulta predefinida. No almacenan datos, sino que ofrecen una forma de abstraer y simplificar el acceso a la información.

- **Crear vistas**: Con `CREATE VIEW`, se puede definir una vista basada en una consulta. Esto puede ser útil para simplificar consultas complejas o controlar el acceso a ciertos datos.
- **Modificar vistas**: Algunas bases de datos permiten modificar una vista existente usando `CREATE OR REPLACE VIEW`.
- **Borrar vistas**: El comando `DROP VIEW` elimina una vista sin afectar los datos de las tablas subyacentes.

2.1.4. Creación, modificación y borrado de índices.

Los índices mejoran la eficiencia en la búsqueda y recuperación de datos. Actúan como estructuras adicionales que permiten acelerar el acceso a los registros.

- **Crear índices**: Se utiliza el comando `CREATE INDEX` para crear un índice basado en una o más columnas de una tabla, mejorando así la velocidad de las consultas.
- **Modificar índices**: Para modificar un índice, es necesario eliminarlo y volver a crearlo con la nueva configuración.
- **Borrar índices**: Con `DROP INDEX`, se puede eliminar un índice cuando ya no es necesario o si se desea optimizar la base de datos de otra manera.

2.1.5. Especificación de restricciones de integridad.

Las restricciones de integridad son reglas aplicadas a las columnas de una tabla para asegurar que los datos cumplan con ciertas condiciones, garantizando así su exactitud y coherencia.

- **Clave primaria (PRIMARY KEY)**: Asegura que cada fila de una tabla tenga un valor único y no nulo.
- **Clave foránea (FOREIGN KEY)**: Define relaciones entre tablas, asegurando que los valores en una columna correspondan a valores existentes en otra tabla.
- **Restricción de unicidad (UNIQUE)**: Garantiza que los valores en una columna o grupo de columnas sean únicos.
- **Restricción de no nulo (NOT NULL)**: Impide que una columna contenga valores nulos.
- **Restricción de comprobación (CHECK)**: Define una condición que los datos deben cumplir para ser válidos.
- **Restricciones con acciones (ON DELETE/ON UPDATE)**: Establecen qué sucede cuando una fila relacionada es eliminada o actualizada.

Código de ejemplo completo 2.1

```sql
-- Creación de una tabla con diferentes tipos de datos y restricciones
CREATE TABLE empleados (
    id INT PRIMARY KEY,             -- Clave primaria
    nombre VARCHAR(100) NOT NULL,    -- Nombre obligatorio
    fecha_nacimiento DATE,           -- Fecha de nacimiento
    salario DECIMAL(10, 2) CHECK (salario > 0),  -- Salario con restricción de valor positivo
    departamento_id INT,             -- Relación con la tabla departamentos
    FOREIGN KEY (departamento_id) REFERENCES departamentos(id)  -- Clave foránea
);

-- Modificación de la tabla para agregar una nueva columna
ALTER TABLE empleados ADD direccion VARCHAR(255);

-- Borrar la tabla empleados
DROP TABLE empleados;

-- Creación de una vista para mostrar empleados con salario mayor a 3000
CREATE VIEW vista_empleados AS
SELECT nombre, salario
FROM empleados
WHERE salario > 3000;

-- Modificación de la vista para incluir la columna dirección
CREATE OR REPLACE VIEW vista_empleados AS
SELECT nombre, salario, direccion
FROM empleados
WHERE salario > 3000;

-- Eliminación de la vista
DROP VIEW vista_empleados;

-- Creación de un índice en la columna nombre
CREATE INDEX idx_nombre ON empleados(nombre);

-- Borrar el índice
DROP INDEX idx_nombre;

-- Creación de la tabla departamentos con una clave primaria
CREATE TABLE departamentos (
    id INT PRIMARY KEY,
    nombre VARCHAR(50) UNIQUE   -- Nombre único para cada departamento
);

-- Relación entre empleados y departamentos con claves foráneas y acciones ON DELETE/ON UPDATE
CREATE TABLE empleados (
    id INT PRIMARY KEY,
    nombre VARCHAR(100),
    departamento_id INT,
    FOREIGN KEY (departamento_id) REFERENCES departamentos(id)
    ON DELETE CASCADE  -- Si se borra un departamento, también se borran los empleados asociados
    ON UPDATE CASCADE  -- Si se actualiza el id de un departamento, se actualiza en empleados
);
```

2.2. El lenguaje de manipulación de datos (DML):

El Lenguaje de Manipulación de Datos (DML) se utiliza para realizar operaciones en los datos almacenados en una base de datos. Las operaciones más comunes incluyen la selección de datos, la inserción de nuevos registros, la modificación de datos existentes y la eliminación de registros. El DML no altera la estructura de las tablas ni otros objetos de la base de datos, sino que se enfoca exclusivamente en los datos.

2.2.1. Construcción de consultas de selección: Agregación, Subconsultas, Unión, Intersección, Diferencia.

Las consultas de selección se utilizan para recuperar datos de una o varias tablas. Estas consultas pueden incluir operaciones complejas como agregaciones, subconsultas, uniones, intersecciones y diferencias, lo que permite obtener datos procesados y filtrados según diversas condiciones.

1. **Agregación**: Las funciones de agregación permiten realizar cálculos sobre un conjunto de filas y devolver un único valor. Ejemplos comunes de funciones de agregación son:
    - **COUNT**: Cuenta el número de filas que coinciden con una condición.
    - **SUM**: Calcula la suma de los valores en una columna.
    - **AVG**: Devuelve el valor promedio de una columna.
    - **MAX** y **MIN**: Devuelven el valor máximo y mínimo de una columna, respectivamente.
2. Las funciones de agregación pueden combinarse con la cláusula `GROUP BY` para agrupar resultados según una o más columnas.
3. **Subconsultas**: Una subconsulta es una consulta anidada dentro de otra consulta. Las subconsultas pueden devolver un único valor o un conjunto de resultados, y se utilizan comúnmente para filtrar resultados o comparar valores con otras tablas o consultas.
4. **Unión (UNION)**: La cláusula `UNION` se utiliza para combinar los resultados de dos o más consultas `SELECT`. Las consultas unidas deben tener el mismo número de columnas y tipos de datos compatibles. El `UNION` elimina los duplicados de los resultados, mientras que `UNION ALL` incluye duplicados.
5. **Intersección (INTERSECT)**: La intersección devuelve las filas que son comunes a dos conjuntos de resultados. Al igual que con la `UNION`, las consultas deben tener el mismo número de columnas y tipos de datos.
6. **Diferencia (EXCEPT)**: La diferencia de conjuntos, mediante `EXCEPT`, devuelve las filas que están en el primer conjunto de resultados, pero no en el segundo.

2.2.2. Construcción de consultas de inserción.

Las consultas de inserción permiten agregar nuevos registros a una tabla. El comando `INSERT INTO` se utiliza para este propósito, especificando los valores que se insertarán en cada columna. Existen dos formas principales de usar `INSERT`:

1. **Insertar una fila completa**: Se deben especificar todos los valores de las columnas de la tabla, respetando el orden en el que están definidas.

2. **Insertar en columnas específicas**: Solo se proporcionan valores para ciertas columnas, y las demás toman valores predeterminados o nulos.

2.2.3. Construcción de consultas de modificación.

Las consultas de modificación permiten actualizar los datos existentes en una tabla. El comando `UPDATE` se usa para modificar una o más columnas de las filas que cumplen una condición especificada con `WHERE`.

- Si no se especifica la cláusula `WHERE`, todas las filas de la tabla serán actualizadas.
- Es posible actualizar una o varias columnas al mismo tiempo.

2.2.4. Construcción de consultas de borrado.

Las consultas de borrado eliminan registros de una tabla utilizando el comando `DELETE`. Al igual que con `UPDATE`, la cláusula `WHERE` define qué filas serán eliminadas.

- Si no se especifica `WHERE`, todas las filas de la tabla serán eliminadas.
- Las eliminaciones pueden estar restringidas por claves foráneas que protegen la integridad referencial.

Código de ejemplo completo 2.2

```sql
-- Creación de la tabla empleados
CREATE TABLE empleados (
    id INT PRIMARY KEY,
    nombre VARCHAR(100),
    salario DECIMAL(10, 2),
    departamento_id INT
);

-- 2.2.1. Consultas de selección con agregación, subconsultas, unión, intersección y diferencia

-- Agregación: promedio de salarios y total de empleados por departamento
SELECT departamento_id, AVG(salario) AS salario_promedio, COUNT(*) AS total_empleados
FROM empleados
GROUP BY departamento_id;

-- Subconsulta: selección de empleados con salario mayor al promedio general
SELECT nombre, salario
FROM empleados
WHERE salario > (SELECT AVG(salario) FROM empleados);

-- Unión: combinar empleados de dos departamentos diferentes
SELECT nombre FROM empleados WHERE departamento_id = 1
UNION
SELECT nombre FROM empleados WHERE departamento_id = 2;

-- Intersección: empleados con salario mayor a 3000 en los departamentos 1 y 2
SELECT nombre FROM empleados WHERE departamento_id = 1 AND salario > 3000
INTERSECT
SELECT nombre FROM empleados WHERE departamento_id = 2 AND salario > 3000;

-- Diferencia: empleados del departamento 1 que no están en el departamento 2
SELECT nombre FROM empleados WHERE departamento_id = 1
EXCEPT
SELECT nombre FROM empleados WHERE departamento_id = 2;

-- 2.2.2. Consulta de inserción

-- Insertar un nuevo empleado en la tabla
INSERT INTO empleados (id, nombre, salario, departamento_id)
VALUES (1, 'Juan Pérez', 3500.00, 1);

-- Insertar solo en columnas específicas
INSERT INTO empleados (id, nombre)
VALUES (2, 'Ana Gómez');

-- 2.2.3. Consulta de modificación

-- Actualizar el salario de un empleado
UPDATE empleados
SET salario = 4000.00
WHERE id = 1;

-- Actualizar el salario y el departamento de un empleado
UPDATE empleados
SET salario = 4500.00, departamento_id = 2
WHERE id = 2;

-- 2.2.4. Consulta de borrado

-- Eliminar un empleado específico
DELETE FROM empleados
WHERE id = 1;

-- Eliminar todos los empleados del departamento 2
DELETE FROM empleados
WHERE departamento_id = 2;
```

2.3. Cláusulas del lenguaje para la agrupación y ordenación de las consultas.

Las cláusulas de agrupación y ordenación son elementos clave en SQL que permiten organizar y presentar los resultados de una consulta de manera estructurada y fácil de interpretar. Estas cláusulas hacen que las consultas SQL sean más potentes y flexibles, permitiendo a los usuarios realizar análisis más complejos sobre los datos.

2.3.1. Cláusula `GROUP BY`

La cláusula `GROUP BY` permite agrupar filas que tienen los mismos valores en una o más columnas y aplicar funciones de agregación, como `COUNT`, `SUM`, `AVG`, `MAX`, y `MIN`. El `GROUP BY` es útil para resumir información y realizar cálculos sobre grupos de datos.

1. **Estructura básica**:
    - La cláusula `GROUP BY` se utiliza después de la cláusula `WHERE` y antes de `ORDER BY`.
    - Las columnas que aparecen en `GROUP BY` deben coincidir con las que se mencionan en la selección, excepto cuando se utilizan funciones de agregación.
2. **Función en combinación con `HAVING`**:
    - La cláusula `HAVING` permite filtrar grupos que se han creado con `GROUP BY`. A diferencia de `WHERE`, que filtra filas antes de la agrupación, `HAVING` filtra después de que los grupos han sido formados.
3. **Ejemplos comunes**:
    - Contar el número de empleados en cada departamento.
    - Sumar los salarios por departamento.
    - Calcular el promedio de ventas de cada categoría de producto.

2.3.2. Cláusula `ORDER BY`

La cláusula `ORDER BY` se utiliza para ordenar los resultados de una consulta en un orden específico, ya sea ascendente (`ASC`, por defecto) o descendente (`DESC`). La ordenación puede basarse en una o más columnas.

1. **Estructura básica**:
    - `ORDER BY` se coloca al final de la consulta.
    - Se puede ordenar por una o varias columnas, y es posible especificar diferentes órdenes (ascendente o descendente) para cada una de ellas.

2. **Combinación con otras cláusulas**:
    - `ORDER BY` puede combinarse con `GROUP BY` para ordenar los resultados agrupados.
    - Es posible ordenar por columnas que no están incluidas en la lista de selección, siempre y cuando sean parte de las tablas consultadas.
3. **Ejemplos comunes**:
    - Ordenar una lista de empleados por salario de mayor a menor.
    - Ordenar productos por fecha de creación, primero los más recientes.
    - Ordenar las ventas por mes en orden ascendente para analizar tendencias.

2.3.3. Uso conjunto de `GROUP BY` y `ORDER BY`

En muchos casos, se utilizan las cláusulas `GROUP BY` y `ORDER BY` juntas para primero agrupar los datos y luego ordenar los resultados en función de una o más columnas. Esta combinación es útil para resumir datos y luego presentarlos en un orden lógico.

1. **Agrupación seguida de ordenación**:
    - Primero se agrupan los datos según las columnas deseadas.
    - Posteriormente, los resultados se ordenan según una columna agregada o cualquier otra columna relevante.
2. **Ejemplos comunes**:
    - Agrupar las ventas por categoría de producto y luego ordenar los resultados por la suma total de ventas en cada categoría.
    - Agrupar a los empleados por departamento y ordenar los departamentos según el número de empleados.

Código de Ejemplo Completo 2.3.

```sql
-- 2.3.1. Uso de la cláusula GROUP BY para agrupar los empleados por departamento
-- y calcular el salario promedio en cada uno de ellos
SELECT departamento_id, AVG(salario) AS salario_promedio
FROM empleados
GROUP BY departamento_id;

-- 2.3.1. Uso de la cláusula GROUP BY con HAVING para filtrar los departamentos
-- con un salario promedio mayor a 4000
SELECT departamento_id, AVG(salario) AS salario_promedio
FROM empleados
GROUP BY departamento_id
HAVING AVG(salario) > 4000;

-- 2.3.2. Uso de la cláusula ORDER BY para ordenar los empleados por fecha de contratación de más reciente a más antigua
SELECT nombre, fecha_contratacion
FROM empleados
ORDER BY fecha_contratacion DESC;

-- 2.3.2. Uso de la cláusula ORDER BY para ordenar los empleados por salario ascendente
SELECT nombre, salario
FROM empleados
ORDER BY salario ASC;

-- 2.3.3. Uso conjunto de GROUP BY y ORDER BY para agrupar a los empleados por departamento
-- y luego ordenar los resultados por el salario promedio en orden descendente
SELECT departamento_id, AVG(salario) AS salario_promedio
FROM empleados
GROUP BY departamento_id
ORDER BY salario_promedio DESC;

-- 2.3.3. Uso conjunto de GROUP BY y ORDER BY para contar los empleados en cada departamento
-- y ordenar los resultados por el número de empleados en orden descendente
SELECT departamento_id, COUNT(*) AS numero_empleados
FROM empleados
GROUP BY departamento_id
ORDER BY numero_empleados DESC;

```

2.4. Capacidades aritméticas, lógicas y de comparación del lenguaje.

Las capacidades aritméticas, lógicas y de comparación en SQL son fundamentales para construir consultas que permitan filtrar, calcular y analizar datos en una base de datos. Estas capacidades permiten a los usuarios realizar operaciones matemáticas sobre los valores almacenados, hacer comparaciones entre los registros y combinar condiciones para realizar consultas más complejas y precisas.

2.4.1. Operadores aritméticos

Los operadores aritméticos permiten realizar cálculos matemáticos directamente sobre los valores de las columnas de una tabla. Estos operadores son útiles cuando se necesita manipular datos numéricos en una consulta SQL.

1. **Suma (+)**:
    - Permite sumar valores numéricos de una o más columnas.
2. **Resta (-)**:
    - Permite restar valores numéricos.

3. **Multiplicación (*)**:
    - Realiza la multiplicación de los valores numéricos.
4. **División (/)**:
    - Permite dividir un valor por otro.
5. **Módulo (%)**:
    - Devuelve el resto de una división entre dos valores.

Estos operadores pueden utilizarse en la cláusula `SELECT` para generar nuevas columnas con valores calculados o en la cláusula `WHERE` para definir condiciones basadas en cálculos.

2.4.2. Operadores lógicos

Los operadores lógicos permiten combinar condiciones en una consulta SQL. Los operadores lógicos son utilizados comúnmente en la cláusula `WHERE` para establecer múltiples condiciones que los datos deben cumplir.

1. **AND**:
    - Se utiliza para combinar dos o más condiciones y devuelve resultados solo si todas las condiciones son verdaderas.
2. **OR**:
    - Devuelve resultados si al menos una de las condiciones es verdadera.
3. **NOT**:
    - Se utiliza para invertir el valor de una condición. Devuelve verdadero si la condición es falsa.
4. **BETWEEN**:
    - Se utiliza para filtrar valores que se encuentren dentro de un rango especificado.
5. **IN**:
    - Permite especificar una lista de valores y filtrar los registros que coincidan con alguno de ellos.
6. **IS NULL**:
    - Se utiliza para comprobar si un valor es nulo.

2.4.3. Operadores de comparación

Los operadores de comparación permiten comparar valores entre columnas, o entre una columna y un valor específico, para definir condiciones que los registros deben cumplir.

1. **Igual (=)**:
    - Devuelve resultados si el valor de una columna es igual al valor especificado.
2. **Distinto (<> o !=)**:
    - Devuelve resultados si el valor de una columna no es igual al valor especificado.
3. **Mayor que (>)**:
    - Devuelve resultados si el valor de una columna es mayor que el valor especificado.
4. **Menor que (<)**:
    - Devuelve resultados si el valor de una columna es menor que el valor especificado.
5. **Mayor o igual que (>=)**:
    - Devuelve resultados si el valor de una columna es mayor o igual que el valor especificado.
6. **Menor o igual que (<=)**:
    - Devuelve resultados si el valor de una columna es menor o igual que el valor especificado.
7. **LIKE**:
    - Se utiliza para buscar un patrón en una columna de texto. Admite caracteres comodín como % para representar cualquier número de caracteres o _ para representar un solo carácter.

Código de Ejemplo Completo 2.4.

```sql
-- 2.4.1. Operaciones aritméticas
-- Calcular un aumento del 10% en el salario de todos los empleados
SELECT nombre, salario, salario * 1.10 AS salario_con_aumento
FROM empleados;

-- 2.4.1. Operaciones aritméticas
-- Calcular la edad que tendrán los empleados dentro de 5 años
SELECT nombre, edad, edad + 5 AS edad_en_5_anos
FROM empleados;

-- 2.4.2. Operadores lógicos
-- Seleccionar empleados cuyo salario esté entre 3000 y 5000 y que pertenezcan al departamento 1
SELECT nombre, salario
FROM empleados
WHERE salario BETWEEN 3000 AND 5000
AND departamento_id = 1;

-- 2.4.2. Operadores lógicos
-- Seleccionar empleados que pertenezcan al departamento 1 o al departamento 2
SELECT nombre, departamento_id
FROM empleados
WHERE departamento_id = 1 OR departamento_id = 2;

-- 2.4.2. Uso de NOT
-- Seleccionar empleados cuyo salario no sea mayor a 4000
SELECT nombre, salario
FROM empleados
WHERE NOT salario > 4000;

-- 2.4.3. Operadores de comparación
-- Seleccionar empleados con salario mayor a 3500 y edad menor a 30
SELECT nombre, salario, edad
FROM empleados
WHERE salario > 3500 AND edad < 30;

-- 2.4.3. Operadores de comparación
-- Seleccionar empleados cuyo nombre empiece con la letra 'A'
SELECT nombre
FROM empleados
WHERE nombre LIKE 'A%';

-- 2.4.3. Uso de IS NULL
-- Seleccionar empleados que no tienen departamento asignado (en caso de que hubiera algún valor nulo en departamento_id)
SELECT nombre
FROM empleados
WHERE departamento_id IS NULL;
```

2.5. Funciones agregadas del lenguaje.

Las funciones agregadas en SQL son utilizadas para realizar cálculos y operaciones sobre un conjunto de filas y devolver un único valor como resultado. Estas funciones se aplican comúnmente junto con las cláusulas `GROUP BY` y `HAVING` para agrupar y filtrar datos. Las funciones agregadas permiten resumir información, lo que resulta útil en el análisis de datos, la elaboración de informes y la toma de decisiones basada en datos. A continuación, se desarrollan las funciones agregadas más importantes.

2.5.1. `COUNT()`: Contar el número de filas

La función `COUNT()` devuelve el número total de filas en un conjunto de resultados. Puede contar todas las filas o solo aquellas que no contienen valores nulos en una columna específica.

1. **Sintaxis básica**:
    - `COUNT(*)`: Cuenta todas las filas, incluyendo las que contienen valores nulos.
    - `COUNT(columna)`: Cuenta solo las filas donde la columna especificada no es nula.
2. **Aplicaciones comunes**:
    - Contar el número total de registros en una tabla.
    - Contar el número de registros en función de una condición.

2.5.2. `SUM()`: Sumar los valores de una columna

La función `SUM()` se utiliza para sumar los valores numéricos de una columna. Solo se puede aplicar a columnas que contienen datos numéricos.

1. **Sintaxis básica**:
    - `SUM(columna)`: Devuelve la suma de todos los valores no nulos de la columna especificada.
2. **Aplicaciones comunes**:
    - Calcular el total de ventas de una empresa.
    - Sumar los salarios de los empleados en un departamento específico.

2.5.3. `AVG()`: Calcular el promedio

La función `AVG()` devuelve el promedio de los valores de una columna. Al igual que `SUM()`, esta función solo se aplica a columnas con datos numéricos.

1. **Sintaxis básica**:
    - `AVG(columna)`: Devuelve el promedio de los valores no nulos de la columna especificada.
2. **Aplicaciones comunes**:
    - Calcular el salario promedio de los empleados.
    - Determinar el promedio de ventas en un período de tiempo.

2.5.4. `MAX()`: Devolver el valor máximo

La función `MAX()` devuelve el valor más alto de una columna. Se puede aplicar tanto a columnas numéricas como a columnas que contienen fechas o textos.

1. **Sintaxis básica**:
    - `MAX(columna)`: Devuelve el valor máximo de la columna especificada.
2. **Aplicaciones comunes**:
    - Determinar el salario más alto en una empresa.
    - Encontrar la fecha más reciente de un conjunto de transacciones.

2.5.5. `MIN()`: Devolver el valor mínimo

La función `MIN()` es el opuesto de `MAX()`. Devuelve el valor más bajo de una columna y se puede aplicar a columnas numéricas, de fecha o texto.

1. **Sintaxis básica**:
    - `MIN(columna)`: Devuelve el valor mínimo de la columna especificada.
2. **Aplicaciones comunes**:
    - Encontrar el salario más bajo entre los empleados.
    - Determinar la fecha más antigua de un conjunto de transacciones.

2.5.6. Uso combinado de funciones agregadas con `GROUP BY`

Las funciones agregadas suelen utilizarse en combinación con la cláusula `GROUP BY`, que agrupa las filas que tienen los mismos valores en una o más columnas. Después de agrupar los datos, las funciones agregadas se aplican a cada grupo de filas.

1. **`GROUP BY` con `COUNT()`, `SUM()`, `AVG()`, `MAX()`, y `MIN()`**:
    - Agrupar por una columna (por ejemplo, departamento) y aplicar funciones agregadas para realizar cálculos sobre cada grupo de datos.

2.5.7. Uso de `HAVING` con funciones agregadas

La cláusula `HAVING` se utiliza para filtrar los resultados después de que se han aplicado las funciones agregadas y `GROUP BY`. A diferencia de `WHERE`, que filtra filas individuales,

`HAVING` se usa para filtrar grupos de resultados basados en condiciones aplicadas a funciones agregadas.

1. **Sintaxis básica**:
   - `SELECT columna, FUNCION_AGREGADA FROM tabla GROUP BY columna HAVING condición`.
2. **Aplicaciones comunes**:
   - Filtrar grupos de datos que cumplan con ciertas condiciones, como mostrar solo los departamentos con un promedio salarial superior a 4000.

Código de Ejemplo Completo 2.5.

```sql
-- 2.5.1. Contar el número de empleados en la tabla
SELECT COUNT(*) AS total_empleados
FROM empleados;

-- 2.5.2. Sumar los salarios de todos los empleados
SELECT SUM(salario) AS total_salarios
FROM empleados;

-- 2.5.3. Calcular el salario promedio de los empleados
SELECT AVG(salario) AS salario_promedio
FROM empleados;

-- 2.5.4. Encontrar el salario máximo de los empleados
SELECT MAX(salario) AS salario_maximo
FROM empleados;

-- 2.5.5. Encontrar el salario mínimo de los empleados
SELECT MIN(salario) AS salario_minimo
FROM empleados;

-- 2.5.6. Uso de GROUP BY para agrupar por departamento y calcular el salario promedio por departamento
SELECT departamento_id, AVG(salario) AS salario_promedio
FROM empleados
GROUP BY departamento_id;

-- 2.5.7. Uso de HAVING para filtrar los departamentos con un salario promedio mayor a 4000
SELECT departamento_id, AVG(salario) AS salario_promedio
FROM empleados
GROUP BY departamento_id
HAVING AVG(salario) > 4000;

-- 2.5.7. Uso de GROUP BY y funciones agregadas para contar empleados y calcular salarios totales por departamento
SELECT departamento_id, COUNT(*) AS numero_empleados, SUM(salario) AS total_salarios
FROM empleados
GROUP BY departamento_id;
```

2.6. Tratamiento de valores nulos.

En SQL, un valor nulo (`NULL`) representa la ausencia de un valor o un valor desconocido. Los valores nulos son comunes en bases de datos y es esencial aprender a manejarlos adecuadamente, ya que no se comportan como otros valores en operaciones o

comparaciones. El tratamiento de valores nulos incluye la capacidad de detectarlos, reemplazarlos y evitar problemas en consultas y cálculos.

2.6.1. ¿Qué es un valor nulo?

Un valor nulo indica que el valor de un campo es desconocido o no se ha proporcionado. No es lo mismo que un valor vacío, cero o una cadena vacía. En SQL, las operaciones con valores nulos suelen resultar en otro valor nulo, lo que requiere un manejo específico para obtener resultados adecuados.

- **No es lo mismo que cero**: Un valor nulo en una columna numérica no es igual a 0.
- **No es lo mismo que una cadena vacía**: En una columna de texto, un valor nulo no es igual a una cadena vacía (`''`).

2.6.2. Uso de `IS NULL` y `IS NOT NULL`

Para detectar valores nulos en una columna, se utilizan las cláusulas `IS NULL` y `IS NOT NULL`. Estas cláusulas permiten filtrar registros que contienen o no contienen valores nulos en columnas específicas.

- **`IS NULL`**: Se utiliza para seleccionar filas donde una columna tiene un valor nulo.
- **`IS NOT NULL`**: Selecciona filas donde una columna no contiene valores nulos.

2.6.3. Operaciones aritméticas y lógicas con valores nulos

Las operaciones aritméticas o lógicas que involucren un valor nulo devuelven nulo. Por ejemplo, sumar cualquier número con un valor nulo devolverá un valor nulo.

- Ejemplo: `SELECT 100 + NULL;` devolverá `NULL`.
- Este comportamiento se aplica también a las comparaciones. Comparar un valor nulo con cualquier otro valor (incluso otro nulo) no devuelve verdadero ni falso, sino nulo.

2.6.4. Uso de la función `COALESCE()`

La función `COALESCE()` es muy útil para manejar valores nulos. Esta función devuelve el primer valor no nulo de una lista de expresiones. Si todas las expresiones son nulas, devuelve `NULL`.

- **Sintaxis básica**:COALESCE(expresion1, expresion2, ..., expresionN)

La función evalúa cada expresión en orden y devuelve la primera que no sea nula.

Ejemplo: `COALESCE(NULL, NULL, 'Valor Alternativo')` devolverá `'Valor Alternativo'`

2.6.5. Uso de la función `IFNULL()` o `NVL()`

En algunas bases de datos, como MySQL y Oracle, existen funciones específicas como `IFNULL()` o `NVL()` que se comportan de manera similar a `COALESCE()`. Estas funciones permiten especificar un valor predeterminado cuando se encuentra un valor nulo en una columna.

- **`IFNULL(expresion, valor)` (MySQL)**: Devuelve el valor de la expresión si no es nulo, o el valor alternativo si la expresión es nula.
- **`NVL(expresion, valor)` (Oracle)**: Funciona de manera similar a `IFNULL()`.

2.6.6. Funciones agregadas y valores nulos

Las funciones agregadas como `COUNT()`, `SUM()`, `AVG()`, `MAX()`, y `MIN()` manejan valores nulos de manera particular:

- **`COUNT(*)`**: Cuenta todas las filas, incluyendo las que tienen valores nulos.
- **`COUNT(columna)`**: Cuenta solo las filas donde la columna no es nula.
- **`SUM(columna)` y `AVG(columna)`**: Ignoran los valores nulos en los cálculos.
- **`MAX(columna)` y `MIN(columna)`**: Ignoran los valores nulos al buscar el valor máximo o mínimo.

2.6.7. Reemplazo de valores nulos al mostrar resultados

En algunas situaciones, es necesario reemplazar valores nulos con un valor predeterminado al mostrar los resultados de una consulta. Para ello, se puede utilizar la función `COALESCE()` o, en bases de datos específicas, `IFNULL()` o `NVL()`.

- **Reemplazo de nulos en consultas**: Es común utilizar estas funciones para mostrar un valor alternativo, como 0 o una cadena vacía, en lugar de un valor nulo en los resultados.

Código de Ejemplo Completo 2.6.

```sql
-- 2.6.2. Uso de IS NULL y IS NOT NULL
-- Seleccionar empleados cuyo salario sea nulo
SELECT nombre
FROM empleados
WHERE salario IS NULL;

-- Seleccionar empleados cuyo departamento no sea nulo
SELECT nombre, departamento_id
FROM empleados
WHERE departamento_id IS NOT NULL;

-- 2.6.4. Uso de COALESCE para reemplazar valores nulos
-- Reemplazar el salario nulo con 0 y mostrar los resultados
SELECT nombre, COALESCE(salario, 0) AS salario
FROM empleados;

-- 2.6.5. Uso de IFNULL (en MySQL) o NVL (en Oracle) para reemplazar valores nulos
-- Reemplazar el departamento nulo con 'No asignado'
SELECT nombre, IFNULL(departamento_id, 'No asignado') AS departamento
FROM empleados;

-- 2.6.6. Funciones agregadas y valores nulos
-- Contar todos los empleados, incluidos los que tienen valores nulos
SELECT COUNT(*) AS total_empleados
FROM empleados;

-- Contar solo los empleados cuyo salario no sea nulo
SELECT COUNT(salario) AS empleados_con_salario
FROM empleados;

-- Calcular el salario promedio, ignorando los valores nulos
SELECT AVG(salario) AS salario_promedio
FROM empleados;

-- 2.6.7. Reemplazo de valores nulos en resultados de consultas
-- Mostrar los nombres de empleados y la fecha de contratación,
-- reemplazando valores nulos en la fecha con 'Fecha no disponible'
SELECT nombre, COALESCE(fecha_contratacion, 'Fecha no disponible') AS fecha_contratacion
FROM empleados;

-- 2.6.3. Operaciones aritméticas con valores nulos
-- Intentar sumar salarios, si es nulo el resultado será NULL
SELECT nombre, salario, salario + 500 AS salario_con_aumento
FROM empleados;
```

2.7. Construcción de consultas anidadas.

Las consultas anidadas, también conocidas como subconsultas, son consultas que se encuentran dentro de otra consulta. La consulta interna se ejecuta primero, y su resultado se utiliza como entrada para la consulta externa. Las consultas anidadas permiten realizar consultas complejas, donde los resultados de una consulta dependen de los resultados de otra.

Las subconsultas pueden ubicarse en varias partes de una consulta SQL, como en la cláusula `SELECT`, `FROM`, `WHERE`, o incluso en las funciones de agregación. Las consultas anidadas son especialmente útiles cuando necesitamos comparar los datos de una tabla

con los de otra, o cuando necesitamos realizar cálculos intermedios antes de ejecutar la consulta principal.

2.7.1. Tipos de consultas anidadas

Existen diferentes tipos de consultas anidadas, dependiendo de dónde se ubiquen en la consulta principal y cómo interactúan con ella:

1. **Subconsultas en la cláusula `WHERE`**: Este tipo de consulta anidada se utiliza para devolver valores que sirven como condición para la consulta principal. La subconsulta se evalúa primero, y su resultado se utiliza como criterio de comparación en la cláusula `WHERE`.
2. **Subconsultas en la cláusula `FROM`**: Aquí, la subconsulta actúa como una tabla temporal. La consulta principal utiliza los resultados de la subconsulta como si fueran una tabla. Esto es útil cuando los datos de una tabla necesitan ser preprocesados antes de que se pueda realizar la consulta principal.
3. **Subconsultas en la cláusula `SELECT`**: En este caso, las subconsultas devuelven valores individuales que se utilizan en las columnas de la consulta principal. Estas subconsultas devuelven un único valor escalar que puede ser calculado para cada fila de la consulta principal.
4. **Subconsultas con operadores de comparación**: Las subconsultas pueden combinarse con operadores de comparación como `IN`, `NOT IN`, `EXISTS`, `ALL`, y `ANY`, lo que permite realizar comparaciones entre el resultado de la subconsulta y los valores en la consulta principal.

2.7.2. Subconsultas en la cláusula `WHERE`

Una de las formas más comunes de utilizar consultas anidadas es en la cláusula `WHERE`. La subconsulta devuelve un valor o conjunto de valores que se usan como criterio para filtrar los datos en la consulta principal.

- **Uso de `IN`**: La subconsulta devuelve una lista de valores, y la consulta principal selecciona las filas cuyos valores coinciden con alguno de los resultados devueltos por la subconsulta.
- **Uso de `EXISTS`**: La subconsulta devuelve un conjunto de filas, y `EXISTS` comprueba si la subconsulta devuelve al menos una fila. Si es así, la consulta principal devuelve las filas correspondientes.

2.7.3. Subconsultas en la cláusula `FROM`

Las subconsultas en la cláusula `FROM` actúan como tablas temporales para la consulta principal. Estas subconsultas permiten realizar cálculos o agregaciones intermedias que luego se utilizan en la consulta principal. Este tipo de subconsulta es útil cuando necesitas tratar un conjunto de resultados como una tabla antes de realizar la consulta principal.

2.7.4. Subconsultas en la cláusula `SELECT`

Las subconsultas en la cláusula `SELECT` devuelven un único valor escalar que se utiliza en la consulta principal. Estas subconsultas pueden ser útiles cuando se necesita realizar un cálculo o comparación que involucra datos de otra tabla o consulta.

2.7.5. Subconsultas con operadores de comparación

Los operadores de comparación permiten utilizar subconsultas para comparar el resultado de la subconsulta con los valores de la consulta principal. Algunos operadores comunes incluyen:

- **`IN`**: Selecciona las filas que coinciden con cualquiera de los valores devueltos por la subconsulta.
- **`NOT IN`**: Selecciona las filas que no coinciden con los valores devueltos por la subconsulta.
- **`EXISTS`**: Comprueba si la subconsulta devuelve al menos una fila.
- **`ANY`** y **`ALL`**: Permiten realizar comparaciones con todos los valores devueltos por la subconsulta.

Código de Ejemplo Completo 2.7.

```sql
-- Creación de la tabla empleados
CREATE TABLE empleados (
    id INT PRIMARY KEY,
    nombre VARCHAR(100),
    salario DECIMAL(10, 2),
    departamento_id INT,
    fecha_contratacion DATE
);

-- Creación de la tabla departamentos
CREATE TABLE departamentos (
    id INT PRIMARY KEY,
    nombre_departamento VARCHAR(100)
);

-- Inserción de datos en la tabla empleados
INSERT INTO empleados (id, nombre, salario, departamento_id, fecha_contratacion)
VALUES
(1, 'Juan Pérez', 3500.00, 1, '2020-01-15'),
(2, 'Ana Gómez', 4000.00, 1, '2019-02-10'),
(3, 'Carlos Díaz', 3000.00, 2, '2021-03-20'),
(4, 'Lucía Fernández', 4500.00, 2, '2018-04-05'),
(5, 'Sofía Rodríguez', 5000.00, 3, '2022-05-30');

-- Inserción de datos en la tabla departamentos
INSERT INTO departamentos (id, nombre_departamento)
VALUES
(1, 'Recursos Humanos'),
(2, 'Tecnología'),
(3, 'Finanzas');

-- 2.7.2. Subconsulta en la cláusula WHERE: Encontrar empleados que tienen un salario mayor que el promedio salarial
SELECT nombre, salario
FROM empleados
WHERE salario > (SELECT AVG(salario) FROM empleados);

-- 2.7.3. Subconsulta en la cláusula FROM: Obtener el salario promedio por departamento
SELECT d.nombre_departamento, t.salario_promedio
FROM departamentos d
JOIN (SELECT departamento_id, AVG(salario) AS salario_promedio
      FROM empleados
      GROUP BY departamento_id) t
ON d.id = t.departamento_id;

-- 2.7.4. Subconsulta en la cláusula SELECT: Mostrar empleados junto con el salario promedio de su departamento
SELECT nombre, salario,
(SELECT AVG(salario)
 FROM empleados e
 WHERE e.departamento_id = empleados.departamento_id) AS salario_promedio_departamento
FROM empleados;

-- 2.7.5. Subconsulta con el operador EXISTS: Seleccionar empleados que trabajen en departamentos con salarios superiores a 4000
SELECT nombre
FROM empleados
WHERE EXISTS (SELECT 1
              FROM empleados e
              WHERE e.departamento_id = empleados.departamento_id
              AND e.salario > 4000);

-- 2.7.5. Subconsulta con el operador IN: Seleccionar empleados que trabajen en los mismos departamentos que Carlos Díaz
SELECT nombre
FROM empleados
WHERE departamento_id IN (SELECT departamento_id
                          FROM empleados
                          WHERE nombre = 'Carlos Díaz');
```

2.8.Unión, intersección y diferencia de consultas.

En SQL, es común necesitar combinar los resultados de varias consultas para obtener una vista completa de los datos. Para este fin, existen tres operaciones fundamentales que permiten combinar los resultados de diferentes consultas: **UNIÓN**, **INTERSECCIÓN**, y **DIFERENCIA** (a menudo implementada con `EXCEPT`). Estas operaciones son esenciales para realizar comparaciones y combinar conjuntos de datos de manera eficiente.

2.8.1. `UNION`: Unión de Consultas

La operación `UNION` se utiliza para combinar los resultados de dos o más consultas en un solo conjunto de resultados. La operación de unión elimina los duplicados de los resultados combinados, a menos que se utilice `UNION ALL`, que incluye los duplicados.

1. **Requisitos**:
    - Todas las consultas deben devolver el mismo número de columnas.
    - Las columnas correspondientes de las diferentes consultas deben tener tipos de datos compatibles.
2. **Aplicaciones comunes**:
    - Combinar resultados de tablas similares.
    - Unir datos de diferentes fuentes que tienen estructuras equivalentes.

2.8.2. `INTERSECT`: Intersección de Consultas

La operación `INTERSECT` devuelve solo las filas que son comunes a los resultados de dos consultas. Es decir, devuelve las filas que aparecen en ambas consultas.

1. **Requisitos**:
    - Las dos consultas deben devolver el mismo número de columnas.
    - Las columnas correspondientes deben tener tipos de datos compatibles.
2. **Aplicaciones comunes**:
    - Identificar registros que aparecen en ambas tablas o consultas.
    - Comparar conjuntos de datos y obtener coincidencias exactas.

2.8.3. `EXCEPT`: Diferencia de Consultas

La operación `EXCEPT` devuelve las filas que están en la primera consulta, pero no en la segunda. Es decir, muestra las filas exclusivas de la primera consulta que no tienen una correspondencia en la segunda.

1. **Requisitos**:
    - Las dos consultas deben devolver el mismo número de columnas.
    - Las columnas correspondientes deben tener tipos de datos compatibles.
2. **Aplicaciones comunes**:
    - Identificar registros presentes en una tabla, pero ausentes en otra.
    - Realizar comparaciones entre conjuntos de datos para obtener diferencias.

2.8.4. Operadores de Unión, Intersección y Diferencia

1. **UNION ALL**: A diferencia de `UNION`, `UNION ALL` no elimina los duplicados. Esta operación incluye todas las filas de las consultas, incluso si algunas aparecen más de una vez.
2. **INTERSECT**: Este operador permite combinar dos conjuntos y devolver solo los valores comunes entre ambos. Es útil cuando necesitamos ver coincidencias exactas entre dos conjuntos de datos.
3. **EXCEPT**: Con `EXCEPT`, se eliminan del conjunto de resultados aquellas filas que se encuentren en la segunda consulta. Este operador se utiliza cuando queremos identificar diferencias entre dos conjuntos de datos.

2.8.5. Uso de Cláusulas `ORDER BY` en Combinación con Estas Operaciones

Es posible ordenar los resultados de las consultas combinadas utilizando la cláusula `ORDER BY`. Sin embargo, la cláusula `ORDER BY` solo se aplica a la consulta final y no a cada consulta individual que forma parte de la combinación.

Código de Ejemplo Completo 2.8.

```sql
-- 2.8.1. Uso de UNION para combinar empleados y contratistas sin duplicados
SELECT nombre, salario, departamento_id
FROM empleados
UNION
SELECT nombre, salario, departamento_id
FROM contratistas;

-- 2.8.1. Uso de UNION ALL para combinar empleados y contratistas incluyendo duplicados
SELECT nombre, salario, departamento_id
FROM empleados
UNION ALL
SELECT nombre, salario, departamento_id
FROM contratistas;

-- 2.8.2. Uso de INTERSECT para encontrar personas que son tanto empleados como contratistas
SELECT nombre, salario, departamento_id
FROM empleados
INTERSECT
SELECT nombre, salario, departamento_id
FROM contratistas;

-- 2.8.3. Uso de EXCEPT para encontrar empleados que no son contratistas
SELECT nombre, salario, departamento_id
FROM empleados
EXCEPT
SELECT nombre, salario, departamento_id
FROM contratistas;

-- 2.8.3. Uso de EXCEPT para encontrar contratistas que no son empleados
SELECT nombre, salario, departamento_id
FROM contratistas
EXCEPT
SELECT nombre, salario, departamento_id
FROM empleados;

-- 2.8.5. Uso de ORDER BY en combinación con UNION para ordenar los resultados combinados
SELECT nombre, salario, departamento_id
FROM empleados
UNION
SELECT nombre, salario, departamento_id
FROM contratistas
ORDER BY salario DESC;
```

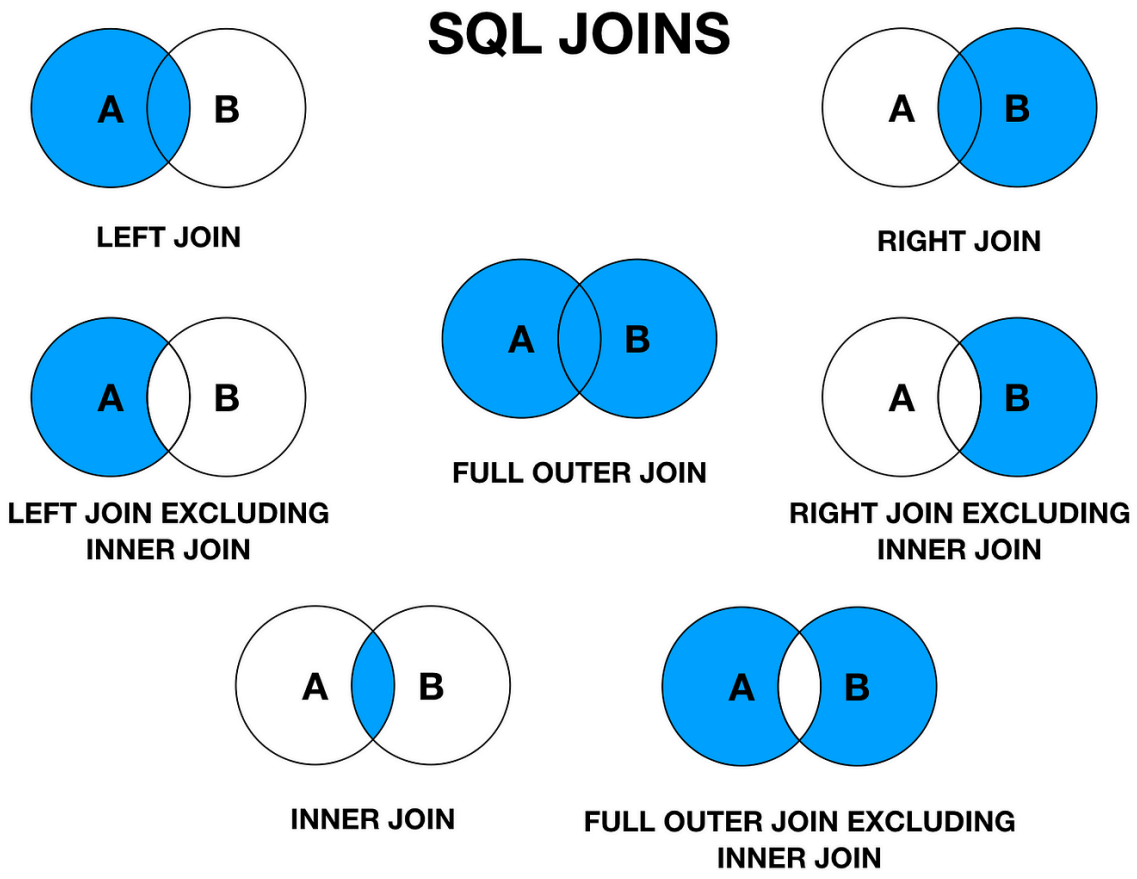
2.9. Consultas de tablas cruzadas.

En bases de datos relacionales como MySQL, las consultas de tablas cruzadas (joins) permiten combinar datos de dos o más tablas relacionadas. Los datos en las bases de datos suelen estar distribuidos en varias tablas, y las consultas cruzadas son esenciales para unir esta información y obtener una visión completa.

MySQL soporta varios tipos de combinaciones o **joins** que permiten a los usuarios cruzar datos entre tablas según las necesidades del análisis. Los joins más comunes son: **INNER JOIN**, **LEFT JOIN**, **RIGHT JOIN**, **FULL OUTER JOIN** y **CROSS JOIN**.

2.9.1. Tipos de Consultas de Tablas Cruzadas

1. **INNER JOIN**: Este tipo de combinación devuelve solo las filas que tienen coincidencias en ambas tablas. Es el join más utilizado, ya que permite relacionar tablas basadas en una condición específica, como una clave foránea.
2. **LEFT JOIN (LEFT OUTER JOIN)**: Devuelve todas las filas de la tabla de la izquierda (la primera en la consulta) y las filas coincidentes de la tabla de la derecha. Si no hay coincidencias, las columnas de la tabla de la derecha se rellenan con valores nulos.
3. **RIGHT JOIN (RIGHT OUTER JOIN)**: Es similar a `LEFT JOIN`, pero devuelve todas las filas de la tabla de la derecha y las filas coincidentes de la tabla de la izquierda. Las filas de la izquierda sin coincidencias se rellenan con valores nulos.
4. **FULL OUTER JOIN**: Devuelve todas las filas de ambas tablas, con valores nulos para las filas que no tienen coincidencias en la otra tabla. MySQL no soporta nativamente `FULL OUTER JOIN`, pero se puede emular combinando `LEFT JOIN` y `RIGHT JOIN` con una unión `UNION`.
5. **CROSS JOIN**: Realiza un producto cartesiano entre dos tablas, combinando cada fila de la primera tabla con cada fila de la segunda. Se utiliza con poca frecuencia debido a la gran cantidad de combinaciones que puede generar.
6. **SELF JOIN**: Es una combinación de una tabla consigo misma, útil cuando se requiere comparar filas dentro de la misma tabla. Se suele utilizar en estructuras jerárquicas o para comparar relaciones dentro de una misma entidad.



Código de Ejemplo Completo 2.9.

```sql
-- 1. INNER JOIN: Obtener empleados con sus departamentos correspondientes
SELECT empleados.nombre, empleados.salario, departamentos.nombre_departamento
FROM empleados
INNER JOIN departamentos ON empleados.departamento_id = departamentos.id;

-- 2. LEFT JOIN: Obtener todos los empleados, incluso aquellos sin departamento asignado
SELECT empleados.nombre, empleados.salario, departamentos.nombre_departamento
FROM empleados
LEFT JOIN departamentos ON empleados.departamento_id = departamentos.id;

-- 3. RIGHT JOIN: Obtener todos los departamentos, incluso los que no tienen empleados asignados
SELECT empleados.nombre, empleados.salario, departamentos.nombre_departamento
FROM empleados
RIGHT JOIN departamentos ON empleados.departamento_id = departamentos.id;

-- 4. FULL OUTER JOIN: Simulación en MySQL usando UNION de LEFT JOIN y RIGHT JOIN
SELECT empleados.nombre, empleados.salario, departamentos.nombre_departamento
FROM empleados
LEFT JOIN departamentos ON empleados.departamento_id = departamentos.id
UNION
SELECT empleados.nombre, empleados.salario, departamentos.nombre_departamento
FROM empleados
RIGHT JOIN departamentos ON empleados.departamento_id = departamentos.id;

-- 5. CROSS JOIN: Obtener el producto cartesiano entre empleados y departamentos
SELECT empleados.nombre, departamentos.nombre_departamento
FROM empleados
CROSS JOIN departamentos;

-- 6. SELF JOIN: Relacionar empleados con sus supervisores (simulación)
-- Asumiendo que los empleados pueden tener un supervisor_id que apunta a otro empleado
ALTER TABLE empleados ADD supervisor_id INT;

-- Actualizar la tabla empleados para agregar supervisores
UPDATE empleados SET supervisor_id = 2 WHERE id = 1;  -- Juan Pérez tiene como supervisor a Ana Gómez

-- Obtener empleados y sus supervisores (SELF JOIN)
SELECT e1.nombre AS empleado, e2.nombre AS supervisor
FROM empleados e1
LEFT JOIN empleados e2 ON e1.supervisor_id = e2.id;
```

2.10. Otras cláusulas del lenguaje.

En SQL, además de las cláusulas fundamentales como `SELECT`, `WHERE` y `JOIN`, existen muchas otras funcionalidades que permiten controlar y optimizar las consultas. Estas cláusulas y funciones proporcionan más flexibilidad y poder al momento de gestionar los datos. A continuación, se describen otras características importantes que permiten optimizar, manipular y presentar los datos de manera más avanzada.

2.10.1. `DISTINCT`: Eliminación de Duplicados

La cláusula `DISTINCT` se utiliza en consultas `SELECT` para eliminar filas duplicadas y devolver solo valores únicos. Es útil cuando se necesita una lista sin duplicados.

2.10.2. `LIMIT` y `OFFSET`: Control de Resultados

`LIMIT` restringe la cantidad de filas que se devuelven en una consulta, mientras que `OFFSET` salta un número específico de filas antes de devolver los resultados. Estas cláusulas son especialmente útiles para la paginación.

2.10.3. `HAVING`: Filtrar Grupos

`HAVING` se utiliza para filtrar grupos después de aplicar `GROUP BY`. Permite aplicar condiciones a los grupos generados, a diferencia de `WHERE` que filtra filas antes de la agrupación.

2.10.4. `ALIAS`: Renombrar Columnas y Tablas

Los alias permiten asignar nombres temporales a columnas o tablas. Esto es útil para mejorar la legibilidad y la organización de los resultados de las consultas.

2.10.5. `CASE`: Condiciones Condicionales

`CASE` permite realizar comparaciones condicionales dentro de una consulta, similar a una estructura `IF` en otros lenguajes de programación. Puedes definir múltiples condiciones y devolver diferentes valores según los resultados.

2.10.6. `CONCAT()`: Concatenación de Cadenas

La función `CONCAT()` combina varias cadenas en una sola. Es útil para crear resultados más legibles o formatear datos.

2.10.7. `WITH`: Subconsultas Comunes (CTE)

La cláusula `WITH` permite definir subconsultas comunes (Common Table Expressions, CTE) que se pueden reutilizar dentro de una consulta principal. Esto facilita la lectura y la reutilización de subconsultas.

2.10.8. Funciones de Ventana: `PARTITION BY`

Las funciones de ventana (`OVER` y `PARTITION BY`) permiten realizar cálculos sobre un conjunto de filas relacionadas, manteniendo las filas originales en la consulta. Es útil para agregar datos sin perder el detalle fila por fila.

2.10.9. `IF()`: Funciones Condicionales

`IF()` es una función condicional que devuelve un valor si una condición es verdadera y otro valor si es falsa. Es útil cuando se necesita tomar decisiones dentro de una consulta.

2.10.10. Otras Funciones Útiles

- **`COALESCE()`**: Devuelve el primer valor no nulo en una lista de expresiones.
- **`GROUP_CONCAT()`**: Devuelve valores agregados de una columna, concatenados en una sola cadena.
- **`ROUND()`**: Redondea un número a un número específico de decimales.
- **`LEAD()` y `LAG()`**: Devuelven el valor de una fila anterior o posterior en un conjunto de resultados, respectivamente.

Código de Ejemplo Completo 2.10.

```sql
-- 1. Uso de DISTINCT para eliminar duplicados
SELECT DISTINCT departamento_id FROM empleados;

-- 2. Uso de LIMIT y OFFSET para limitar y paginar resultados
SELECT nombre, salario FROM empleados LIMIT 3 OFFSET 1;

-- 3. Uso de HAVING para filtrar grupos con más de un empleado por departamento
SELECT departamento_id, COUNT(*) AS num_empleados
FROM empleados
GROUP BY departamento_id
HAVING COUNT(*) > 1;

-- 4. Uso de alias para renombrar columnas
SELECT nombre AS empleado, salario AS sueldo
FROM empleados;

-- 5. Uso de CASE para clasificar empleados según su salario
SELECT nombre,
    CASE
        WHEN salario > 4000 THEN 'Alto'
        WHEN salario BETWEEN 3000 AND 4000 THEN 'Medio'
        ELSE 'Bajo'
    END AS clasificacion_salarial
FROM empleados;

-- 6. Uso de CONCAT para unir el nombre y apellido en una sola columna
SELECT CONCAT(nombre, ' - ', salario) AS empleado_detalle
FROM empleados;

-- 7. Uso de WITH para definir subconsultas reutilizables
WITH SalariosAltos AS (
    SELECT nombre, salario
    FROM empleados
    WHERE salario > 4000
)
SELECT * FROM SalariosAltos;

-- 8. Uso de PARTITION BY para calcular el salario máximo por departamento
SELECT nombre, salario,
       MAX(salario) OVER (PARTITION BY departamento_id) AS max_salario_departamento
FROM empleados;

-- 9. Uso de IF para decidir entre diferentes valores en base a una condición
SELECT nombre,
       IF(departamento_id IS NULL, 'Sin departamento', 'Con departamento') AS estado_departamento
FROM empleados;

-- 10. Uso de COALESCE para manejar valores nulos en la columna departamento_id
SELECT nombre, COALESCE(departamento_id, 'No asignado') AS departamento
FROM empleados;

-- 11. Uso de GROUP_CONCAT para concatenar nombres de empleados por departamento
SELECT departamento_id, GROUP_CONCAT(nombre SEPARATOR ', ') AS empleados
FROM empleados
GROUP BY departamento_id;

-- 12. Uso de ROUND para redondear los salarios a enteros
SELECT nombre, ROUND(salario) AS salario_redondeado
FROM empleados;

-- 13. Uso de LEAD para obtener el salario del próximo empleado
SELECT nombre, salario,
       LEAD(salario) OVER (ORDER BY fecha_contratacion) AS siguiente_salario
FROM empleados;
```

2.11. Extensiones del lenguaje:

Las extensiones del lenguaje SQL proporcionan herramientas avanzadas para la gestión de bases de datos. Estas funcionalidades incluyen la creación y manipulación de vistas, la especificación de restricciones de integridad, la gestión de permisos a través de instrucciones de autorización, y el control de transacciones. A continuación, desarrollamos cada punto de estas extensiones, que son esenciales para un uso eficiente y seguro de las bases de datos.

2.11.1. Creación, manipulación y borrado de vistas.

Una **vista** es una tabla virtual basada en el resultado de una consulta. No almacena datos por sí misma, sino que presenta los datos de una o más tablas de una manera específica. Las vistas son útiles para simplificar consultas complejas, controlar el acceso a los datos y proporcionar diferentes perspectivas de los mismos.

1. **Creación de vistas**:
    - Las vistas se crean utilizando el comando `CREATE VIEW`. Puedes incluir cualquier consulta válida en la definición de la vista, lo que permite mostrar los datos de la manera deseada.
2. **Modificación de vistas**:
    - Para modificar una vista, puedes utilizar `CREATE OR REPLACE VIEW`, lo que permite redefinir la consulta sin eliminar la vista original.
3. **Eliminación de vistas**:
    - Las vistas se eliminan utilizando `DROP VIEW`, lo que borra la vista pero no los datos subyacentes.

2.11.2. Especificación de restricciones de integridad.

Las restricciones de integridad se utilizan para asegurar la consistencia y validez de los datos en la base de datos. Estas restricciones se aplican a nivel de columna o tabla, y permiten controlar la entrada de datos para garantizar que sigan las reglas establecidas.

1. **Clave primaria (PRIMARY KEY)**:
    - Garantiza que cada fila de una tabla tenga un identificador único y no nulo.
2. **Clave foránea (FOREIGN KEY)**:

- Establece relaciones entre tablas, asegurando que el valor en una columna coincida con el valor en la clave primaria de otra tabla.

3. **Restricción de unicidad (UNIQUE)**:
   - Garantiza que los valores en una columna o conjunto de columnas sean únicos.
4. **Restricción de no nulo (NOT NULL)**:
   - Impide que una columna contenga valores nulos.
5. **Restricción de comprobación (CHECK)**:
   - Permite definir una condición que los datos deben cumplir para ser aceptados en la tabla.

2.11.3. Instrucciones de autorización.

Las instrucciones de autorización permiten asignar permisos a diferentes usuarios o roles dentro de la base de datos. Esto es crucial para controlar quién puede acceder, modificar o borrar los datos.

1. **GRANT**:
   - Permite asignar permisos a un usuario o rol. Los permisos pueden incluir acceso para seleccionar, insertar, actualizar o borrar datos en una tabla.
2. **REVOKE**:
   - Permite retirar permisos previamente otorgados a un usuario o rol.
3. **ROLES**:
   - Los roles son grupos de permisos que se pueden asignar a múltiples usuarios, facilitando la gestión de permisos.

2.11.4. Control de las transacciones.

El control de transacciones es una parte crucial en el manejo de bases de datos, ya que permite agrupar una serie de operaciones en una sola transacción. Esto asegura que, si ocurre algún error, todas las operaciones de la transacción se revierten, manteniendo la consistencia de los datos.

1. **BEGIN TRANSACTION**:
   - Inicia una transacción.
2. **COMMIT**:
   - Confirma todos los cambios realizados durante la transacción, haciéndolos permanentes.
3. **ROLLBACK**:

- Revierte todos los cambios realizados durante la transacción en caso de error, restaurando los datos al estado anterior al inicio de la transacción.

4. **SAVEPOINT**:
   - Crea un punto de guardado dentro de una transacción al que se puede volver en caso de que solo parte de la transacción deba ser revertida.
5. **AUTOCOMMIT**:
   - Determina si las operaciones se confirman automáticamente o si necesitan ser explicitadas con `COMMIT`.

Código de Ejemplo Completo 2.11.

```sql
-- 2.11.1. Creación, manipulación y borrado de vistas

-- Creación de una vista que muestra empleados con salarios superiores a 4000
CREATE VIEW empleados_altos AS
SELECT nombre, salario
FROM empleados
WHERE salario > 4000;

-- Consulta de la vista
SELECT * FROM empleados_altos;

-- Modificación de la vista para incluir el departamento
CREATE OR REPLACE VIEW empleados_altos AS
SELECT nombre, salario, departamento_id
FROM empleados
WHERE salario > 4000;

-- Borrado de la vista
DROP VIEW empleados_altos;

-- 2.11.2. Especificación de restricciones de integridad

-- Creación de la tabla empleados con restricciones
CREATE TABLE empleados (
    id INT PRIMARY KEY,  -- Clave primaria
    nombre VARCHAR(100) NOT NULL,  -- No se permite que el nombre sea nulo
    salario DECIMAL(10, 2) CHECK (salario > 0),  -- El salario debe ser mayor que 0
    departamento_id INT,
    FOREIGN KEY (departamento_id) REFERENCES departamentos(id)  -- Clave foránea
);

-- 2.11.3. Instrucciones de autorización

-- Otorgar permisos a un usuario
GRANT SELECT, INSERT ON empleados TO 'usuario1';

-- Revocar permisos de un usuario
REVOKE INSERT ON empleados FROM 'usuario1';

-- Creación de un rol para gerentes con permisos de actualización y eliminación
CREATE ROLE gerente;
GRANT UPDATE, DELETE ON empleados TO gerente;

-- Asignación del rol a un usuario
GRANT gerente TO 'usuario2';

-- 2.11.4. Control de las transacciones

-- Inicio de una transacción
BEGIN TRANSACTION;

-- Inserción de un nuevo empleado
INSERT INTO empleados (id, nombre, salario, departamento_id)
VALUES (6, 'Marta Ramírez', 4800.00, 1);

-- Guardar un punto intermedio
SAVEPOINT save1;

-- Actualización del salario de un empleado
UPDATE empleados SET salario = 5000 WHERE id = 6;

-- Revertir la actualización al punto guardado
ROLLBACK TO save1;

-- Confirmación de los cambios
COMMIT;
```

2.12. El lenguaje de control de datos (DCL):

El **Lenguaje de Control de Datos (DCL)** es una parte fundamental de SQL que se encarga de la gestión de permisos y el control de transacciones dentro de la base de datos. DCL asegura que solo los usuarios autorizados puedan acceder y manipular los datos. Además, permite controlar el flujo de las transacciones para asegurar que las operaciones se ejecuten de manera adecuada y coherente.

2.12.1. Transacciones.

Una **transacción** en bases de datos es un conjunto de operaciones SQL que se ejecutan como una unidad atómica. Todas las operaciones dentro de una transacción deben completarse con éxito; si alguna falla, todas las operaciones se revierten, asegurando la integridad de los datos.

Las operaciones más comunes en la gestión de transacciones son:

1. **BEGIN TRANSACTION**: Inicia una nueva transacción.
2. **COMMIT**: Finaliza la transacción y confirma todos los cambios realizados.
3. **ROLLBACK**: Revierte los cambios realizados en la transacción y devuelve los datos a su estado anterior.

2.12.2. Propiedades de las transacciones: atomicidad, consistencia, aislamiento y permanencia:

Las transacciones en bases de datos deben cumplir con las propiedades **ACID**: **Atomicidad, Consistencia, Aislamiento y Durabilidad (Permanencia)**. Estas propiedades aseguran que las transacciones sean fiables y que los datos se mantengan íntegros.

1. **Atomicidad (Atomicity)**: Garantiza que una transacción se ejecute completamente o no se ejecute en absoluto. Si alguna operación falla, todos los cambios se deshacen y la base de datos vuelve a su estado anterior. Se asegura que las transacciones sean "todo o nada".
2. **Consistencia (Consistency)**: Asegura que una transacción lleve la base de datos de un estado válido a otro estado válido, cumpliendo todas las reglas de integridad y restricciones definidas. Cualquier transacción que viole estas reglas será revertida.
3. **Aislamiento (Isolation)**: Garantiza que las transacciones concurrentes no interfieran entre sí. Los cambios realizados en una transacción no son visibles para otras hasta que se confirmen, proporcionando independencia entre transacciones.

4. **Durabilidad (Durability)**: Asegura que una vez confirmada una transacción, sus cambios son permanentes, incluso en caso de fallos del sistema. Los datos se guardan de manera persistente y no se pierden tras un `COMMIT`.

2.12.2.1. Estados de una transacción: activa, parcialmente comprometida, fallida, abortada y comprometida.

Las transacciones pasan por diferentes estados durante su ejecución:

1. **Activa**: La transacción está en ejecución y puede realizar cambios en la base de datos.
2. **Parcialmente Comprometida**: La transacción ha completado todas las operaciones, pero aún no ha sido confirmada con un `COMMIT`.
3. **Fallida**: Ocurrió un error durante la ejecución de la transacción, lo que impide su éxito.
4. **Abortada**: La transacción fue revertida debido a un fallo o cancelación. Los cambios realizados hasta ese punto son revertidos con `ROLLBACK`.
5. **Comprometida**: La transacción ha finalizado exitosamente y sus cambios han sido confirmados con `COMMIT`.

2.12.2.2. Consultas y almacenamiento de estructuras en XML.

En algunas bases de datos, como MySQL, es posible almacenar y consultar datos en formato XML. Aunque SQL es un lenguaje estructurado para datos relacionales, el soporte para XML permite combinar estructuras de datos jerárquicas (como XML) dentro de un sistema de bases de datos relacional.

2.12.2.3. Estructura del diccionario de datos.

El **diccionario de datos** es una colección de metadatos que describe la estructura de la base de datos, como tablas, columnas, índices, restricciones y relaciones entre tablas. Cada vez que se crea o modifica un objeto de base de datos (como una tabla o vista), la información se almacena y actualiza en el diccionario de datos. En sistemas como Oracle y MySQL, existen vistas que proporcionan acceso a estos metadatos.

2.12.3. Control de las transacciones.

El control de las transacciones permite asegurar la coherencia y fiabilidad de los datos durante las operaciones. Además de `COMMIT` y `ROLLBACK`, otros conceptos importantes incluyen:

1. **SAVEPOINT**: Permite crear puntos de guardado dentro de una transacción. Si es necesario, se puede volver a un `SAVEPOINT` específico en lugar de realizar un `ROLLBACK` total.
2. **AUTOCOMMIT**: Algunas bases de datos permiten el modo de `AUTOCOMMIT`, donde cada operación SQL se confirma automáticamente sin necesidad de un `COMMIT` explícito.

2.12.4. Privilegios: autorizaciones y desautorizaciones.

Los **privilegios** en SQL se gestionan mediante las instrucciones **GRANT** y **REVOKE**. Estas instrucciones permiten controlar quién puede acceder y modificar los datos de la base de datos.

1. **GRANT**: Otorga permisos a un usuario o rol para realizar operaciones en la base de datos, como `SELECT`, `INSERT`, `UPDATE` o `DELETE`.
2. **REVOKE**: Retira los permisos previamente otorgados a un usuario o rol.

El control de privilegios es esencial para la seguridad de la base de datos, asegurando que solo los usuarios autorizados puedan realizar acciones específicas.

Código de Ejemplo Completo 2.12.

```sql
-- 2.12.1. Inicio de una transacción para insertar datos en la tabla empleados
START TRANSACTION;

-- Inserción de datos en la tabla departamentos
INSERT INTO departamentos (id, nombre_departamento)
VALUES (1, 'Recursos Humanos'), (2, 'Tecnología');

-- Inserción de datos en la tabla empleados
INSERT INTO empleados (id, nombre, salario, departamento_id)
VALUES (1, 'Juan Pérez', 3500.00, 1), (2, 'Ana Gómez', 4000.00, 1);

-- Punto de guardado para revertir parcialmente si es necesario
SAVEPOINT save1;

-- Actualización del salario de un empleado
UPDATE empleados SET salario = 5000 WHERE id = 1;

-- Reversión parcial a un punto de guardado
ROLLBACK TO save1;

-- Confirmación de la transacción
COMMIT;

-- 2.12.2.2. Consultas sobre estructuras XML (MySQL no tiene soporte nativo para XML avanzado como otros sistemas)
-- Ejemplo conceptual de cómo almacenar un XML en MySQL (como texto)
CREATE TABLE documentos_xml (
    id INT PRIMARY KEY,
    contenido XML
);

-- Inserción de un documento XML
INSERT INTO documentos_xml (id, contenido)
VALUES (1, '<empleado><nombre>Juan</nombre></empleado>');

-- Ejemplo de consulta de XML usando ExtractValue (solo para versiones antiguas de MySQL)
SELECT ExtractValue(contenido, '/empleado/nombre') AS nombre FROM documentos_xml WHERE id = 1;

-- 2.12.2.3. Consultas sobre el diccionario de datos en MySQL
-- Consultar las tablas creadas en la base de datos actual
SELECT table_name
FROM information_schema.tables
WHERE table_schema = DATABASE();

-- Consultar las columnas de una tabla específica
SELECT column_name, data_type
FROM information_schema.columns
WHERE table_name = 'empleados';

-- 2.12.3. Gestión de transacciones en MySQL
-- Inicio de otra transacción
START TRANSACTION;

-- Inserción de un nuevo empleado
INSERT INTO empleados (id, nombre, salario, departamento_id)
VALUES (3, 'Carlos Díaz', 3000.00, 2);

-- Creación de otro SAVEPOINT para posibles reversiones
SAVEPOINT save2;

-- Modificación del salario de otro empleado
UPDATE empleados SET salario = 4500 WHERE id = 2;

-- Reversión total de la transacción si es necesario
ROLLBACK;
```

```sql
-- 2.12.4. Privilegios: Autorizaciones y desautorizaciones en MySQL
-- Otorgar permisos de SELECT e INSERT a un usuario
GRANT SELECT, INSERT ON empleados TO 'usuario1'@'localhost';

-- Revocar permisos de INSERT de un usuario
REVOKE INSERT ON empleados FROM 'usuario1'@'localhost';

-- Creación de un rol 'gerente' con permisos completos sobre la tabla empleados
CREATE ROLE 'gerente';
GRANT SELECT, INSERT, UPDATE, DELETE ON empleados TO 'gerente';

-- Asignar el rol a un usuario específico
GRANT 'gerente' TO 'usuario2'@'localhost';

-- Mostrar los privilegios de un usuario
SHOW GRANTS FOR 'usuario2'@'localhost';

-- Revocar el rol de un usuario
REVOKE 'gerente' FROM 'usuario2'@'localhost';
```

2.13. Procesamiento y optimización de consultas:

El procesamiento y la optimización de consultas son aspectos fundamentales en la administración de bases de datos, ya que permiten mejorar el rendimiento y la eficiencia al recuperar, manipular y gestionar grandes volúmenes de datos. Las consultas mal optimizadas pueden afectar el rendimiento de la base de datos, generando tiempos de respuesta lentos y un uso excesivo de recursos. A continuación, se describen los diferentes aspectos involucrados en el procesamiento y optimización de consultas.

2.13.1. Procesamiento de una consulta.

El procesamiento de una consulta SQL implica una serie de pasos que el motor de la base de datos sigue para analizar, ejecutar y devolver los resultados solicitados. Los pasos típicos del procesamiento de una consulta son:

1. **Análisis léxico y sintáctico**:
   - El motor de la base de datos analiza la consulta para verificar que sea sintácticamente correcta. Se valida que las palabras clave, nombres de tablas, columnas y operadores estén correctamente definidos.
2. **Optimización de la consulta**:
   - En esta fase, el optimizador de la base de datos examina diferentes formas de ejecutar la consulta y selecciona el plan de ejecución más eficiente. Este proceso puede depender de factores como índices, estadísticas de la base de datos y el tamaño de las tablas involucradas.
3. **Generación del plan de ejecución**:
   - El motor de la base de datos crea un "plan de ejecución", que es un conjunto de pasos que indica cómo se accederá a los datos, en qué orden se

realizarán las operaciones y qué métodos de acceso (como índices) se utilizarán.

4. **Ejecución**:
    - El plan de ejecución se lleva a cabo, accediendo a las tablas y realizando las operaciones necesarias (búsqueda, filtrado, unión, etc.) para obtener los resultados.
5. **Retorno de resultados**:
    - Finalmente, los resultados de la consulta se devuelven al usuario o aplicación que emitió la solicitud.

2.13.2. Tipos de optimización: basada en reglas, basada en costes, otros.

El optimizador de consultas en una base de datos es responsable de elegir el plan de ejecución más eficiente. Existen diferentes enfoques de optimización, que se pueden clasificar en:

1. **Optimización basada en reglas**:
    - Este enfoque utiliza un conjunto de reglas predefinidas para determinar el orden y las operaciones que se realizarán en la consulta. Las reglas pueden priorizar, por ejemplo, las operaciones de selección antes que las de unión.
    - **Ejemplo de reglas comunes**:
        - Ejecutar las operaciones de filtrado (`WHERE`) antes de las uniones (`JOIN`).
        - Aplicar las operaciones que reduzcan el conjunto de datos antes de realizar cálculos o agregaciones.
    - Este método es rápido, pero no siempre selecciona el plan de ejecución más óptimo en todos los escenarios.
2. **Optimización basada en costes**:
    - En este enfoque, el optimizador calcula el "coste" de ejecutar la consulta mediante diferentes planes de ejecución posibles. El coste se evalúa en términos de recursos como el tiempo de CPU, las lecturas de disco y el uso de memoria. El plan con el coste estimado más bajo se selecciona para la ejecución.
    - **Factores que influyen en el coste**:
        - El tamaño de las tablas involucradas.
        - El número de registros devueltos por cada operación.
        - La disponibilidad de índices adecuados.
        - Las estadísticas de la base de datos (frecuencia de valores, distribución de datos).
3. **Otros tipos de optimización**:

- **Optimización heurística**: Basada en principios generales que suponen que ciertos patrones en las consultas siempre son más eficientes que otros (similar a la optimización por reglas).
- **Optimización adaptativa**: Algunos motores de bases de datos pueden cambiar el plan de ejecución en tiempo real si detectan que la ejecución de la consulta no está rindiendo como se esperaba.

2.13.3. Herramientas de la BBDD para la optimización de consultas.

Las bases de datos modernas ofrecen herramientas que permiten a los administradores y desarrolladores analizar y optimizar sus consultas. Algunas de estas herramientas incluyen:

1. **Plan de ejecución**: Un plan de ejecución muestra el conjunto de pasos que sigue la base de datos para ejecutar una consulta. Los usuarios pueden generar y analizar estos planes para identificar cuellos de botella o áreas que podrían optimizarse. En MySQL, se puede utilizar el comando `EXPLAIN` para obtener el plan de ejecución de una consulta.
2. **Índices**: Los índices son estructuras que mejoran la velocidad de recuperación de datos. Crear índices en columnas que se utilizan frecuentemente en las cláusulas `WHERE`, `JOIN` o `ORDER BY` puede mejorar significativamente el rendimiento de las consultas.
3. **Estadísticas de la base de datos**: Las estadísticas proporcionan información sobre la distribución de los datos en las tablas. El optimizador utiliza estas estadísticas para estimar los costes de diferentes planes de ejecución. En MySQL, las estadísticas se recopilan automáticamente, pero también se pueden actualizar manualmente.
4. **Consultas particionadas**: La partición de tablas permite dividir grandes volúmenes de datos en partes más manejables, mejorando así el rendimiento de las consultas. Cada partición se puede gestionar de manera independiente, lo que facilita el acceso a conjuntos de datos específicos sin necesidad de escanear toda la tabla.

Código de Ejemplo Completo 2.13.

```sql
-- 1. Procesamiento de una consulta
-- Ejecución de una consulta simple para obtener empleados con salarios mayores a 3000
SELECT * FROM empleados WHERE salario > 3000;

-- 2. Optimización basada en reglas y costes
-- Uso de EXPLAIN para obtener el plan de ejecución de la consulta anterior
EXPLAIN SELECT * FROM empleados WHERE salario > 3000;

-- 3. Herramientas de optimización
-- Actualización de las estadísticas para optimizar el plan de ejecución
ANALYZE TABLE empleados;

-- Uso de particiones en una tabla de ventas para optimización
CREATE TABLE ventas (
    id INT,
    fecha DATE,
    cantidad DECIMAL(10, 2)
)
PARTITION BY RANGE (YEAR(fecha)) (
    PARTITION p2020 VALUES LESS THAN (2021),
    PARTITION p2021 VALUES LESS THAN (2022)
);

-- Inserción de datos en la tabla ventas
INSERT INTO ventas (id, fecha, cantidad) VALUES (1, '2020-05-15', 150.00), (2, '2021-07-20', 200.00);

-- Consultar particiones
SELECT * FROM ventas WHERE fecha BETWEEN '2020-01-01' AND '2020-12-31';
```

2.14. Triggers

Un **trigger** en bases de datos es un procedimiento almacenado que se ejecuta automáticamente en respuesta a ciertos eventos que ocurren en una tabla o vista. Los triggers se utilizan para garantizar la integridad de los datos, automatizar procesos y responder a cambios en los datos sin intervención manual. Los eventos que pueden activar un trigger incluyen operaciones como `INSERT`, `UPDATE`, y `DELETE`.

A continuación, se describe el concepto de triggers, sus tipos, cómo funcionan y ejemplos de uso para maximizar su utilidad en la gestión de bases de datos.

2.14.1. Tipos de Triggers

Los triggers pueden clasificarse de diferentes maneras según cuándo se ejecutan y qué eventos los activan. Los dos tipos principales de triggers son:

1. **Triggers antes del evento (`BEFORE`)**:

- Estos triggers se ejecutan **antes** de que ocurra el evento que los activa. Se utilizan típicamente para validar o modificar los datos antes de que se inserten, actualicen o eliminen.
- **Ejemplo**: Validar que el salario de un empleado no sea menor que un cierto valor antes de permitir una actualización en la tabla.

2. **Triggers después del evento (`AFTER`)**:
   - Estos triggers se ejecutan **después** de que el evento ha ocurrido. Se usan comúnmente para realizar acciones dependientes del cambio de datos, como actualizar otras tablas, crear registros en una tabla de auditoría, o enviar notificaciones.
   - **Ejemplo**: Registrar un cambio en la tabla de empleados en una tabla de auditoría después de que se haya modificado un salario.

2.14.2. Eventos que Activan los Triggers

Un trigger puede activarse por diferentes eventos SQL en una tabla. Estos eventos incluyen:

1. `INSERT`:
   - El trigger se activa cuando se inserta un nuevo registro en la tabla.
   - **Ejemplo**: Cuando se añade un nuevo empleado a la tabla `empleados`, se puede activar un trigger para registrar la fecha de creación en una tabla de auditoría.
2. `UPDATE`:
   - El trigger se ejecuta cuando se actualiza un registro en la tabla.
   - **Ejemplo**: Si se actualiza el salario de un empleado, el trigger puede verificar que el nuevo salario sea mayor que el anterior y registrar este cambio en una tabla de auditoría.
3. `DELETE`:
   - El trigger se activa cuando se elimina un registro de la tabla.
   - **Ejemplo**: Cuando se elimina un empleado, se puede activar un trigger para registrar esta eliminación en una tabla de auditoría o realizar acciones relacionadas, como reactivar una posición vacante.

2.14.3. Elementos de un Trigger

Un trigger se define con varios componentes esenciales:

1. **Evento que lo activa** (`INSERT`, `UPDATE`, `DELETE`).

2. **Momento en que se activa** (`BEFORE` o `AFTER` el evento).
3. **Tabla objetivo**: La tabla en la que ocurre el evento que activa el trigger.
4. **Condición opcional**: Una lógica que puede limitar cuándo el trigger debe ejecutarse (por ejemplo, solo si un valor específico ha cambiado).
5. **Acciones**: El conjunto de instrucciones SQL que se ejecutarán cuando se dispare el trigger.

2.14.4. Triggers en MySQL

En MySQL, los triggers se utilizan para garantizar la coherencia y ejecutar acciones automatizadas en respuesta a cambios en los datos. Se definen con el comando `CREATE TRIGGER` y pueden asociarse a eventos específicos en tablas.

- **Sintaxis básica de un trigger**:

```
CREATE TRIGGER trigger_name
BEFORE|AFTER INSERT|UPDATE|DELETE
ON tabla
FOR EACH ROW
BEGIN
    -- Instrucciones SQL a ejecutar
END;
```

2.14.5. Limitaciones y Consideraciones de Uso de los Triggers

1. **Impacto en el rendimiento**:
    - Dado que los triggers se ejecutan automáticamente cada vez que se produce un evento, pueden impactar negativamente en el rendimiento si no se utilizan adecuadamente, especialmente en tablas con gran cantidad de registros o transacciones frecuentes.
2. **Complejidad**:
    - Es importante tener en cuenta que los triggers pueden aumentar la complejidad de la base de datos, ya que pueden ejecutar procesos automáticos que no son fácilmente visibles en las consultas normales.
3. **Recursión**:
    - Algunos sistemas de bases de datos, como MySQL, no permiten triggers recursivos, es decir, triggers que se activen a sí mismos de manera indefinida.

2.14.6. Ejemplos de Uso de Triggers

Los triggers se pueden utilizar en una variedad de escenarios. A continuación, se describen algunos de los casos más comunes:

1. **Auditoría**:
    - Registrar cambios en los datos. Por ejemplo, cuando un empleado cambia su salario, un trigger puede almacenar el cambio en una tabla de auditoría con la fecha del cambio y el usuario que realizó la modificación.
2. **Validación de datos**:
    - Validar que los datos cumplan con ciertos requisitos antes de insertarse o actualizarse. Por ejemplo, asegurarse de que el salario de un empleado no sea menor que un mínimo permitido.
3. **Mantenimiento de integridad referencial**:
    - Actualizar tablas relacionadas o mantener integridad referencial entre varias tablas.
4. **Automatización de procesos**:
    - Ejecutar procesos automáticos en respuesta a cambios, como enviar notificaciones o crear tareas basadas en los cambios en los datos.

Código de Ejemplo Completo 2.14.

```sql
-- Creación de la tabla empleados y la tabla de auditoría
CREATE TABLE empleados (
    id INT PRIMARY KEY,
    nombre VARCHAR(100),
    salario DECIMAL(10, 2),
    departamento_id INT
);

CREATE TABLE auditoria_salarios (
    id INT PRIMARY KEY AUTO_INCREMENT,
    empleado_id INT,
    salario_anterior DECIMAL(10, 2),
    salario_nuevo DECIMAL(10, 2),
    fecha_cambio TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- 1. Creación de un trigger que se ejecuta antes de actualizar el salario de un empleado
CREATE TRIGGER antes_actualizar_salario
BEFORE UPDATE ON empleados
FOR EACH ROW
BEGIN
    -- Insertar el cambio en la tabla de auditoría
    IF NEW.salario <> OLD.salario THEN
        INSERT INTO auditoria_salarios (empleado_id, salario_anterior, salario_nuevo)
        VALUES (OLD.id, OLD.salario, NEW.salario);
    END IF;
END;

-- 2. Inserción de datos de ejemplo en la tabla empleados
INSERT INTO empleados (id, nombre, salario, departamento_id)
VALUES (1, 'Juan Pérez', 3000.00, 1),
       (2, 'Ana Gómez', 4000.00, 2);

-- 3. Actualización de un salario (activará el trigger)
UPDATE empleados SET salario = 4500.00 WHERE id = 1;

-- 4. Verificación de los cambios en la tabla de auditoría
SELECT * FROM auditoria_salarios;

-- 5. Creación de un trigger que se ejecuta después de la inserción de un nuevo empleado
CREATE TRIGGER despues_insertar_empleado
AFTER INSERT ON empleados
FOR EACH ROW
BEGIN
    -- Insertar un mensaje de bienvenida a una tabla de mensajes (por ejemplo)
    INSERT INTO mensajes_bienvenida (empleado_id, mensaje)
    VALUES (NEW.id, CONCAT('Bienvenido, ', NEW.nombre, ', a la empresa!'));
END;
```