28-11-2024

SQL HANDBOOK

TEMAS SELECTOS DE ESTADÍSTICA

Sergio Alberto Álvarez Montemayor
EXPEDIENTE 300706

Tabla de contenido

¿Qué es SQL?

Definición de SQL

SQL, que significa **Structured Query Language** (Lenguaje de Consulta Estructurado), es un lenguaje estándar utilizado para gestionar y manipular bases de datos relacionales. SQL permite interactuar con bases de datos de manera sencilla y eficiente, permitiendo realizar operaciones como la consulta, actualización, inserción y eliminación de datos.

En resumen, SQL es un lenguaje diseñado específicamente para trabajar con bases de datos. Está basado en la manipulación de tablas que organizan la información en columnas y filas, y en el uso de operaciones para recuperar, modificar y gestionar esos datos.

Historia y Evolución de SQL

SQL fue desarrollado en la década de 1970 por investigadores de IBM bajo el nombre inicial de **SEQUEL** (Structured English Query Language). El lenguaje fue diseñado como parte de un proyecto para crear una base de datos relacional llamada **System R**. Con el tiempo, SEQUEL fue renombrado a **SQL** debido a razones de marca registrada.

A lo largo de los años, SQL ha evolucionado en respuesta a los avances en la tecnología de bases de datos y las necesidades de los usuarios. En 1986, SQL fue adoptado como estándar por el **American National Standards Institute (ANSI)**, y más tarde por la **International Organization for Standardization (ISO)**. Desde entonces, SQL ha sido adoptado por la mayoría de los sistemas de bases de datos relacionales.

Hoy en día, SQL es el estándar universal para interactuar con bases de datos, utilizado en sistemas como **MySQL**, **PostgreSQL**, **Microsoft SQL Server**, **Oracle Database**, **SQLite**, y muchos más.

¿Por qué SQL es importante?

SQL es fundamental para trabajar con bases de datos relacionales debido a las siguientes razones:

- **Facilidad de uso**: SQL tiene una sintaxis sencilla y legible. Los comandos en SQL se parecen al inglés, lo que facilita su aprendizaje y uso.
- **Estandarización**: SQL es un lenguaje estándar ampliamente aceptado, lo que significa que se puede utilizar en una variedad de sistemas de bases de datos sin necesidad de aprender un lenguaje específico para cada uno.

- **Eficiencia**: SQL permite realizar consultas complejas de manera eficiente y con un alto rendimiento. También ofrece optimización de consultas y la posibilidad de crear índices para mejorar la velocidad de las consultas.
- **Versatilidad**: SQL permite realizar una amplia gama de operaciones sobre los datos, como filtrado, agrupación, ordenación, uniones entre tablas, transacciones y control de acceso.
- **Interoperabilidad**: SQL se puede integrar con otros lenguajes de programación, como **Python**, **Java**, **PHP** o **C#**, para permitir aplicaciones más completas que interactúan con bases de datos.

Principales Características de SQL

1. **Lenguaje Declarativo**: SQL es un lenguaje declarativo, lo que significa que el usuario especifica *qué* quiere hacer (por ejemplo, recuperar datos) en lugar de *cómo* hacerlo (cómo procesar los datos).
2. **Operaciones en Datos Relacionales**: SQL trabaja sobre el modelo de datos relacional, que organiza la información en tablas (o relaciones). Estas tablas pueden estar interrelacionadas mediante claves primarias y foráneas, lo que permite realizar consultas complejas entre diferentes tablas.
3. **Comandos Estructurados**: SQL utiliza un conjunto de comandos estructurados que se agrupan en diferentes categorías:
    - **DDL (Data Definition Language)**: Comandos para definir la estructura de la base de datos, como CREATE, ALTER, DROP.
    - **DML (Data Manipulation Language)**: Comandos para manipular datos dentro de las tablas, como SELECT, INSERT, UPDATE, DELETE.
    - **DCL (Data Control Language)**: Comandos para definir los permisos y el acceso a los datos, como GRANT, REVOKE.
    - **TCL (Transaction Control Language)**: Comandos para gestionar las transacciones de la base de datos, como COMMIT, ROLLBACK, SAVEPOINT.

4. **Transacciones**: SQL soporta transacciones, que son secuencias de operaciones que se ejecutan como una única unidad. Las transacciones tienen cuatro propiedades fundamentales conocidas como **ACID**:
    - **Atomicidad**: Todas las operaciones dentro de una transacción se completan correctamente o ninguna se realiza.
    - **Consistencia**: La base de datos pasa de un estado válido a otro estado válido.
    - **Aislamiento**: Las transacciones concurrentes no interfieren entre sí.
    - **Durabilidad**: Los cambios realizados por una transacción son permanentes, incluso en caso de fallos del sistema.

Componentes de una Base de Datos Relacional

En SQL, la información se organiza en **tablas**, que están compuestas por **filas** (registros) y **columnas** (campos). A continuación se describen los componentes clave que interactúan con SQL en una base de datos relacional:

1. **Tablas**: Son la estructura básica donde se almacenan los datos en una base de datos. Cada tabla se define por un conjunto de columnas que describen los atributos del dato almacenado. Ejemplo de tabla:

ID	NOMBRE	EDAD	SALARIO
1	Juan	30	2000
2	María	25	2200
3	Pedro	40	2500

2. **Filas** (Registros): Representan los datos almacenados en las tablas. Cada fila contiene valores para cada una de las columnas de la tabla.
3. **Columnas** (Campos): Cada columna de una tabla representa un atributo de los datos almacenados, y se define con un nombre y un tipo de dato específico (como INTEGER, VARCHAR, DATE, etc.).

4. **Claves**:
    - **Clave primaria (Primary Key)**: Una columna o un conjunto de columnas que identifican de manera única cada fila en la tabla.
    - **Clave foránea (Foreign Key)**: Una columna en una tabla que hace referencia a la clave primaria de otra tabla, permitiendo establecer relaciones entre las tablas.

Tipos de Consultas en SQL

SQL permite realizar consultas para interactuar con los datos en la base de datos. Las consultas se pueden clasificar en los siguientes tipos:

- **Consultas SELECT**: Son las más comunes, utilizadas para obtener datos de una o más tablas.
- SELECT nombre, edad FROM empleados WHERE salario > 2000;
- **Consultas de inserción**: Utilizadas para agregar nuevos registros a una tabla.
- INSERT INTO empleados (nombre, edad, salario) VALUES ('Carlos', 35, 3000);
- **Consultas de actualización**: Se utilizan para modificar datos existentes en una tabla.
- UPDATE empleados SET salario = 3200 WHERE nombre = 'Carlos';
- **Consultas de eliminación**: Permiten borrar registros de una tabla.
- DELETE FROM empleados WHERE nombre = 'Carlos';

¿Por qué aprender SQL?

- **Habilidades requeridas en el mercado laboral**: SQL es uno de los lenguajes más demandados en el ámbito de la tecnología y la gestión de datos. Muchas empresas dependen de SQL para almacenar, gestionar y analizar grandes volúmenes de datos.
- **Fundamental para el análisis de datos**: Si estás interesado en el análisis de datos o el desarrollo de software, SQL es una habilidad esencial, ya que permite extraer, limpiar y manipular los datos que provienen de diversas fuentes.

- **Universalidad**: SQL es utilizado por la mayoría de los sistemas de bases de datos relacionales (como MySQL, PostgreSQL, Microsoft SQL Server, Oracle, etc.), por lo que aprender SQL te proporcionará una base sólida para trabajar con diferentes plataformas.

Diseño de Base de Datos Relacional

El diseño de una base de datos relacional es un proceso fundamental para crear una estructura que sea eficiente, flexible y escalable. Este proceso no solo implica crear tablas, sino también definir relaciones entre ellas, garantizar la integridad de los datos y optimizar el rendimiento. A continuación, exploramos los principios clave y las mejores prácticas para diseñar una base de datos relacional.

1. Principios del Diseño de Base de Datos Relacional

El diseño de una base de datos relacional implica varios pasos que incluyen la organización de los datos, la definición de relaciones entre las tablas, y la implementación de reglas para garantizar que los datos sean coherentes, completos y eficientemente accesibles.

Los **principios clave** del diseño de base de datos relacional incluyen:

1. **Normalización**:
    - La **normalización** es el proceso de organizar los datos en tablas de manera que se reduzca la redundancia y se eviten las dependencias no deseadas. El objetivo es asegurar que la base de datos esté en **formas normales** que optimicen la integridad y la eficiencia.
    - Las **formas normales** son reglas que definen cómo deben organizarse las tablas:
        - **Primera Forma Normal (1NF)**: Cada columna de la tabla debe contener solo un valor atómico (sin listas ni valores múltiples). Además, cada fila debe ser única, lo que significa que debe tener una clave primaria.
        - **Segunda Forma Normal (2NF)**: Cumple con 1NF y elimina las dependencias parciales, es decir, las columnas no clave deben depender completamente de la clave primaria.
        - **Tercera Forma Normal (3NF)**: Cumple con 2NF y elimina las

dependencias transitivas, es decir, no debe haber dependencias entre las columnas no clave.

- **Cuarta Forma Normal (4NF)**: Elimina las dependencias multivaluadas, donde una columna depende de más de un conjunto de columnas.

Ejemplo de normalización:

Supón que tenemos la siguiente tabla no normalizada:

ID_VENTA	ID_CLIENTE	CLIENTE_NOMBRE	PRODUCTO	PRECIO
1001	1	Juan Pérez	Camisa	30
1002	2	María Gómez	Pantalón	40
1003	1	Juan Pérez	Zapatos	50

Para llevarla a 1NF, debemos separar las combinaciones de productos y precios en filas separadas, ya que cada fila debe tener solo un valor atómico. Luego, para llegar a 2NF, podemos dividir la tabla en dos tablas: una para las ventas y otra para los clientes, eliminando la redundancia de los datos del cliente en cada fila.

2. **Desnormalización**: La **desnormalización** es el proceso opuesto a la normalización, donde se introducen redundancias de datos para mejorar el rendimiento de las consultas, a veces a costa de la integridad. Se utiliza en bases de datos con grandes volúmenes de datos donde las consultas complejas y las uniones entre tablas son costosas en términos de tiempo de ejecución.
3. **Consistencia de los Datos**:
    - Es esencial garantizar que los datos sean correctos y completos. Esto se logra a través de **restricciones de integridad**, como claves primarias, claves foráneas y restricciones de unicidad.
    - Las **claves primarias** identifican de forma única cada fila en una tabla.
    - Las **claves foráneas** aseguran que las relaciones entre las tablas sean válidas y

que los datos estén correctamente relacionados.

4. **Escalabilidad y Rendimiento**:
    - El diseño de la base de datos debe tener en cuenta la **escalabilidad**, lo que significa que debe ser capaz de manejar un aumento en el volumen de datos sin perder rendimiento. Esto incluye la optimización de las consultas, el uso adecuado de índices y la gestión de la carga de trabajo.
    - La implementación de **índices** es crucial para acelerar las búsquedas y las consultas sobre las tablas, pero debe hacerse de manera cuidadosa, ya que los índices también pueden afectar el rendimiento de las inserciones y actualizaciones.

2. Pasos para Diseñar una Base de Datos Relacional

El proceso de diseño de una base de datos relacional consta de varias etapas clave:

Paso 1: Recolección de Requisitos

Antes de comenzar a diseñar la base de datos, es esencial comprender las necesidades del negocio y los requisitos funcionales. Esto involucra trabajar estrechamente con los usuarios o stakeholders para entender:

- Qué datos necesitan almacenarse.
- Cómo se interrelacionan esos datos.
- Qué operaciones se deben realizar sobre los datos (consultas, actualizaciones, etc.).

Un buen análisis de requisitos asegura que la base de datos satisfaga las necesidades reales de los usuarios y no se pierda tiempo diseñando una estructura innecesariamente compleja.

Paso 2: Creación del Modelo Entidad-Relación (ER)

El **Modelo Entidad-Relación (ER)** es una herramienta visual que ayuda a conceptualizar la estructura de la base de datos. En este modelo, se representan las **entidades** (tablas) y las **relaciones** entre ellas.

- **Entidad**: Cualquier objeto o concepto del mundo real que tiene relevancia para el negocio y se puede identificar de forma única. Ejemplos de entidades son: Cliente, Producto, Empleado, etc.
- **Atributos**: Son las características de las entidades. Por ejemplo, un Cliente podría tener atributos como nombre, dirección, y teléfono.
- **Relaciones**: Describen cómo se conectan las entidades entre sí. Pueden ser de tipo uno a uno, uno a muchos o muchos a muchos. Por ejemplo, un Cliente puede realizar muchas Ventas, pero cada Venta está asociada con un solo Cliente.

A través del modelo ER, podemos visualizar las tablas y las relaciones clave entre ellas.

Paso 3: Definición de Tablas y Relaciones

Una vez que se ha creado el modelo ER, el siguiente paso es transformar este modelo en tablas reales de base de datos. En esta etapa, definimos:

- Las **tablas** y sus **columnas**.
- La **clave primaria** para cada tabla.
- Las **claves foráneas** para representar las relaciones entre tablas.

Ejemplo: Supongamos que tenemos un modelo ER con dos entidades: Cliente y Venta. Definimos las tablas como sigue:

- **Cliente**:

id_cliente	**nombre**	**teléfono**
1	Juan Pérez	555-1234
2	María Gómez	555-5678

- **Venta**:

id_venta	id_cliente	fecha	monto
1001	1	2024-11-01	150
1002	2	2024-11-05	200

Aquí, id_cliente en la tabla Venta es una **clave foránea** que hace referencia a la tabla Cliente.

Paso 4: Aplicar Reglas de Integridad

Las **reglas de integridad** aseguran que los datos sean consistentes y correctos. Las principales son:

- **Integridad de entidad**: Cada tabla debe tener una clave primaria única que identifique de manera única cada fila.
- **Integridad referencial**: Las claves foráneas deben hacer referencia a una clave primaria existente. Si se elimina o actualiza una fila en la tabla relacionada, la base de datos debe asegurarse de que los datos en las tablas relacionadas también se actualicen o eliminen en consecuencia (a través de acciones como **CASCADE**).
- **Restricciones de unicidad y valores nulos**: Debemos definir qué columnas deben ser únicas y cuáles pueden aceptar valores nulos.

Paso 5: Normalización y Optimización

Una vez que se han definido las tablas y relaciones, es crucial aplicar el proceso de **normalización** para evitar redundancia de datos y garantizar que la base de datos sea eficiente. Asegúrate de que cada tabla esté en al menos 3NF.

Sin embargo, en ciertos casos, puede ser necesario desnormalizar para mejorar el rendimiento de las consultas complejas, especialmente cuando se trata de bases de datos de gran volumen.

Paso 6: Implementación y Mantenimiento

Una vez que se ha diseñado y normalizado la base de datos, se puede proceder con su implementación utilizando SQL para crear las tablas, relaciones, y restricciones. Es importante monitorear continuamente el rendimiento de la base de datos y realizar ajustes según sea

necesario.

3. Buenas Prácticas en el Diseño de Bases de Datos Relacionales

1. **Utilizar nombres claros y significativos**: Nombrar las tablas y columnas de manera que reflejen claramente los datos que contienen.
2. **Evitar la redundancia**: Utilizar claves foráneas para establecer relaciones entre tablas en lugar de duplicar datos.
3. **Normalizar hasta la 3NF**: Siempre que sea posible, normalizar la base de datos hasta la tercera forma normal para evitar anomalías y redundancias

. 4. **Documentar el diseño**: Mantener documentación actualizada sobre el modelo de datos y las relaciones entre tablas.

Consultas Simples en SQL

Las consultas SQL simples son las más básicas y fundamentales que se utilizan para interactuar con bases de datos. A través de estas consultas, podemos recuperar, insertar, actualizar y eliminar datos de las tablas de una base de datos sin necesidad de realizar un JOIN (que implica la combinación de múltiples tablas). Este capítulo se centra en las consultas más simples para recuperar y manipular datos en una sola tabla.

1. Introducción a las Consultas Simples

Las consultas simples en SQL son aquellas que operan sobre una única tabla y no requieren la combinación de múltiples tablas a través de relaciones. Se utilizan principalmente para **seleccionar (consultar) datos**, **insertar nuevos registros**, **modificar datos existentes**, y **eliminar registros**.

En SQL, la estructura básica de una consulta depende de la acción que desees realizar. A continuación, exploraremos las consultas más comunes:

- **SELECT**: Para recuperar datos.
- **INSERT**: Para insertar datos.
- **UPDATE**: Para actualizar datos.
- **DELETE**: Para eliminar datos.

2. Consulta SELECT: Recuperar Datos

El comando **SELECT** se utiliza para recuperar datos de una o más columnas de una tabla. A continuación se muestra la sintaxis básica de una consulta **SELECT**:

SELECT columna1, columna2, ... FROM nombre_tabla;

- columna1, columna2, etc. son los nombres de las columnas que deseas recuperar.
- nombre_tabla es el nombre de la tabla de la que deseas obtener los datos.

Si deseas recuperar todos los datos de la tabla, puedes usar el símbolo * para indicar que deseas seleccionar todas las columnas:

SELECT * FROM empleados;

Ejemplo:

Si tienes una tabla empleados con las siguientes columnas: id_empleado, nombre, edad, salario, y deseas obtener todos los datos de todos los empleados, la consulta sería:

SELECT * FROM empleados;

Esto devolvería todos los registros de la tabla empleados.

3. Filtrar Resultados con WHERE

La cláusula **WHERE** se utiliza para filtrar los resultados de una consulta **SELECT** en función de una condición específica. Esto permite obtener solo los datos que cumplen con ciertas condiciones.

La sintaxis es la siguiente:

SELECT columna1, columna2, ... FROM nombre_tabla WHERE condición;

Ejemplo:

Supongamos que quieres obtener todos los empleados cuyo salario sea mayor a 2000. La consulta sería:

SELECT * FROM empleados WHERE salario > 2000;

Si deseas consultar solo los empleados cuyo nombre sea "Juan Pérez":

SELECT * FROM empleados WHERE nombre = 'Juan Pérez';

Condiciones comunes en WHERE:

- **=**: Igual a
- **<>** o **!=**: Diferente de
- **>**: Mayor que
- **<**: Menor que

- **>=**: Mayor o igual que
- **<=**: Menor o igual que
- **BETWEEN**: Para un rango de valores
- **IN**: Para un conjunto de valores específicos
- **LIKE**: Para coincidencias parciales (uso de comodines)
- **IS NULL**: Para comprobar valores nulos

4. Ordenar Resultados con ORDER BY

La cláusula **ORDER BY** se utiliza para ordenar los resultados de una consulta en función de una o más columnas. Los resultados pueden ordenarse de forma ascendente (**ASC**) o descendente (**DESC**).

La sintaxis es la siguiente:

SELECT columna1, columna2, ... FROM nombre_tabla ORDER BY columna1 [ASC|DESC];

Ejemplo:

Para obtener todos los empleados ordenados por salario de menor a mayor:

SELECT * FROM empleados ORDER BY salario ASC;

Para obtener todos los empleados ordenados por salario de mayor a menor:

SELECT * FROM empleados ORDER BY salario DESC;

También puedes ordenar por múltiples columnas. Por ejemplo, si deseas ordenar primero por edad y luego por salario, ambos de forma ascendente:

SELECT * FROM empleados ORDER BY edad ASC, salario ASC;

5. Limitar el Número de Resultados con LIMIT (o TOP)

En algunas bases de datos (como MySQL o PostgreSQL), puedes limitar el número de registros devueltos por una consulta utilizando la palabra clave **LIMIT**. En SQL Server, se usa **TOP** para el mismo propósito.

La sintaxis básica de **LIMIT** es:

SELECT columna1, columna2, ... FROM nombre_tabla LIMIT número;

Ejemplo:

Si deseas obtener solo los primeros 5 empleados con el salario más alto, puedes usar la siguiente consulta:

SELECT * FROM empleados ORDER BY salario DESC LIMIT 5;

En SQL Server, la consulta equivalente sería:

SELECT TOP 5 * FROM empleados ORDER BY salario DESC;

6. Insertar Datos con INSERT INTO

El comando **INSERT INTO** se utiliza para agregar nuevos registros a una tabla. Su sintaxis básica es la siguiente:

INSERT INTO nombre_tabla (columna1, columna2, ...) VALUES (valor1, valor2, ...);

Ejemplo:

Supón que deseas insertar un nuevo empleado en la tabla empleados con los siguientes datos: id_empleado = 6, nombre = 'Carlos López', edad = 30, salario = 2200. La consulta sería:

INSERT INTO empleados (id_empleado, nombre, edad, salario)

VALUES (6, 'Carlos López', 30, 2200);

7. Actualizar Datos con UPDATE

El comando **UPDATE** se utiliza para modificar los registros existentes en una tabla. La sintaxis básica de **UPDATE** es:

UPDATE nombre_tabla SET columna1 = valor1, columna2 = valor2, ... WHERE condición;

Es importante utilizar la cláusula **WHERE** para especificar qué registros deben ser actualizados, de lo contrario, **todos los registros de la tabla** se actualizarán.

Ejemplo:

Si deseas actualizar el salario de un empleado llamado "Juan Pérez" para que su salario sea 2500, la consulta sería:

UPDATE empleados SET salario = 2500 WHERE nombre = 'Juan Pérez';

Si no incluyes una condición en el **WHERE**, todos los registros de la tabla serán actualizados. Por ejemplo:

UPDATE empleados SET salario = 2500; -- Esto cambiaría el salario de todos los empleados

8. Eliminar Datos con DELETE

El comando **DELETE** se utiliza para eliminar registros de una tabla. La sintaxis básica de **DELETE** es:

DELETE FROM nombre_tabla WHERE condición;

Es crucial usar la cláusula **WHERE** para evitar eliminar todos los registros de la tabla.

Ejemplo:

Para eliminar un empleado llamado "Carlos López" de la tabla empleados, puedes usar la siguiente consulta:

DELETE FROM empleados WHERE nombre = 'Carlos López';

Si deseas eliminar todos los registros de la tabla (cuidado con esta acción, ya que no se puede deshacer), puedes usar:

DELETE FROM empleados;

9. Funciones de Agregación en Consultas Simples

SQL también permite realizar operaciones de agregación sobre los datos, como contar registros, obtener promedios, sumas, etc. Las funciones de agregación comunes son:

- **COUNT()**: Cuenta el número de registros.
- **SUM()**: Suma los valores de una columna.
- **AVG()**: Calcula el promedio de los valores de una columna.
- **MAX()**: Obtiene el valor máximo de una columna.
- **MIN()**: Obtiene el valor mínimo de una columna.

Ejemplo:

Si deseas contar el número total de empleados en la tabla:

SELECT COUNT(*) FROM empleados;

Si deseas obtener la suma total de los salarios de todos los empleados:

SELECT SUM(salario) FROM empleados;

Consultas JOIN

En SQL, las consultas **JOIN** son utilizadas para combinar datos de dos o más tablas en una sola consulta. Aunque en este capítulo no vamos a centrarnos en los tipos específicos de JOIN (como **INNER JOIN**, **LEFT JOIN**, etc.), es importante comprender la estructura básica de un JOIN, cómo funciona y cómo se puede usar para recuperar información de múltiples tablas de una manera eficiente.

1. Introducción a las Consultas JOIN

Cuando tenemos una base de datos relacional, es común que los datos se distribuyan en varias tablas. Cada tabla contiene información relacionada pero separada por motivos de eficiencia,

integridad y organización. Las consultas JOIN permiten que combinemos esta información, de modo que podamos obtener un conjunto de resultados más completo.

La clave para realizar un JOIN entre dos o más tablas es que deben existir **relaciones entre ellas**. Estas relaciones generalmente se establecen mediante **claves primarias** y **claves foráneas**.

2. Sintaxis General de una Consulta JOIN

En su forma más básica, un **JOIN** se utiliza para combinar columnas de dos o más tablas basadas en una condición que vincula las tablas entre sí, usualmente a través de las claves foráneas.

La sintaxis general para un JOIN es:

```
SELECT columnas
FROM tabla1
JOIN tabla2 ON tabla1.columna_relacionada = tabla2.columna_relacionada;
```

- **tabla1** y **tabla2** son las tablas que estás combinando.
- **columna_relacionada** es la columna que conecta ambas tablas. Usualmente, una columna en **tabla1** es una clave foránea que se relaciona con una clave primaria en **tabla2**.

Ejemplo:

Imagina que tienes dos tablas:

- **empleados**: Contiene información sobre empleados.

id_empleado	nombre	departamento_id
1	Juan Pérez	1
2	Ana López	2

- **departamentos**: Contiene información sobre los departamentos.

id_departamento	nombre_departamento
1	Recursos Humanos
2	IT

Si quieres obtener una lista de empleados con el nombre de su departamento, puedes hacer lo siguiente:

```
SELECT empleados.nombre, departamentos.nombre_departamento
FROM empleados
JOIN departamentos ON empleados.departamento_id = departamentos.id_departamento;
```

Este **JOIN** combina las dos tablas, basándose en la relación entre empleados.departamento_id y

departamentos.id_departamento, y devuelve una tabla con el nombre de cada empleado junto con el nombre del departamento al que pertenece.

3. Entendiendo el Comportamiento de los JOIN

Aunque no estamos explorando los diferentes tipos de JOIN, es importante entender el comportamiento básico de cómo funcionan las uniones entre tablas:

- **La cláusula ON** define la condición de unión. En nuestro ejemplo, estamos uniendo las tablas empleados y departamentos donde empleados.departamento_id es igual a departamentos.id_departamento.
- Si no hay ninguna coincidencia en la condición de unión, el resultado de la consulta no incluirá esas filas de la tabla principal (empleados en este caso). Dependiendo del tipo de JOIN que utilices, podrías obtener más o menos registros, pero esto es algo que exploraremos en detalle cuando cubramos los diferentes tipos de JOIN más adelante.

4. Utilizando Alias para Mayor Claridad

Cuando trabajamos con consultas JOIN, especialmente con tablas que tienen nombres largos o cuando estamos uniendo varias tablas, puede ser útil usar **alias** para referirse a las tablas de manera más corta y clara.

La sintaxis de alias es la siguiente:

```
SELECT e.nombre, d.nombre_departamento
FROM empleados AS e
JOIN departamentos AS d ON e.departamento_id = d.id_departamento;
```

En este caso, hemos asignado un alias a la tabla empleados como e y a la tabla departamentos como d. Esto hace que la consulta sea más legible, especialmente cuando hay muchas tablas involucradas.

5. Combinar Varias Tablas con JOIN

Si necesitas combinar más de dos tablas en una sola consulta, simplemente puedes agregar más cláusulas **JOIN**. Es importante que cada tabla esté relacionada con la tabla anterior a través de una condición **ON**.

Ejemplo de combinación de tres tablas:

Supón que tenemos tres tablas:

- **empleados**: Contiene información de los empleados.
- **departamentos**: Contiene información de los departamentos.
- **proyectos**: Contiene información sobre los proyectos en los que los empleados trabajan.

Tablas:

Empleados:

ID_EMPLEADO	NOMBRE	DEPARTAMENTO_ID
1	Juan Pérez	1
2	Ana López	2

Departamentos:

ID_DEPARTAMENTO	NOMBRE_DEPARTAMENTO
1	Recursos Humanos
2	IT

Proyectos:

ID_PROYECTO	NOMBRE_PROYECTO	ID_EMPLEADO
101	Proyecto A	1
102	Proyecto B	2

Para obtener una lista de empleados, sus departamentos y los proyectos en los que trabajan, la consulta JOIN sería:

```
SELECT e.nombre, d.nombre_departamento, p.nombre_proyecto
FROM empleados AS e
JOIN departamentos AS d ON e.departamento_id = d.id_departamento
JOIN proyectos AS p ON e.id_empleado = p.id_empleado;
```

Este **JOIN** combina las tres tablas basándose en las relaciones entre las columnas correspondientes (departamento_id, id_empleado) y devuelve el nombre del empleado, el nombre del departamento y el nombre del proyecto.

6. Consideraciones al Utilizar JOIN

Al utilizar **JOIN** en consultas, hay algunos aspectos a tener en cuenta:

1. **Integridad de las relaciones**:
    - Asegúrate de que las claves foráneas en tus tablas estén bien definidas y que las relaciones entre tablas sean correctas. Si las relaciones no están definidas correctamente, el JOIN no devolverá los resultados esperados.
2. **Uso de alias**:
    - Usar alias es útil para simplificar consultas, especialmente cuando se combinan varias tablas con columnas de nombres similares. Evita ambigüedades en las consultas utilizando alias para cada tabla.
3. **Condiciones de unión**:
    - La cláusula **ON** es crítica en los JOIN. Si las condiciones de unión son incorrectas o faltan, los resultados pueden ser erróneos o vacíos.
4. **Orden de las tablas**:
    - El orden de las tablas en la consulta también importa. Generalmente, la tabla que contiene los datos "principales" o "primarios" se coloca al principio y las tablas relacionadas se unen posteriormente.

Tipos de JOIN en SQL

En SQL, **JOIN** es una de las técnicas más poderosas para combinar datos de varias tablas. Existen diferentes tipos de **JOIN** que nos permiten combinar las tablas de distintas maneras, dependiendo de cómo queramos que se manejen las coincidencias entre las filas de las tablas involucradas. A continuación, exploraremos los principales tipos de **JOIN** que se utilizan en las bases de datos relacionales: **INNER JOIN**, **LEFT JOIN (o LEFT OUTER JOIN)**, **RIGHT JOIN (o RIGHT OUTER JOIN)**, **FULL JOIN (o FULL OUTER JOIN)**, y **CROSS JOIN**.

1. INNER JOIN (Unión Interna)

El **INNER JOIN** es el tipo de JOIN más común y se utiliza para devolver las filas que tienen coincidencias en ambas tablas. Cuando se realiza un **INNER JOIN**, solo se incluyen en los resultados aquellos registros que cumplen con la condición de unión entre las dos tablas.

Sintaxis:

SELECT columnas

FROM tabla1

INNER JOIN tabla2 ON tabla1.columna_relacionada = tabla2.columna_relacionada;

Ejemplo:

Considerando las tablas **empleados** y **departamentos**:

- **Empleados**:

ID_EMPLEADO	NOMBRE	DEPARTAMENTO_ID
1	Juan Pérez	1
2	Ana López	2
3	Luis Gómez	3

- **Departamentos**:

ID_DEPARTAMENTO	NOMBRE_DEPARTAMENTO
1	Recursos Humanos
2	IT

Si quieres obtener los empleados que están asignados a un departamento existente, puedes utilizar un **INNER JOIN**:

SELECT empleados.nombre, departamentos.nombre_departamento

FROM empleados

INNER JOIN departamentos ON empleados.departamento_id = departamentos.id_departamento;

Resultado:

NOMBRE	NOMBRE_DEPARTAMENTO
Juan Pérez	Recursos Humanos
Ana López	IT

En este caso, solo se devuelven los empleados que tienen un **departamento_id** correspondiente a un **id_departamento** en la tabla **departamentos**.

2. LEFT JOIN (o LEFT OUTER JOIN) (Unión Externa Izquierda)

El **LEFT JOIN** (también conocido como **LEFT OUTER JOIN**) devuelve **todas las filas de la tabla de la izquierda** (la primera tabla mencionada) y las filas coincidentes de la tabla de la derecha. Si no hay una coincidencia en la tabla de la derecha, el resultado incluirá **NULL** en las columnas de la tabla de la derecha.

Sintaxis:

SELECT columnas

FROM tabla1

LEFT JOIN tabla2 ON tabla1.columna_relacionada = tabla2.columna_relacionada;

Ejemplo:

Supongamos que quieres obtener todos los empleados, junto con los departamentos a los que están asignados. Si algún empleado no tiene asignado un departamento, todavía quieres que aparezca en el resultado, pero con los datos del departamento como **NULL**.

SELECT empleados.nombre, departamentos.nombre_departamento

FROM empleados

LEFT JOIN departamentos ON empleados.departamento_id = departamentos.id_departamento;

Resultado:

NOMBRE	NOMBRE_DEPARTAMENTO
Juan Pérez	Recursos Humanos
Ana López	IT
Luis Gómez	NULL

En este caso, **Luis Gómez** no tiene asignado un departamento, por lo que el campo **nombre_departamento** es **NULL**.

3. RIGHT JOIN (o RIGHT OUTER JOIN) (Unión Externa Derecha)

El **RIGHT JOIN** (también conocido como **RIGHT OUTER JOIN**) es similar al **LEFT JOIN**, pero devuelve **todas las filas de la tabla de la derecha** (la segunda tabla mencionada) y las filas coincidentes de la tabla de la izquierda. Si no hay una coincidencia en la tabla de la izquierda, el resultado incluirá **NULL** en las columnas de la tabla de la izquierda.

Sintaxis:

SELECT columnas

FROM tabla1

RIGHT JOIN tabla2 ON tabla1.columna_relacionada = tabla2.columna_relacionada;

Ejemplo:

Si quieres obtener todos los departamentos y los empleados asignados a cada uno, puedes usar un **RIGHT JOIN**. Si un departamento no tiene empleados asignados, aparecerá con **NULL** en la columna del empleado.

SELECT empleados.nombre, departamentos.nombre_departamento

FROM empleados

RIGHT JOIN departamentos ON empleados.departamento_id = departamentos.id_departamento;

Resultado:

NOMBRE	NOMBRE_DEPARTAMENTO
Juan Pérez	Recursos Humanos
Ana López	IT
NULL	Marketing

En este caso, **Marketing** es un departamento que no tiene empleados asignados, por lo que la columna **nombre** es **NULL**.

4. FULL JOIN (o FULL OUTER JOIN) (Unión Externa Completa)

El **FULL JOIN** (o **FULL OUTER JOIN**) devuelve **todas las filas de ambas tablas**, independientemente de si hay una coincidencia o no. Si no hay una coincidencia en alguna de las tablas, se rellena con **NULL** los campos correspondientes a la tabla que no tiene coincidencia.

Sintaxis:

SELECT columnas

FROM tabla1

FULL JOIN tabla2 ON tabla1.columna_relacionada = tabla2.columna_relacionada;

Ejemplo:

Si quieres obtener todos los empleados y todos los departamentos, incluso si no hay una coincidencia, puedes utilizar un **FULL JOIN**:

SELECT empleados.nombre, departamentos.nombre_departamento

FROM empleados

FULL JOIN departamentos ON empleados.departamento_id = departamentos.id_departamento;

Resultado:

NOMBRE	NOMBRE_DEPARTAMENTO
Juan Pérez	Recursos Humanos
Ana López	IT
Luis Gómez	NULL
NULL	Marketing

asignado, por lo que su campo de departamento es **NULL**, y **Marketing** es un departamento sin empleados asignados, por lo que su campo de empleado también es **NULL**.

5. CROSS JOIN (Unión Cruzada)

El **CROSS JOIN** genera un **producto cartesiano** entre las dos tablas. Esto significa que cada fila de la primera tabla se combina con **todas las filas de la segunda tabla**, sin necesidad de una condición de unión. Esto puede resultar en un gran número de filas si las tablas tienen muchas filas.

Sintaxis:

SELECT columnas

FROM tabla1

CROSS JOIN tabla2;

Ejemplo:

Si tienes dos tablas **empleados** y **departamentos** y quieres obtener todas las combinaciones posibles entre empleados y departamentos, puedes utilizar un **CROSS JOIN**:

SELECT empleados.nombre, departamentos.nombre_departamento

FROM empleados

CROSS JOIN departamentos;

Resultado:

NOMBRE	NOMBRE_DEPARTAMENTO
Juan Pérez	Recursos Humanos
Juan Pérez	IT
Ana López	Recursos Humanos
Ana López	IT
Luis Gómez	Recursos Humanos
Luis Gómez	IT

En este caso, el **CROSS JOIN** ha combinado todas las filas de **empleados** con todas las filas de **departamentos**, generando un total de 6 combinaciones.

Operaciones con Conjuntos en SQL

En SQL, las **operaciones con conjuntos** permiten combinar los resultados de múltiples consultas de una manera que simula operaciones matemáticas con conjuntos. Estas operaciones son fundamentales para la manipulación y la recuperación eficiente de datos en bases de datos relacionales.

Las principales operaciones con conjuntos que SQL soporta son:

- **UNION**
- **INTERSECT**
- **EXCEPT** (o **MINUS** en algunos sistemas de bases de datos)

Estas operaciones permiten combinar o comparar los resultados de múltiples consultas de una forma lógica, en función de los conjuntos de datos que se generen de cada consulta.

1. UNION (Unión de Conjuntos)

La operación **UNION** en SQL se utiliza para combinar los resultados de dos o más consultas **SELECT** en un solo conjunto de resultados. La particularidad de **UNION** es que elimina las filas duplicadas, es decir, solo se retornan filas únicas, aunque las consultas subyacentes puedan devolver resultados duplicados.

Sintaxis:

```
SELECT columnas
FROM tabla1
UNION
SELECT columnas
FROM tabla2;
```

- **SELECT columnas**: Especifica las columnas que se seleccionan de cada tabla.
- **tabla1** y **tabla2**: Son las tablas de las cuales se están seleccionando los datos.

Ejemplo:

Imagina que tienes las siguientes tablas de empleados:

- **empleados_espana**:

id_empleado	nombre	ciudad
1	Juan Pérez	Madrid
2	Ana López	Barcelona

- **empleados_mexico**:

id_empleado	nombre	ciudad
3	Luis Gómez	Ciudad de México
4	Carlos Ruiz	Monterrey

Si deseas obtener una lista combinada de todos los empleados de **España** y **México**, puedes utilizar **UNION**:

```
SELECT nombre, ciudad
FROM empleados_espana
UNION
```

```
SELECT nombre, ciudad
FROM empleados_mexico;
```

Resultado:

NOMBRE	CIUDAD
Juan Pérez	Madrid
Ana López	Barcelona
Luis Gómez	Ciudad de México
Carlos Ruiz	Monterrey

En este caso, la operación **UNION** combina los resultados de las dos tablas, pero elimina cualquier fila duplicada (si las hubiera).

2. UNION ALL (Unión sin Eliminar Duplicados)

A diferencia de **UNION**, la operación **UNION ALL** combina los resultados de dos consultas sin eliminar los duplicados. Si ambas consultas devuelven las mismas filas, esas filas aparecerán varias veces en el conjunto de resultados.

Sintaxis:

```
SELECT columnas
FROM tabla1
UNION ALL
SELECT columnas
FROM tabla2;
```

Ejemplo:

Siguiendo el mismo ejemplo con las tablas **empleados_espana** y **empleados_mexico**, si deseas combinar los resultados de ambas tablas sin eliminar los duplicados, utiliza **UNION ALL**:

```
SELECT nombre, ciudad
FROM empleados_espana
```

```
UNION ALL
SELECT nombre, ciudad
FROM empleados_mexico;
```

Resultado:

NOMBRE	CIUDAD
Juan Pérez	Madrd
Ana López	Barcelona
Luis Gómez	Ciudad de México
Carlos Ruiz	Monterrey

Si hubiera empleados con los mismos nombres y ciudades en ambas tablas, aparecerían varias veces en el conjunto de resultados.

3. INTERSECT (Intersección de Conjuntos)

La operación **INTERSECT** devuelve solo las filas que están presentes en ambas consultas **SELECT**. Es decir, solo se devuelven las filas que existen en ambos conjuntos de resultados, similar a la intersección de dos conjuntos en matemáticas.

Sintaxis:

```
SELECT columnas
FROM tabla1
INTERSECT
SELECT columnas
FROM tabla2;
```

Ejemplo:

Imagina que tienes dos tablas de empleados de dos departamentos diferentes, y deseas obtener una lista de empleados que están presentes en ambos departamentos:

- **Empleados_ventas**:

ID_EMPLEADO	NOMBRE	DEPARTAMENTO
1	Juan Pérez	Ventas
2	Ana López	Ventas

- **Empleados_marketing**:

ID_EMPLEADO	NOMBRE	DEPARTAMENTO
2	Ana López	Marketing
3	Luis Gómez	Marketing

Puedes utilizar **INTERSECT** para obtener los empleados que están tanto en **Ventas** como en **Marketing**:

SELECT nombre

FROM empleados_ventas

INTERSECT

SELECT nombre

FROM empleados_marketing;

Resultado:

NOMBRE
Ana López

En este caso, **Ana López** es la única que está presente en ambas tablas (en los departamentos de **Ventas** y **Marketing**).

4. EXCEPT (o MINUS en algunos sistemas)

La operación **EXCEPT** (también conocida como **MINUS** en algunos sistemas de bases de datos como Oracle) devuelve las filas de la primera consulta que **no están presentes** en la segunda consulta. En otras palabras, te da la diferencia entre dos conjuntos.

Sintaxis:

SELECT columnas

FROM tabla1

EXCEPT

SELECT columnas

FROM tabla2;

Ejemplo:

Supongamos que tienes las siguientes tablas de empleados y deseas obtener la lista de empleados de **Ventas** que no están en **Marketing**:

- **Empleados_ventas**:

ID_EMPLEADO	NOMBRE	DEPARTAMENTO
1	Juan Pérez	Ventas
2	Ana López	Ventas

- **empleados_marketing**:

id_empleado	nombre	departamento
2	Ana López	Marketing
3	Luis Gómez	Marketing

Puedes usar **EXCEPT** para encontrar empleados que están solo en **Ventas**:

SELECT nombre

FROM empleados_ventas

EXCEPT

SELECT nombre

FROM empleados_marketing;

Resultado:

nombre

Juan Pérez

En este caso, **Juan Pérez** es el único empleado de **Ventas** que no está en **Marketing**.

5. NOT IN y NOT EXISTS

Aunque no son estrictamente operaciones con conjuntos, las cláusulas **NOT IN** y **NOT EXISTS** pueden lograr resultados similares a **EXCEPT**. Estas cláusulas permiten excluir ciertos registros que cumplen con una condición específica.

Ejemplo usando NOT IN:

```
SELECT nombre
FROM empleados_ventas
WHERE nombre NOT IN (SELECT nombre FROM empleados_marketing);
```

Resultado:

nombre
Juan Pérez

Esta consulta es equivalente al ejemplo anterior con **EXCEPT** y excluye de la lista de **empleados_ventas** aquellos empleados que también están en **empleados_marketing**.

Agrupaciones y Funciones de Agregación en SQL

En SQL, las **agrupaciones** y las **funciones de agregación** son herramientas esenciales para realizar análisis y resúmenes de datos. Permiten combinar filas de una tabla en grupos y realizar cálculos agregados, como sumas, promedios, máximos, mínimos, entre otros. Esto es especialmente útil cuando trabajamos con grandes volúmenes de datos y necesitamos obtener información consolidada.

Las funciones de agregación permiten calcular valores resumidos sobre un conjunto de filas, y las **agrupaciones** agrupan las filas en función de una o más columnas antes de aplicar dichas funciones.

1. Agrupación de Datos (GROUP BY)

La cláusula **GROUP BY** se utiliza para agrupar las filas de una tabla en función de una o más columnas. Una vez que los datos están agrupados, se pueden aplicar funciones de agregación (como **COUNT**, **SUM**, **AVG**, etc.) a cada grupo.

Sintaxis Básica:

SELECT columna1, columna2, función_de_agregación(columna3)

FROM tabla

GROUP BY columna1, columna2;

- **columna1, columna2**: Son las columnas por las que quieres agrupar los resultados.
- **función_de_agregación(columna3)**: Es la función que aplicas a las filas de cada grupo (por ejemplo, **SUM**, **AVG**, **COUNT**, etc.).

Ejemplo de uso de GROUP BY:

Imagina que tienes una tabla llamada **ventas** con los siguientes datos:

- **ventas**:

id_venta	producto	cantidad	precio
1	Producto A	5	100
2	Producto B	3	200
3	Producto A	2	100
4	Producto C	1	300
5	Producto B	4	200

Si deseas obtener el **total de ventas por producto**, puedes utilizar la cláusula **GROUP BY** junto con una función de agregación:

SELECT producto, SUM(cantidad * precio) AS total_ventas

FROM ventas

GROUP BY producto;

Resultado:

producto total_ventas

Producto A 700

Producto B 1400

Producto C 300

En este caso, **GROUP BY** agrupa las ventas por el nombre del producto, y la función **SUM()** calcula el total de ventas para cada grupo.

2. Funciones de Agregación

Las **funciones de agregación** realizan cálculos sobre un conjunto de datos y devuelven un único valor resumido para cada grupo de registros. Las funciones más comunes son:

- **COUNT()**: Cuenta el número de filas o el número de 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.
- **MIN()**: Devuelve el valor mínimo de una columna.
- **MAX()**: Devuelve el valor máximo de una columna.

Ejemplo de uso de funciones de agregación:

Imagina que quieres obtener el **total de ventas**, el **promedio de ventas** y el **número de transacciones** para cada producto en la tabla **ventas**:

```
SELECT producto,
    COUNT(id_venta) AS num_ventas,
    SUM(cantidad * precio) AS total_ventas,
    AVG(cantidad * precio) AS promedio_venta,
    MIN(cantidad * precio) AS venta_minima,
    MAX(cantidad * precio) AS venta_maxima
FROM ventas
GROUP BY producto;
```

Resultado:

producto	num_ventas	total_ventas	promedio_venta	venta_minima	venta_maxima
Producto A	2	700	350	500	600
Producto B	2	1400	700	600	800
Producto C	1	300	300	300	300

En este ejemplo, se han utilizado varias funciones de agregación:

- **COUNT(id_venta)**: Cuenta el número de ventas por producto.
- **SUM(cantidad * precio)**: Suma el total de ventas por producto.

- **AVG(cantidad * precio)**: Calcula el promedio de ventas por producto.
- **MIN(cantidad * precio)** y **MAX(cantidad * precio)**: Devuelven el valor mínimo y máximo de ventas por producto.

3. Uso de HAVING con GROUP BY

La cláusula **HAVING** se utiliza para filtrar los resultados después de realizar la agrupación. Es similar a la cláusula **WHERE**, pero **WHERE** filtra las filas antes de agrupar, mientras que **HAVING** filtra los grupos después de aplicar la agregación.

Sintaxis:

SELECT columna1, columna2, función_de_agregación(columna3)

FROM tabla

GROUP BY columna1, columna2

HAVING condición;

Ejemplo con HAVING:

Si deseas obtener solo los productos cuyo total de ventas sea mayor a 1000, puedes usar **HAVING** para filtrar los resultados después de aplicar la agregación:

SELECT producto, SUM(cantidad * precio) AS total_ventas

FROM ventas

GROUP BY producto

HAVING SUM(cantidad * precio) > 1000;

Resultado:

producto	**total_ventas**
Producto B	1400

En este caso, **HAVING** filtra los grupos (productos) cuyo total de ventas es superior a 1000.

4. Agrupaciones por Varias Columnas

Se pueden agrupar los datos por varias columnas. Cuando se agrupan por múltiples columnas, los resultados serán más detallados, ya que cada combinación única de los valores de esas columnas formará un grupo.

Ejemplo de agrupación por varias columnas:

Si quieres conocer el total de ventas por **producto** y **ciudad**, puedes hacer una agrupación por ambas columnas:

SELECT producto, ciudad, SUM(cantidad * precio) AS total_ventas

FROM ventas

GROUP BY producto, ciudad;

Resultado:

roducto	**ciudad**	**total_ventas**
Producto A	Madrid	500
Producto A	Barcelona	200
Producto B	Ciudad de México	600
Producto B	Monterrey	800
Producto C	Madrid	300

Aquí, **GROUP BY producto, ciudad** agrupa las ventas por cada combinación única de **producto** y **ciudad**.

5. NULL y Agrupaciones

En SQL, los valores **NULL** se tratan de manera especial en las agrupaciones. Por defecto, las filas con valores **NULL** en las columnas de agrupación se consideran un grupo único. Esto puede ser útil cuando quieres manejar datos faltantes de forma explícita.

Ejemplo de agrupación con NULL:

Imagina que tienes una tabla de empleados con la siguiente estructura:

- **empleados**:

id_empleado	**nombre**	**departamento_id**
1	Juan Pérez	1
2	Ana López	2
3	Luis Gómez	NULL

Si deseas obtener el número de empleados por **departamento_id**, y también ver aquellos

empleados que no están asignados a ningún departamento (**NULL**), puedes hacerlo con una consulta de agrupación:

```
SELECT departamento_id, COUNT(id_empleado) AS num_empleados
FROM empleados
GROUP BY departamento_id;
```

Resultado:

departamento_id	num_empleados
1	1
2	1
NULL	1

Aquí, **NULL** en **departamento_id** forma su propio grupo, ya que no hay un valor asignado para ese campo en uno de los empleados.

Funciones Escalares en Consultas de Selección en SQL

En SQL, las **funciones escalares** son funciones que operan sobre un solo valor y devuelven un resultado único. Estas funciones son útiles para transformar o manipular datos de las columnas en las consultas de selección sin necesidad de agregaciones o agrupamientos. Las funciones escalares se aplican a los valores de una columna individualmente y se utilizan comúnmente en las cláusulas **SELECT**, **WHERE**, **ORDER BY**, etc.

Existen diferentes tipos de funciones escalares en SQL, que se pueden clasificar en varias categorías, tales como:

- **Funciones de cadena** (para manipulación de texto)
- **Funciones numéricas** (para operaciones aritméticas)
- **Funciones de fecha y hora** (para trabajar con fechas y horas)
- **Funciones de conversión** (para cambiar de tipo de datos)
- **Funciones de sistema** (para obtener información del sistema)

1. Funciones de Cadena

Las funciones de cadena permiten manipular y transformar cadenas de texto en SQL. Estas

funciones son especialmente útiles para trabajar con valores textuales.

Principales funciones de cadena:

- **CONCAT()**: Concatena dos o más cadenas de texto.
- **LENGTH()** (o **LEN()** en algunos sistemas): Devuelve la longitud de una cadena de texto.
- **UPPER()**: Convierte una cadena a mayúsculas.
- **LOWER()**: Convierte una cadena a minúsculas.
- **SUBSTRING()**: Extrae una subcadena de una cadena de texto.
- **TRIM()**: Elimina los espacios en blanco al principio y al final de una cadena.
- **REPLACE()**: Reemplaza todas las ocurrencias de un substring por otro.

Ejemplo de funciones de cadena:

Supón que tienes una tabla **clientes** con la siguiente estructura:

- **clientes**:

id_cliente	nombre	ciudad
1	Juan Pérez	Madrid
2	Ana López	Barcelona
3	Luis Gómez	Ciudad de México

Si deseas manipular los datos de texto, puedes utilizar funciones como las siguientes:

```
SELECT nombre,
    UPPER(nombre) AS nombre_mayusculas,
    LENGTH(nombre) AS longitud_nombre,
    CONCAT(nombre, ' de ', ciudad) AS descripcion_completa
FROM clientes;
```

Resultado:

nombre	nombre_mayusculas	longitud_nombre	descripcion_completa
Juan Pérez	JUAN PÉREZ	10	Juan Pérez de Madrid
Ana López	ANA LÓPEZ	9	Ana López de Barcelona
Luis Gómez	LUIS GÓMEZ	10	Luis Gómez de Ciudad de México

En este ejemplo:

- **UPPER(nombre)** convierte el nombre del cliente a mayúsculas.
- **LENGTH(nombre)** devuelve la longitud del nombre.
- **CONCAT(nombre, ' de ', ciudad)** concatena el nombre del cliente con su ciudad.

2. Funciones Numéricas

Las funciones numéricas permiten realizar operaciones matemáticas sobre los valores numéricos en una consulta. Algunas de las funciones numéricas más comunes son:

- **ROUND()**: Redondea un número a un número específico de decimales.
- **CEIL()** (o **CEILING()**): Redondea un número hacia arriba al siguiente entero.
- **FLOOR()**: Redondea un número hacia abajo al entero más cercano.
- **ABS()**: Devuelve el valor absoluto de un número.
- **MOD()**: Devuelve el residuo de una división.

Ejemplo de funciones numéricas:

Supón que tienes una tabla **productos** con la siguiente estructura:

- **productos**:

id_producto	nombre	precio
1	Producto A	99.99
2	Producto B	149.50
3	Producto C	120.20

Si deseas realizar operaciones numéricas sobre los precios de los productos, puedes utilizar funciones como:

```
SELECT nombre,
    ROUND(precio, 2) AS precio_redondeado,
    CEIL(precio) AS precio_ceil,
    FLOOR(precio) AS precio_floor,
    ABS(precio - 100) AS diferencia_con_100
FROM productos;
```

Resultado:

nombre	precio_redondeado	precio_ceil	precio_floor	diferencia_con_100
Producto A	99.99	100	99	0.01
Producto B	149.50	150	149	49.50
Producto C	120.20	121	120	20.20

En este caso:

- **ROUND(precio, 2)** redondea el precio a dos decimales.
- **CEIL(precio)** redondea el precio hacia arriba al siguiente número entero.
- **FLOOR(precio)** redondea el precio hacia abajo al número entero más cercano.
- **ABS(precio - 100)** calcula la diferencia absoluta entre el precio y 100.

3. Funciones de Fecha y Hora

Las funciones de fecha y hora en SQL son fundamentales cuando se trabaja con información temporal. Estas funciones permiten manipular, calcular y formatear fechas y horas de manera eficiente.

Principales funciones de fecha y hora:

- **NOW()** o **CURRENT_TIMESTAMP**: Devuelve la fecha y hora actuales.
- **CURDATE()**: Devuelve solo la fecha actual (sin la hora).
- **DATE()**: Extrae la parte de la fecha de una fecha completa.
- **YEAR()**: Devuelve el año de una fecha.
- **MONTH()**: Devuelve el mes de una fecha.
- **DAY()**: Devuelve el día de una fecha.
- **DATEADD()**: Suma un intervalo de tiempo a una fecha.
- **DATEDIFF()**: Calcula la diferencia entre dos fechas.

Ejemplo de funciones de fecha y hora:

Supongamos que tienes una tabla **pedidos** con la siguiente estructura:

- **pedidos**:

id_pedido	fecha_pedido	total
1	2024-11-01 10:00:00	200
2	2024-11-10 15:30:00	150
3	2024-11-20 08:45:00	250

Si deseas trabajar con las fechas de los pedidos, puedes utilizar funciones como las siguientes:

```
SELECT id_pedido,
    fecha_pedido,
    YEAR(fecha_pedido) AS anio,
    MONTH(fecha_pedido) AS mes,
    DAY(fecha_pedido) AS dia,
    DATEDIFF(NOW(), fecha_pedido) AS dias_transcurridos
FROM pedidos;
```

Resultado:

id_pedido	fecha_pedido	anio	mes	dia	dias_transcurridos
1	2024-11-01 10:00:00	2024	11	1	27
2	2024-11-10 15:30:00	2024	11	10	18
3	2024-11-20 08:45:00	2024	11	20	8

En este caso:

- **YEAR(fecha_pedido)** extrae el año de la fecha.
- **MONTH(fecha_pedido)** extrae el mes de la fecha.
- **DAY(fecha_pedido)** extrae el día de la fecha.
- **DATEDIFF(NOW(), fecha_pedido)** calcula la diferencia en días entre la fecha actual y la fecha del pedido.

4. Funciones de Conversión de Tipos de Datos

Las funciones de conversión permiten convertir valores de un tipo de datos a otro. Estas funciones son útiles cuando es necesario trabajar con datos de diferentes tipos y convertirlos para su correcto uso en las consultas.

Principales funciones de conversión:

- **CAST()**: Convierte un valor de un tipo de datos a otro tipo de datos.
- **CONVERT()**: Similar a **CAST()**, pero con una sintaxis diferente (utilizada especialmente en SQL Server).
- **TO_CHAR()**: Convierte un valor numérico o de fecha a una cadena de texto (en algunos sistemas como Oracle).
- **TO_DATE()**: Convierte una cadena de texto a una fecha (en algunos sistemas como Oracle).

Inserción de Datos en SQL

La **inserción de datos** es uno de los procesos fundamentales en la gestión de bases de datos. En SQL, la **instrucción INSERT** permite agregar nuevos registros (filas) a las tablas de una base de datos. A través de esta operación, podemos introducir datos de forma manual o automática, tanto para un solo registro como para múltiples registros en una sola operación.

Este capítulo explorará cómo insertar datos en una base de datos utilizando la instrucción **INSERT**, incluyendo la inserción de datos de una sola fila, múltiples filas y desde otra tabla.

1. Sintaxis Básica de INSERT

La sintaxis básica de la instrucción **INSERT** es la siguiente:

INSERT INTO nombre_de_tabla (columna1, columna2, columna3, ...)

VALUES (valor1, valor2, valor3, ...);

- **nombre_de_tabla**: Es el nombre de la tabla en la que se desean insertar los datos.
- **columna1, columna2, columna3, ...**: Son los nombres de las columnas en la tabla en las que se insertarán los valores.
- **valor1, valor2, valor3, ...**: Son los valores correspondientes que se insertarán en las columnas respectivas.

Ejemplo de inserción simple:

Imagina que tienes una tabla **clientes** con la siguiente estructura:

- **clientes**:

id_cliente	nombre	ciudad
1	Juan Pérez	Madrid

id_cliente	nombre	ciudad
2	Ana López	Barcelona

Para insertar un nuevo cliente, puedes utilizar la siguiente consulta SQL:

INSERT INTO clientes (id_cliente, nombre, ciudad)

VALUES (3, 'Luis Gómez', 'Valencia');

Esto agregará una nueva fila a la tabla **clientes**:

id_cliente	nombre	ciudad
1	Juan Pérez	Madrid
2	Ana López	Barcelona
3	Luis Gómez	Valencia

2. Inserción sin Especificar Columnas

Si deseas insertar un nuevo registro y proporcionar un valor para todas las columnas en orden secuencial, no es necesario especificar los nombres de las columnas, pero debes asegurarte de que los valores se correspondan con el orden de las columnas en la tabla.

Sintaxis sin columnas explícitas:

INSERT INTO nombre_de_tabla

VALUES (valor1, valor2, valor3, ...);

Ejemplo de inserción sin especificar columnas:

Usando la misma tabla **clientes**, si las columnas se insertan en el mismo orden que están definidas en la tabla, puedes hacer:

INSERT INTO clientes

VALUES (4, 'María Fernández', 'Sevilla');

Este comando insertará la siguiente fila en la tabla:

id_cliente	nombre	ciudad
1	Juan Pérez	Madrid
2	Ana López	Barcelona

id_cliente	nombre	ciudad
3	Luis Gómez	Valencia
4	María Fernández	Sevilla

Nota: Si la tabla tiene una columna con valores generados automáticamente (como una columna de clave primaria con **AUTO_INCREMENT** o **SERIAL**), puedes omitir esa columna en la declaración **INSERT**. La base de datos asignará automáticamente el valor a dicha columna.

3. Inserción de Múltiples Filas

Puedes insertar múltiples filas en una sola sentencia **INSERT**. Esto es útil cuando deseas agregar varios registros a la vez sin tener que escribir múltiples instrucciones **INSERT**.

Sintaxis para múltiples filas:

```
INSERT INTO nombre_de_tabla (columna1, columna2, columna3, ...)
VALUES
  (valor1a, valor2a, valor3a, ...),
  (valor1b, valor2b, valor3b, ...),
  (valor1c, valor2c, valor3c, ...);
```

Ejemplo de inserción de múltiples filas:

Para agregar varios clientes a la tabla **clientes**:

```
INSERT INTO clientes (id_cliente, nombre, ciudad)
VALUES
  (5, 'Carlos Ruiz', 'Madrid'),
  (6, 'Patricia Martín', 'Barcelona'),
  (7, 'Antonio García', 'Sevilla');
```

Esto agregará tres filas de datos:

id_cliente	nombre	ciudad
1	Juan Pérez	Madrid
2	Ana López	Barcelona
3	Luis Gómez	Valencia

id_cliente	nombre	ciudad
4	María Fernández	Sevilla
5	Carlos Ruiz	Madrid
6	Patricia Martín	Barcelona
7	Antonio García	Sevilla

4. Inserción de Datos desde Otras Tablas (INSERT SELECT)

En lugar de insertar datos manualmente, también puedes insertar registros en una tabla a partir de los resultados de una **consulta SELECT** de otra tabla. Este tipo de inserción es útil cuando deseas copiar datos de una tabla a otra.

Sintaxis de inserción desde una consulta SELECT:

INSERT INTO tabla_destino (columna1, columna2, columna3, ...)

SELECT columna1, columna2, columna3, ...

FROM tabla_origen

WHERE condición;

- **tabla_destino**: Es la tabla en la que deseas insertar los datos.
- **tabla_origen**: Es la tabla desde la que se seleccionarán los datos.
- **condición**: Es una condición opcional que limita los registros que serán insertados.

Ejemplo de inserción usando SELECT:

Supón que tienes una tabla **empleados** y una tabla **nuevos_empleados**. Si deseas copiar los registros de **nuevos_empleados** a la tabla **empleados**, puedes hacerlo con la siguiente consulta:

INSERT INTO empleados (id_empleado, nombre, puesto)

SELECT id_empleado, nombre, puesto

FROM nuevos_empleados

WHERE puesto = 'Desarrollador';

Este comando selecciona los empleados que son **Desarrolladores** de la tabla **nuevos_empleados** y los inserta en la tabla **empleados**.

5. Inserción con Subconsultas

Otra forma de insertar datos en una tabla es utilizando una subconsulta, donde los datos que se van a insertar provienen de una consulta anidada.

Sintaxis de inserción con subconsulta:

INSERT INTO tabla_destino (columna1, columna2, columna3, ...)

SELECT (subconsulta);

Ejemplo de inserción con subconsulta:

Supón que tienes una tabla **ventas** y otra tabla **productos**. Si deseas insertar en **ventas** un registro con el **producto** más caro disponible, puedes usar una subconsulta para obtener el precio del producto:

INSERT INTO ventas (producto_id, cantidad, total)

SELECT producto_id, 1, precio

FROM productos

WHERE precio = (SELECT MAX(precio) FROM productos);

En este caso, la subconsulta **SELECT MAX(precio)** obtiene el precio más alto de la tabla **productos** y lo utiliza para insertar un nuevo registro en **ventas**.

6. Comprobaciones y Restricciones al Insertar Datos

Cuando insertas datos, debes tener en cuenta las restricciones y reglas definidas en la tabla, tales como:

- **Restricciones de clave primaria**: No puedes insertar un valor duplicado en una columna definida como **clave primaria**.
- **Restricciones de unicidad (UNIQUE)**: No puedes insertar valores duplicados en columnas definidas con una restricción **UNIQUE**.
- **Restricciones de **NOT NULL**: Si una columna tiene la restricción **NOT NULL**, no puedes insertar valores nulos en esa columna.
- **Restricciones de integridad referencial (FOREIGN KEY)**: Si una columna está definida como clave foránea, el valor insertado debe existir en la tabla relacionada.

Ejemplo con restricción de clave primaria:

Si intentas insertar un valor duplicado en una columna de clave primaria, como **id_cliente** en la tabla

clientes, la base de datos generará un error.

```
INSERT INTO clientes (id_cliente, nombre, ciudad)
VALUES (1, 'Miguel Sánchez', 'Madrid');
```

Este intento fallará porque el **id_cliente** 1 ya existe en la tabla.

Actualización de Datos en SQL

En SQL, la instrucción **UPDATE** se utiliza para modificar los datos existentes en una tabla. Esta operación permite modificar una o más filas en función de una condición especificada, y es esencial para mantener la base de datos actualizada y coherente con los cambios en el mundo real.

Este capítulo cubre cómo realizar actualizaciones de datos con la instrucción **UPDATE**, detallando su sintaxis, cómo utilizar condiciones para especificar qué registros se deben actualizar, y algunas mejores prácticas al realizar este tipo de operaciones.

1. Sintaxis Básica de UPDATE

La sintaxis básica de la instrucción **UPDATE** es la siguiente:

```
UPDATE nombre_de_tabla
SET columna1 = valor1, columna2 = valor2, columna3 = valor3, ...
WHERE condición;
```

- **nombre_de_tabla**: Es el nombre de la tabla que contiene los datos que deseas actualizar.
- **columna1, columna2, columna3, ...**: Son los nombres de las columnas que deseas actualizar.
- **valor1, valor2, valor3, ...**: Son los nuevos valores que se asignarán a las columnas respectivas.
- **condición**: Especifica cuál o cuáles filas deben ser actualizadas. Si no se especifica una condición, se actualizarán todas las filas de la tabla.

Ejemplo de actualización básica:

Supongamos que tienes una tabla **clientes** con la siguiente estructura:

- **clientes**:

id_cliente	nombre	ciudad	edad
1	Juan Pérez	Madrid	30
2	Ana López	Barcelona	25
3	Luis Gómez	Valencia	28

Si deseas actualizar la ciudad de **Juan Pérez** a **"Sevilla"**, puedes usar el siguiente comando:

UPDATE clientes

SET ciudad = 'Sevilla'

WHERE id_cliente = 1;

Después de la actualización, la tabla **clientes** se verá así:

id_cliente	nombre	ciudad	edad
1	Juan Pérez	Sevilla	30
2	Ana López	Barcelona	25
3	Luis Gómez	Valencia	28

En este caso:

- **SET ciudad = 'Sevilla'**: Modifica la columna **ciudad** para el cliente con **id_cliente = 1**.
- **WHERE id_cliente = 1**: Especifica que solo se debe actualizar la fila donde el **id_cliente** sea igual a 1.

Nota: Siempre es recomendable utilizar una condición en el **WHERE** para evitar actualizar todas las filas de la tabla accidentalmente.

2. Actualización de Múltiples Columnas

Puedes actualizar más de una columna en la misma instrucción **UPDATE**. Para hacerlo, solo tienes que separar las asignaciones de columna y valor con comas.

Ejemplo de actualización de múltiples columnas:

Supón que deseas actualizar tanto la **ciudad** como la **edad** de **Ana López**:

UPDATE clientes

SET ciudad = 'Madrid', edad = 26

WHERE id_cliente = 2;

Después de esta actualización, la tabla **clientes** se verá así:

id_cliente	nombre	ciudad	edad
1	Juan Pérez	Sevilla	30
2	Ana López	Madrid	26
3	Luis Gómez	Valencia	28

Aquí:

- **SET ciudad = 'Madrid', edad = 26**: Actualiza tanto la columna **ciudad** como la columna **edad** de la fila correspondiente al **id_cliente = 2**.
- **WHERE id_cliente = 2**: Especifica que solo se debe actualizar la fila con **id_cliente = 2**.

3. Actualización sin Condición (WHERE)

Si omites la cláusula **WHERE** en una instrucción **UPDATE**, **todas** las filas de la tabla se verán afectadas, lo que puede ser peligroso si no se desea modificar todos los registros.

Ejemplo de actualización sin condición:

Si ejecutas la siguiente consulta:

UPDATE clientes

SET ciudad = 'Barcelona';

Resultado: Todas las filas de la tabla **clientes** tendrán su columna **ciudad** actualizada a **"Barcelona"**:

id_cliente	nombre	ciudad	edad
1	Juan Pérez	Barcelona	30
2	Ana López	Barcelona	26
3	Luis Gómez	Barcelona	28

Precaución: Siempre asegúrate de incluir una condición en la cláusula **WHERE** a menos que realmente desees actualizar todos los registros de la tabla. Si accidentalmente actualizas todas las filas, puede ser difícil revertir los cambios.

4. Actualización con Subconsultas

También puedes usar subconsultas en la cláusula **SET** o en la cláusula **WHERE** para actualizar datos en función de los resultados de otras consultas.

Ejemplo de actualización con subconsulta en SET:

Supón que tienes dos tablas: **productos** y **ventas**. La tabla **productos** tiene los precios de los productos, y la tabla **ventas** tiene las cantidades vendidas. Si deseas actualizar el **precio** de los productos en **productos** con el precio más bajo de **ventas**, puedes hacerlo con una subconsulta:

```
UPDATE productos
SET precio = (SELECT MIN(precio) FROM ventas)
WHERE id_producto = 1;
```

En este caso, la subconsulta **SELECT MIN(precio)** obtiene el precio más bajo de la tabla **ventas**, y esa información se utiliza para actualizar la columna **precio** de la tabla **productos**.

Ejemplo de actualización con subconsulta en WHERE:

También es posible usar una subconsulta en la cláusula **WHERE** para determinar qué filas deben ser actualizadas. Por ejemplo, supongamos que deseas actualizar el **precio** de los productos en la tabla **productos** solo si su **precio** es mayor que el **precio promedio** de los productos de **categoria = 'Electrónica'**. Esto se puede hacer con la siguiente consulta:

```
UPDATE productos
SET precio = precio * 0.9
WHERE precio > (SELECT AVG(precio) FROM productos WHERE categoria = 'Electrónica');
```

Este comando actualiza los precios de los productos de la tabla **productos**, aplicando un descuento del 10% a aquellos cuyo **precio** es mayor que el **precio promedio** de los productos electrónicos.

5. Actualización de Datos con Condiciones Complejas

Puedes combinar múltiples condiciones en la cláusula **WHERE** utilizando operadores lógicos como **AND**, **OR** y **NOT**. Esto te permite especificar criterios complejos para determinar qué filas deben ser actualizadas.

Ejemplo de actualización con condiciones complejas:

Supón que deseas actualizar la **edad** de los clientes solo si su **ciudad** es **"Madrid"** y su **edad** es menor de **30 años**:

```
UPDATE clientes
SET edad = 30
WHERE ciudad = 'Madrid' AND edad < 30;
```

Este comando actualizará la **edad** de todos los clientes en **Madrid** que sean menores de **30 años**.

6. Consideraciones y Buenas Prácticas

- **Usar la cláusula WHERE**: Asegúrate siempre de utilizar una cláusula **WHERE** para evitar actualizar todos los registros en la tabla accidentalmente. Las actualizaciones masivas pueden causar efectos no deseados.
- **Hacer un respaldo**: Siempre que vayas a realizar actualizaciones importantes en la base de datos, es recomendable hacer una copia de seguridad (backup) de los datos antes de realizar cualquier operación.
- **Verificar los datos antes de actualizar**: Antes de ejecutar una actualización, verifica qué registros se verán afectados utilizando una consulta **SELECT** con las mismas condiciones que usarás en el **UPDATE**. Por ejemplo:
- SELECT * FROM clientes WHERE ciudad = 'Madrid' AND edad < 30;

Esta consulta te mostrará las filas que serán actualizadas. De esta manera, puedes asegurarte de que la actualización afectará los registros correctos.

- **Evitar la actualización sin respaldo en entornos de producción**: En un entorno de producción, las actualizaciones masivas sin pruebas previas pueden resultar en pérdida de datos importantes.

Eliminación de Datos en SQL

La eliminación de datos en SQL es una operación crucial que permite eliminar registros de una tabla en una base de datos. Esta acción se realiza mediante la instrucción **DELETE**, que elimina filas específicas en función de una condición o elimina todos los registros de una tabla. Es importante tener mucho cuidado al usar esta operación, ya que los datos eliminados no pueden recuperarse sin una copia de seguridad adecuada.

Este capítulo cubrirá cómo utilizar la instrucción **DELETE** para eliminar registros de una base de datos, las mejores prácticas y precauciones para realizar esta operación de manera segura.

1. Sintaxis Básica de DELETE

La sintaxis básica de la instrucción **DELETE** es la siguiente:

```
DELETE FROM nombre_de_tabla
WHERE condición;
```

- **nombre_de_tabla**: Es el nombre de la tabla de la que deseas eliminar registros.
- **condición**: Especifica qué registros deben ser eliminados. Si omites esta condición, **todas las filas de la tabla** serán eliminadas.

Ejemplo de eliminación con condición:

Supón que tienes una tabla **clientes** con la siguiente estructura:

id_cliente	nombre	ciudad	edad
1	Juan Pérez	Madrid	30
2	Ana López	Barcelona	25
3	Luis Gómez	Valencia	28

Si deseas eliminar el cliente con **id_cliente = 2** (Ana López), puedes usar la siguiente consulta:

```
DELETE FROM clientes
WHERE id_cliente = 2;
```

Después de la eliminación, la tabla **clientes** se verá así:

id_cliente	nombre	ciudad	edad
1	Juan Pérez	Madrid	30
3	Luis Gómez	Valencia	28

En este caso:

- **WHERE id_cliente = 2**: Esta condición especifica que solo la fila con **id_cliente = 2** será eliminada.

2. Eliminación de Todos los Registros sin Condición

Si deseas eliminar **todos los registros** de una tabla, puedes omitir la cláusula **WHERE**. **Cuidado**: Esta operación eliminará todos los datos en la tabla y no podrá revertirse sin una copia de seguridad.

Sintaxis de eliminación sin condición:

DELETE FROM nombre_de_tabla;

Ejemplo de eliminación de todos los registros:

Si ejecutas la siguiente consulta:

DELETE FROM clientes;

Resultado: La tabla **clientes** quedará vacía:

id_cliente nombre ciudad edad

Precaución: Esta operación no elimina la estructura de la tabla, solo los datos. Si deseas eliminar tanto la tabla como los datos, puedes usar la instrucción **DROP TABLE**.

3. Eliminación de Registros con Subconsulta

Puedes usar una subconsulta en la cláusula **WHERE** para eliminar registros basados en los resultados de otra consulta. Esto es útil cuando quieres eliminar filas que cumplen con criterios complejos o que dependen de los datos de otras tablas.

Sintaxis de eliminación con subconsulta:

DELETE FROM nombre_de_tabla

WHERE columna IN (subconsulta);

Ejemplo de eliminación con subconsulta:

Supón que tienes dos tablas: **productos** y **ventas**. Si deseas eliminar todos los productos que no han sido vendidos, puedes usar una subconsulta para identificar los productos sin ventas:

DELETE FROM productos

WHERE id_producto NOT IN (SELECT id_producto FROM ventas);

Esta consulta elimina todos los productos de la tabla **productos** que no tienen registros correspondientes en la tabla **ventas**.

4. Eliminación con Condiciones Complejas

Puedes combinar múltiples condiciones en la cláusula **WHERE** utilizando operadores lógicos como **AND**, **OR** y **NOT** para especificar qué filas se deben eliminar. Esto te permite realizar eliminaciones más específicas y ajustadas a tus necesidades.

Ejemplo de eliminación con condiciones complejas:

Si deseas eliminar a todos los clientes que sean de **Madrid** y tengan una **edad mayor de 30 años**,

puedes hacer lo siguiente:

```
DELETE FROM clientes
WHERE ciudad = 'Madrid' AND edad > 30;
```

Este comando eliminará los registros de los clientes en **Madrid** cuya **edad** sea mayor de **30 años**.

5. Eliminación de Registros con JOIN

A veces, la eliminación de registros implica utilizar datos de varias tablas relacionadas. SQL permite realizar eliminaciones utilizando **JOINs** para combinar varias tablas y eliminar registros basados en condiciones que involucren a múltiples tablas.

Sintaxis de eliminación con JOIN:

```
DELETE t1
FROM tabla1 t1
JOIN tabla2 t2 ON t1.columna = t2.columna
WHERE condición;
```

Ejemplo de eliminación con JOIN:

Supón que tienes las tablas **clientes** y **ventas**, y deseas eliminar los **clientes** que hayan realizado una compra en el último mes. Puedes hacerlo con una consulta como esta:

```
DELETE clientes
FROM clientes c
JOIN ventas v ON c.id_cliente = v.id_cliente
WHERE v.fecha_venta > '2024-10-01';
```

Esta consulta elimina a los **clientes** que han realizado compras después del **1 de octubre de 2024**. El **JOIN** une las tablas **clientes** y **ventas**, y la condición en **WHERE** filtra las filas basadas en la fecha de la venta.

6. Comprobaciones y Restricciones al Eliminar Datos

Es importante tener en cuenta las restricciones y reglas definidas en la base de datos cuando realizas eliminaciones de registros:

- **Restricciones de clave foránea (FOREIGN KEY)**: Si una tabla tiene restricciones de **clave foránea**, no podrás eliminar registros de esa tabla si existen filas relacionadas en otras tablas.

Por ejemplo, no podrás eliminar un cliente si existen ventas relacionadas con ese cliente.

- **ON DELETE CASCADE**: Si se ha definido una regla de eliminación en cascada en la relación de clave foránea (es decir, **ON DELETE CASCADE**), la eliminación de un registro en una tabla puede eliminar automáticamente los registros relacionados en otras tablas.

Ejemplo de eliminación con ON DELETE CASCADE:

Supón que tienes dos tablas: **clientes** y **ventas**, y que la columna **id_cliente** en **ventas** es una clave foránea que hace referencia a **clientes**. Si la restricción **ON DELETE CASCADE** está activada, eliminar un cliente también eliminará automáticamente todas sus ventas.

```
DELETE FROM clientes
WHERE id_cliente = 1;
```

Este comando eliminará al cliente con **id_cliente = 1**, y todas las ventas correspondientes a este cliente también serán eliminadas de la tabla **ventas**.

7. Eliminación Lenta de Datos y Paginación

Cuando eliminas un gran número de registros, es recomendable hacerlo en pequeños lotes para evitar que el sistema se sobrecargue. Esto se puede hacer utilizando **paginación** en las consultas de eliminación.

Ejemplo de eliminación con paginación:

Para eliminar registros en lotes de 100 filas, puedes hacerlo en múltiples consultas:

```
DELETE FROM clientes
WHERE id_cliente IN (SELECT id_cliente FROM clientes LIMIT 100);
```

Este proceso se puede repetir hasta que se eliminen todos los registros deseados, de manera más controlada.

8. Consideraciones y Buenas Prácticas

- **Usar la cláusula WHERE**: Al igual que con **UPDATE**, es crucial usar una condición **WHERE** en la instrucción **DELETE** para evitar eliminar todos los registros de la tabla accidentalmente.
- **Verificar antes de eliminar**: Siempre es recomendable hacer una consulta **SELECT** con la misma condición que usarás en el **DELETE** para verificar qué registros se eliminarán. Por ejemplo:

- SELECT * FROM clientes WHERE ciudad = 'Madrid' AND edad > 30;

Esta consulta te permitirá ver los registros que serán eliminados antes de ejecutar la instrucción **DELETE**.

- **Copia de seguridad**: Asegúrate de realizar copias de seguridad de los datos antes de realizar eliminaciones masivas en bases de datos críticas.
- **Revisar restricciones de clave foránea**: Si hay claves foráneas con **ON DELETE CASCADE**, ten en cuenta que la eliminación de registros en la tabla principal puede afectar a otras tablas relacionadas.

Transacciones en SQL

Las transacciones en SQL son un conjunto de operaciones que se ejecutan como una unidad indivisible, es decir, todas las operaciones dentro de una transacción se completan con éxito o ninguna se completa. Las transacciones son fundamentales para mantener la integridad y la consistencia de los datos en una base de datos, especialmente en entornos donde se realizan múltiples modificaciones simultáneas.

Este capítulo cubre los aspectos básicos de las transacciones en SQL, incluyendo su definición, cómo utilizarlas, y los comandos asociados que permiten controlar las transacciones.

1. ¿Qué es una Transacción en SQL?

Una **transacción** es un conjunto de una o más operaciones que se ejecutan como una sola unidad lógica. Las transacciones aseguran que una serie de operaciones en la base de datos se lleven a cabo de manera completa y coherente. Si alguna operación dentro de la transacción falla, todas las operaciones previas se deshacen, garantizando la **consistencia** de los datos.

Las transacciones en SQL deben cumplir con las propiedades ACID:

- **Atomicidad**: La transacción es indivisible, es decir, todas las operaciones dentro de la transacción se ejecutan o ninguna se ejecuta.
- **Consistencia**: La base de datos pasa de un estado consistente a otro estado consistente después de la transacción.
- **Aislamiento**: Las operaciones de una transacción no son visibles para otras transacciones hasta que la transacción se complete.
- **Durabilidad**: Una vez que una transacción se ha completado, sus efectos son permanentes,

incluso en caso de fallos del sistema.

2. Comandos Básicos para Controlar las Transacciones

Existen varios comandos SQL que permiten gestionar las transacciones:

- **BEGIN TRANSACTION**: Inicia una nueva transacción.
- **COMMIT**: Guarda los cambios realizados en la base de datos durante la transacción.
- **ROLLBACK**: Deshace todos los cambios realizados durante la transacción.
- **SAVEPOINT**: Establece un punto de restauración dentro de una transacción.
- **RELEASE SAVEPOINT**: Elimina un punto de restauración previamente definido.
- **SET TRANSACTION**: Establece las propiedades de la transacción, como el nivel de aislamiento.

3. Sintaxis Básica de Transacciones

Iniciar una Transacción

Para iniciar una transacción en SQL, utilizas el comando **BEGIN TRANSACTION** o simplemente **BEGIN**, dependiendo del sistema de gestión de base de datos (DBMS) que estés utilizando:

BEGIN TRANSACTION;

Este comando inicia una nueva transacción en la que todas las operaciones que se realicen serán parte de esta unidad.

Confirmar una Transacción

Una vez que todas las operaciones dentro de la transacción se han completado correctamente, se utiliza **COMMIT** para guardar los cambios realizados:

COMMIT;

El comando **COMMIT** hace permanentes todos los cambios realizados durante la transacción.

Deshacer una Transacción

Si ocurre un error en cualquier operación dentro de la transacción o si se desea revertir todos los cambios realizados, se utiliza **ROLLBACK**:

ROLLBACK;

El comando **ROLLBACK** deshace todos los cambios realizados en la transacción desde su inicio, dejando la base de datos en su estado anterior.

4. Ejemplo de Transacción Básica

Supongamos que tienes una tabla **cuentas** en la que deseas transferir dinero de una cuenta a otra. Este tipo de operación debe ser tratado dentro de una transacción para asegurarse de que ambas modificaciones (la deducción de una cuenta y la adición a otra) se realicen correctamente o, en caso de error, se reviertan todas.

Ejemplo de transferencia de dinero:

```
BEGIN TRANSACTION;

-- Resta dinero de la cuenta de origen
UPDATE cuentas
SET saldo = saldo - 100
WHERE id_cuenta = 1;

-- Suma dinero a la cuenta de destino
UPDATE cuentas
SET saldo = saldo + 100
WHERE id_cuenta = 2;

-- Si todo es correcto, confirmamos la transacción
COMMIT;
```

Si todo va bien, la transacción se confirma con **COMMIT**, y ambos cambios se hacen permanentes.

Caso de error y rollback:

Supón que ocurre un error durante la ejecución de una de las actualizaciones. En ese caso, utilizamos **ROLLBACK** para deshacer la transacción completa:

```
BEGIN TRANSACTION;

-- Resta dinero de la cuenta de origen
UPDATE cuentas
```

```
SET saldo = saldo - 100
WHERE id_cuenta = 1;

-- Aquí, supongamos que la siguiente actualización falla
UPDATE cuentas
SET saldo = saldo + 100
WHERE id_cuenta = 2;

-- Si ocurre un error, deshacemos los cambios
ROLLBACK;
```

Con **ROLLBACK**, la base de datos vuelve a su estado original, como si ninguna de las actualizaciones hubiera ocurrido.

5. SAVEPOINT y ROLLBACK Parcial

Los **SAVEPOINT** permiten establecer puntos intermedios dentro de una transacción. Estos puntos pueden ser útiles si deseas deshacer parte de una transacción sin tener que deshacer toda la transacción.

Sintaxis de SAVEPOINT:

```
SAVEPOINT nombre_punto;
```

Sintaxis de ROLLBACK TO SAVEPOINT:

```
ROLLBACK TO SAVEPOINT nombre_punto;
```

Ejemplo con SAVEPOINT:

Imagina que estás realizando varias actualizaciones dentro de una transacción y deseas deshacer solo algunas de ellas, pero mantener otras:

```
BEGIN TRANSACTION;

-- Realizamos la primera operación
UPDATE cuentas
SET saldo = saldo - 100
```

```
WHERE id_cuenta = 1;

-- Establecemos un punto de restauración
SAVEPOINT punto1;

-- Realizamos la segunda operación
UPDATE cuentas
SET saldo = saldo + 100
WHERE id_cuenta = 2;

-- Supón que algo falla en la segunda operación, pero queremos mantener la primera
ROLLBACK TO SAVEPOINT punto1;

-- Confirmamos lo que ha quedado (solo la primera operación)
COMMIT;
```

En este ejemplo:

- La primera operación se mantiene.
- La segunda operación se deshace hasta el **SAVEPOINT**.
- Finalmente, **COMMIT** confirma los cambios realizados.

6. Niveles de Aislamiento de Transacciones

El **aislamiento** es uno de los aspectos clave de las transacciones en SQL. Define cómo las operaciones de una transacción se ven afectadas por otras transacciones que se ejecutan de manera concurrente. SQL soporta varios niveles de aislamiento, cada uno con un grado diferente de restricción sobre el acceso concurrente a los datos.

Los niveles de aislamiento más comunes son:

1. **Read Uncommitted**: Las transacciones pueden leer datos que no han sido confirmados aún (datos "sucios").
2. **Read Committed**: Las transacciones solo pueden leer datos que hayan sido confirmados.

3. **Repeatable Read**: Asegura que los datos leídos por una transacción no cambien durante su ejecución, pero puede haber **phantom reads** (lecturas fantasma).
4. **Serializable**: El nivel de aislamiento más estricto, donde las transacciones se ejecutan de manera secuencial y no hay interferencia entre ellas.

Sintaxis para establecer un nivel de aislamiento:

SET TRANSACTION ISOLATION LEVEL nivel;

Ejemplo de establecer el nivel de aislamiento:

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

Este comando establece que las transacciones se ejecutarán en un modo completamente aislado, garantizando que no haya interferencia entre ellas.

7. Comprobación de la Transacción

Algunos sistemas de gestión de bases de datos permiten comprobar el estado de una transacción para saber si se ha completado con éxito o si ha fallado. Esto depende del DBMS utilizado, pero en muchos casos, el manejo de errores a través de excepciones o códigos de retorno es suficiente para controlar el flujo de las transacciones.

8. Buenas Prácticas con Transacciones

- **Usa transacciones para operaciones críticas**: Las transacciones son esenciales para operaciones como transferencias de dinero, inserciones múltiples de registros relacionados, o cuando hay varias actualizaciones en varias tablas.
- **Evita transacciones largas**: Mantén las transacciones tan cortas como sea posible para evitar bloqueos prolongados de registros y mejorar el rendimiento.
- **Comprueba los resultados antes de usar COMMIT**: Verifica que todas las operaciones dentro de la transacción se hayan ejecutado correctamente antes de confirmar la transacción con **COMMIT**.
- **Manejo adecuado de errores**: Si alguna operación falla, utiliza **ROLLBACK** para deshacer la transacción y evitar dejar la base de datos en un estado inconsistente.
- **Uso de SAVEPOINT para control parcial**: Utiliza **SAVEPOINT** para permitir la reversión de operaciones específicas dentro de una transacción, sin afectar a todo el proceso.

Glosario de Términos SQL

A continuación se presenta un glosario de los términos más comunes y relevantes en SQL, que te ayudará a comprender mejor los conceptos utilizados a lo largo de este manual y a desarrollar una mayor familiaridad con el lenguaje SQL.

1. **ACID

Un conjunto de propiedades que garantizan que las transacciones en una base de datos se ejecuten de manera fiable: Atomicidad, Consistencia, Aislamiento y Durabilidad.

2. Agregación

El proceso de combinar múltiples filas de datos en un único valor. Las funciones de agregación incluyen SUM, AVG, COUNT, MAX, y MIN.

3. ALTER

Un comando SQL utilizado para modificar la estructura de una tabla existente, como agregar o eliminar columnas o cambiar los tipos de datos.

4. AND

Un operador lógico en SQL que se utiliza para combinar dos o más condiciones en una cláusula WHERE. Todas las condiciones deben ser verdaderas para que se cumpla la condición global.

5. API (Interfaz de Programación de Aplicaciones)

Un conjunto de herramientas y protocolos que permiten que diferentes aplicaciones interactúen con bases de datos o sistemas de software.

6. BASE DE DATOS

Un sistema organizado de datos que permite el almacenamiento, la recuperación y la manipulación de datos en un formato estructurado.

7. BETWEEN

Un operador en SQL que se utiliza para filtrar datos dentro de un rango determinado, como fechas, números o cadenas de texto.

8. CLAVE FORÁNEA

Una columna o conjunto de columnas en una tabla que se refiere a la clave primaria de otra tabla, creando una relación entre ambas tablas.

9. CLAVE PRIMARIA

Una columna o conjunto de columnas que identifican de manera única cada fila en una tabla. No

puede haber valores nulos en la clave primaria.

10. COMMIT

Un comando SQL utilizado para confirmar una transacción y hacer que los cambios sean permanentes en la base de datos.

11. CONCATENAR

La operación de combinar dos o más cadenas de texto en una sola. En SQL se puede lograr usando la función CONCAT.

12. CONSTRAINT

Una restricción en SQL que se aplica a una columna o conjunto de columnas en una tabla para asegurar la integridad de los datos (por ejemplo, PRIMARY KEY, FOREIGN KEY, UNIQUE, etc.).

13. COUNT

Una función de agregación que devuelve el número de filas que cumplen con una condición específica.

14. CREATE TABLE

Un comando SQL utilizado para crear una nueva tabla en una base de datos, especificando su estructura (columnas, tipos de datos, restricciones, etc.).

15. DATABASE MANAGEMENT SYSTEM (DBMS)

Un software que gestiona la creación, mantenimiento y manipulación de bases de datos. Ejemplos incluyen MySQL, PostgreSQL, y SQL Server.

16. DELETE

Un comando SQL utilizado para eliminar una o más filas de una tabla, según una condición especificada en la cláusula WHERE.

17. DISTINCT

Un operador SQL utilizado en una consulta SELECT para devolver solo valores únicos de una columna, eliminando duplicados.

18. DROP

Un comando SQL utilizado para eliminar completamente una tabla, vista u objeto de la base de datos.

19. INNER JOIN

Un tipo de JOIN que devuelve las filas de dos o más tablas que tienen coincidencias basadas en una condición común.

20. INSERT INTO

Un comando SQL utilizado para agregar una nueva fila de datos en una tabla existente.

21. IS NULL

Un operador SQL utilizado para verificar si una columna contiene un valor nulo (NULL).

22. JOIN

Un tipo de operación en SQL que combina filas de dos o más tablas basadas en una condición de coincidencia entre ellas.

23. LEFT JOIN

Un tipo de JOIN que devuelve todas las filas de la tabla de la izquierda y las filas coincidentes de la tabla de la derecha. Si no hay coincidencia, se llenan con valores nulos.

24. LIKE

Un operador en SQL utilizado en una cláusula WHERE para buscar un patrón dentro de una columna de texto. Se usa con los caracteres comodín % (cualquier número de caracteres) y _ (un solo carácter).

25. LIMIT

Un comando en SQL utilizado para restringir el número de filas devueltas por una consulta. Comúnmente utilizado en bases de datos como MySQL.

26. NOT

Un operador lógico utilizado para invertir una condición, de modo que si la condición es verdadera, se convierte en falsa, y viceversa.

27. NULL

Un valor especial en SQL que representa la ausencia de un valor. Se utiliza para indicar que un campo no tiene un valor asignado.

28. ORDER BY

Un comando SQL utilizado para ordenar los resultados de una consulta SELECT en orden ascendente (ASC) o descendente (DESC).

29. PRIMARY KEY

Una restricción que garantiza que los valores en una columna o conjunto de columnas sean únicos y no nulos. Es utilizada para identificar de forma única cada fila en una tabla.

30. ROLLBACK

Un comando SQL utilizado para deshacer una transacción, restaurando la base de datos a su estado

anterior a la transacción.

31. SELECT

El comando SQL más utilizado para seleccionar datos de una o más tablas. Permite recuperar información de las bases de datos según ciertas condiciones.

32. SELECT DISTINCT

Una variante del comando SELECT que devuelve solo los valores únicos de una columna, eliminando los duplicados.

33. SET

Un comando SQL utilizado para cambiar las propiedades de una transacción o para definir el valor de una variable.

34. SQL

Un lenguaje de programación estándar utilizado para gestionar y manipular bases de datos relacionales. Se utiliza para realizar consultas, insertar, actualizar y eliminar datos, así como para crear y modificar estructuras de bases de datos.

35. UNION

Un operador en SQL que se utiliza para combinar los resultados de dos o más consultas SELECT. El operador UNION elimina los duplicados por defecto, mientras que UNION ALL incluye los duplicados.

36. UPDATE

Un comando SQL utilizado para modificar los valores de una o más columnas en filas existentes de una tabla.

37. WHERE

Una cláusula utilizada en las consultas SQL para especificar una condición de filtro que debe cumplirse para seleccionar, actualizar o eliminar filas en una tabla.

38. WITH

Una cláusula en SQL utilizada para definir una **subconsulta común** que puede ser referenciada múltiples veces dentro de una consulta principal. Utilizada comúnmente para mejorar la legibilidad de las consultas complejas.

39. XOR

Un operador lógico en SQL que devuelve verdadero si una de las condiciones es verdadera, pero no ambas.

Documentación Oficial de DBMS

1. **MySQL Documentation**
    - Sitio web oficial de MySQL, que proporciona documentación detallada sobre cómo usar SQL con este sistema de gestión de bases de datos.
    - MySQL Documentation
2. **Microsoft SQL Server Documentation**
    - Página oficial de la documentación de SQL Server, que incluye guías sobre el uso de T-SQL y las mejores prácticas de programación en SQL Server.
    - SQL Server Documentation

Sitios Web de Referencia

1. **SQLBolt**
    - Un sitio interactivo para aprender SQL paso a paso. Ofrece explicaciones claras y ejercicios prácticos con un enfoque en la práctica.
    - SQLBolt
2. **GeeksforGeeks SQL Tutorial**
    - GeeksforGeeks es un sitio web educativo que ofrece tutoriales SQL desde un nivel básico hasta más avanzado, con explicaciones claras y ejemplos prácticos.
    - GeeksforGeeks SQL

Foros y Comunidades

1. **Stack Overflow**
    - Un foro en línea donde puedes hacer preguntas y encontrar respuestas sobre SQL y otros lenguajes de programación. Es una excelente plataforma para resolver dudas prácticas.
    - Stack Overflow - SQL
2. **SQLServerCentral**
    - Una comunidad de usuarios de SQL Server, donde puedes acceder a artículos, foros y

recursos educativos sobre SQL.

- SQLServerCentral

3. **Reddit - r/SQL**
    - Un subforo dedicado a SQL, donde los usuarios comparten recursos, discuten problemas y ofrecen soluciones.
    - Reddit - r/SQL

Conclusión

El lenguaje SQL (Structured Query Language) es una herramienta fundamental en el ámbito de las bases de datos. Su capacidad para almacenar, manipular y recuperar datos de manera eficiente lo convierte en un pilar esencial en la gestión de información, especialmente en el contexto de bases de datos relacionales. A lo largo de este manual, hemos explorado desde los conceptos más básicos hasta las técnicas avanzadas de SQL, cubriendo todo lo necesario para trabajar con bases de datos de manera efectiva.

Lo aprendido:

- Iniciamos con los fundamentos de SQL, comprendiendo su propósito y cómo se usa para interactuar con bases de datos.
- Se realizaron consultas simples, con múltiples tablas mediante operaciones JOIN, y ejecutaron funciones agregadas que simplifican la manipulación de grandes volúmenes de datos.
- Entendimos cómo manejar transacciones, asegurar la integridad de los datos y optimizar consultas para mejorar el rendimiento.
- Aprendimos de las operaciones de inserción, actualización y eliminación de datos, y cómo asegurar la correcta gestión de información a lo largo del ciclo de vida de la base de datos.

Además, exploramos cómo utilizar SQL para manejar datos de manera lógica y estructurada, lo que es clave para la resolución de problemas en proyectos reales, como la creación de informes, análisis de datos, o incluso el desarrollo de aplicaciones complejas que dependen de bases de datos.

SQL no solo es un lenguaje de consulta, sino una habilidad esencial en el mundo moderno donde los datos son un activo valioso para las organizaciones. Aprender a utilizar SQL te abre muchas puertas en diversas áreas como la administración de bases de datos, análisis de datos, ciencia de datos, desarrollo de software, entre otros.