SQL (Structured Query Language) es un lenguaje de programación diseñado específicamente para gestionar y manipular bases de datos relacionales. Es utilizado para realizar diversas operaciones, como la consulta, inserción, actualización, eliminación y gestión de datos en sistemas de gestión de bases de datos (SGBD).

Principales características de SQL:

1. **Lenguaje declarativo**: SQL se centra en describir qué se quiere hacer con los datos, sin necesidad de detallar cómo debe realizarse la operación.
2. **Estándar**: Aunque hay variaciones específicas en diferentes sistemas de bases de datos (como SQL Server, MySQL, Oracle), SQL es un estándar que sigue una especificación definida por el ISO/ANSI.
3. **Relacional**: SQL está basado en el modelo relacional de bases de datos, donde los datos se organizan en tablas que pueden estar relacionadas entre sí a través de claves primarias y foráneas.

SQL no tiene, de manera nativa, **estructuras de control** como las que se encuentran en lenguajes de programación tradicionales, como bucles (`for`, `while`) o condicionales (`if-else`). Sin embargo, algunos sistemas de bases de datos incluyen **"extensiones procedurales"** que agregan estas características al lenguaje SQL, permitiendo realizar tareas más avanzadas como control de flujo y manejo de variables.

¿Qué son las extensiones procedurales?

Son **extensiones** de SQL implementadas por distintos proveedores de bases de datos que permiten utilizar estructuras de control (como bucles, condicionales, procedimientos almacenados, funciones, etc.). Estas extensiones permiten que SQL pueda ser utilizado no solo para manipular datos, sino también para crear **procedimientos más complejos** y reutilizables, proporcionando una lógica similar a la de los lenguajes de programación imperativos.

Ejemplos de extensiones procedurales en diferentes productos:

- **PL/SQL (Procedural Language/SQL)**: Utilizado en Oracle, permite el uso de estructuras de control como `IF`, `LOOP`, manejo de excepciones, y más.
- **T-SQL (Transact-SQL)**: Utilizado en Microsoft SQL Server y Sybase, incluye características procedurales como la declaración de variables, estructuras de control de flujo (`BEGIN...END`, `WHILE`, `IF...ELSE`), y manejo de transacciones.
- **PL/pgSQL (Procedural Language/PostgreSQL)**: La extensión procedural en PostgreSQL, similar a PL/SQL de Oracle, que permite crear funciones, procedimientos almacenados, y control de flujo.
- **SQL/PSM (Persistent Stored Modules)**: Un estándar de SQL para módulos almacenados, implementado por algunos SGBD, que permite definir procedimientos y funciones.

Introducción a SQL: Historia, Evolución y su Importancia en los Sistemas de Bases de Datos

SQL (Structured Query Language) es el lenguaje estándar para interactuar con sistemas de bases de datos relacionales. A lo largo de las décadas, SQL se ha consolidado como una herramienta fundamental para la gestión de datos, debido a su simplicidad y robustez. Aunque su rol principal es la manipulación y definición de datos, su evolución a través del tiempo ha permitido que abarque múltiples funcionalidades, haciéndolo una pieza clave en el desarrollo de aplicaciones empresariales y tecnológicas.

Un Poco de Historia: SEQUEL y el Nacimiento de SQL

SQL tiene sus raíces en un lenguaje desarrollado por IBM en la década de 1970, llamado **SEQUEL** (Structured English Query Language). Este lenguaje fue diseñado para permitir la interacción con su modelo relacional experimental, que en ese entonces representaba un cambio radical en la forma en que se almacenaban y consultaban los datos.

SEQUEL fue el primer intento exitoso de crear un lenguaje fácil de usar para consultas de bases de datos que seguían el modelo relacional propuesto por **Edgar F. Codd** en 1970. Posteriormente, el nombre cambió a **SQL** debido a problemas de marca registrada, pero la esencia del lenguaje se mantuvo.

En 1986, el **ANSI** y el **ISO** establecieron a SQL como el estándar oficial para la gestión de bases de datos relacionales. Desde entonces, SQL ha sufrido varias revisiones y mejoras, siendo adoptado y adaptado por prácticamente todos los sistemas de gestión de bases de datos relacionales (SGBD), tales como **Oracle, MySQL, PostgreSQL** y **SQL Server**.

SQL-99: Un Salto Evolutivo

Uno de los hitos más importantes en la evolución de SQL fue la introducción del estándar **SQL-99** (también conocido como **SQL:1999**). Esta versión añadió nuevas capacidades que ampliaron el alcance de SQL más allá de las simples operaciones de consulta y manipulación de datos. Algunas de las características clave introducidas fueron:

- **Consultas recursivas**: Permitieron realizar consultas sobre estructuras jerárquicas o recursivas, como las relacionadas con la organización de empleados o productos en catálogos.
- **Tipos de datos definidos por el usuario (UDTs)**: Los usuarios podían definir tipos de datos personalizados, brindando más flexibilidad para representar datos complejos.
- **Disparadores (triggers)**: Los disparadores permiten automatizar acciones en la base de datos cuando ocurren eventos específicos, como inserciones o actualizaciones.

- **Funciones definidas por el usuario**: Se dio la posibilidad de crear funciones reutilizables dentro de SQL, lo que mejoró la capacidad de SQL para realizar operaciones personalizadas en los datos.
- **Orientación a objetos**: SQL-99 añadió características para el soporte de conceptos orientados a objetos, como la encapsulación y los métodos asociados a los datos.

Estas características hicieron que SQL se convirtiera en un lenguaje más poderoso y versátil, capaz de manejar desde las operaciones más simples hasta las más complejas dentro de un entorno empresarial.

SQL-99 (también conocido como **SQL:1999**) es una versión del estándar SQL publicada por **ISO** y **ANSI** en 1999. Fue una actualización importante del lenguaje SQL que introdujo varias características nuevas y mejoró aspectos previos del estándar. Esta versión ampliaba las capacidades de SQL al agregar soporte para nuevas funcionalidades que lo hacían más potente y flexible en el manejo de bases de datos.

Principales características de SQL-99:

1. **Tipos de datos extendidos**:
   - Se introdujeron tipos de datos nuevos como `BLOB` (Binary Large Object) y `CLOB` (Character Large Object) para almacenar grandes volúmenes de datos binarios y texto.
   - Tipos de datos **BOOLEAN** fueron también agregados para manejar valores lógicos `TRUE`, `FALSE`, y `UNKNOWN`.
2. **Expresiones regulares y funciones de cadena mejoradas**:
   - La versión SQL-99 incluyó soporte para expresiones regulares en la manipulación de cadenas de caracteres, permitiendo mayor flexibilidad para búsquedas y reemplazos complejos de texto.
3. **Soporte para procedimientos almacenados y funciones**:
   - Introducción de un soporte robusto para **procedimientos almacenados** (stored procedures) y **funciones definidas por el usuario**. Estas características permiten la creación de bloques de código reutilizables dentro del sistema de gestión de bases de datos.
   - Introducción de **control de flujo** (como `IF`, `WHILE`, y `LOOP`) dentro de las funciones y procedimientos, agregando capacidades procedurales a SQL.
4. **Consultas recursivas**:
   - Se agregó soporte para **consultas recursivas** a través de la cláusula `WITH RECURSIVE`. Esto permite realizar consultas que hacen referencia a sí mismas, facilitando el manejo de estructuras jerárquicas como árboles o grafos.

   Ejemplo de consulta recursiva:

```
WITH RECURSIVE descendientes AS (
  SELECT id, nombre
  FROM empleados
```

```
  WHERE jefe_id IS NULL
  UNION ALL
  SELECT e.id, e.nombre
  FROM empleados e
  JOIN descendientes d ON e.jefe_id = d.id
)
SELECT * FROM descendientes;
```

5. **Soporte para disparadores (triggers)**:
   - **Disparadores (triggers)** fueron introducidos, permitiendo ejecutar automáticamente una acción en respuesta a eventos en las tablas, como la inserción, actualización o eliminación de registros.
6. **Tipos de datos definidos por el usuario (UDTs)**:
   - SQL-99 introdujo la capacidad de definir **tipos de datos personalizados** por el usuario, lo que permitió una mayor flexibilidad en la representación y manipulación de datos complejos.
7. **Objetos y orientación a objetos**:
   - Aunque SQL sigue siendo mayoritariamente relacional, SQL-99 dio los primeros pasos hacia la **orientación a objetos**, permitiendo crear tablas con atributos que son objetos y métodos asociados.
8. **Soporte para OLAP (Procesamiento Analítico en Línea)**:
   - SQL-99 mejoró las capacidades de SQL para análisis de datos, introduciendo funciones como `ROLLUP`, `CUBE` y `GROUPING SETS`, que ayudan a realizar cálculos agregados y análisis multidimensionales.

Impacto de SQL-99:

SQL-99 fue una mejora significativa en el estándar, ampliando sus capacidades para soportar aplicaciones más complejas y con mayores requerimientos. Proveedores de bases de datos como Oracle, SQL Server, y PostgreSQL adoptaron varias de estas características en sus implementaciones, aunque no todos los sistemas de gestión de bases de datos las implementan completamente.

¿Qué son las extensiones procedurales?

Son **extensiones** de SQL implementadas por distintos proveedores de bases de datos que permiten utilizar estructuras de control (como bucles, condicionales, procedimientos almacenados, funciones, etc.). Estas extensiones permiten que SQL pueda ser utilizado no solo para manipular datos, sino también para crear **procedimientos más complejos** y reutilizables, proporcionando una lógica similar a la de los lenguajes de programación imperativos.

Ejemplos de extensiones procedurales en diferentes productos:

- **PL/SQL (Procedural Language/SQL)**: Utilizado en Oracle, permite el uso de estructuras de control como `IF`, `LOOP`, manejo de excepciones, y más.
- **T-SQL (Transact-SQL)**: Utilizado en Microsoft SQL Server y Sybase, incluye características procedurales como la declaración de variables, estructuras de control de flujo (`BEGIN...END`, `WHILE`, `IF...ELSE`), y manejo de transacciones.

- **PL/pgSQL (Procedural Language/PostgreSQL)**: La extensión procedural en PostgreSQL, similar a PL/SQL de Oracle, que permite crear funciones, procedimientos almacenados, y control de flujo.
- **SQL/PSM (Persistent Stored Modules)**: Un estándar de SQL para módulos almacenados, implementado por algunos SGBD, que permite definir procedimientos y funciones.

Ejemplo de Procedimiento en T-SQL:

Este es un ejemplo de cómo T-SQL (la extensión procedural en SQL Server) permite usar estructuras de control para crear procedimientos almacenados:

```
CREATE PROCEDURE aumentarSalario
    @porcentaje INT
AS
BEGIN
    IF @porcentaje > 0
    BEGIN
        UPDATE empleados
        SET salario = salario + (salario * @porcentaje / 100);
    END
    ELSE
    BEGIN
        PRINT 'El porcentaje debe ser mayor a 0';
    END
END;
```

En este caso, se utiliza un `IF` para verificar que el porcentaje sea mayor a cero antes de actualizar los salarios de los empleados.

Beneficios de las extensiones procedurales:

- **Automatización**: Permiten ejecutar procedimientos complejos en la base de datos sin intervención manual.
- **Reutilización**: Se pueden crear procedimientos y funciones reutilizables.
- **Mayor control**: Proporcionan un mayor control sobre el flujo de ejecución dentro de la base de datos.

Los **sublenguajes** de SQL son subconjuntos del lenguaje que están diseñados para realizar tareas específicas dentro de una base de datos. Cada uno tiene un propósito particular, como la manipulación de datos, la definición de estructuras de la base de datos o el control de acceso. A continuación te explico cada uno de los sublenguajes y su función dentro de SQL.

Principales Sublenguajes de SQL:

1. **DDL (Data Definition Language)**: **Lenguaje de definición de datos**

- **Propósito**: Define y modifica la estructura de las bases de datos, como crear o eliminar tablas, índices, vistas, entre otros objetos.
- **Comandos principales**:
  - **CREATE**: Crear objetos como tablas, vistas o bases de datos.
    - Ejemplo:

```
CREATE TABLE empleados (
  id INT PRIMARY KEY,
  nombre VARCHAR(50),
  salario DECIMAL(10, 2)
);
```

  - **ALTER**: Modificar la estructura de un objeto existente, como agregar o eliminar columnas en una tabla.
    - Ejemplo:

```
ALTER TABLE empleados ADD direccion
VARCHAR(100);
```

  - **DROP**: Eliminar objetos como tablas, vistas o bases de datos.
    - Ejemplo:

```
DROP TABLE empleados;
```

- **Utilidad**: DDL se usa principalmente cuando necesitas definir, estructurar o modificar la organización de tus datos en una base de datos.

2. **DML (Data Manipulation Language)**: **Lenguaje de manipulación de datos**
   - **Propósito**: Manipula los datos dentro de las tablas. Permite realizar operaciones como consultar, insertar, actualizar o eliminar registros.
   - **Comandos principales**:
     - **SELECT**: Consultar datos de una o más tablas.
       - Ejemplo:

```
SELECT nombre, salario FROM empleados WHERE
salario > 50000;
```

     - **INSERT**: Insertar nuevos registros en una tabla.
       - Ejemplo:

```
INSERT INTO empleados (id, nombre, salario)
VALUES (1, 'Juan Pérez', 50000);
```

     - **UPDATE**: Actualizar registros existentes en una tabla.
       - Ejemplo:

```
UPDATE empleados SET salario = 55000 WHERE id
= 1;
```

     - **DELETE**: Eliminar registros de una tabla.
       - Ejemplo:

```
DELETE FROM empleados WHERE id = 1;
```

- **Utilidad**: DML se usa para gestionar y manipular los datos que están almacenados en la base de datos, ya sea para leer información (SELECT) o modificarla (INSERT, UPDATE, DELETE).

3. **DCL (Data Control Language)**: **Lenguaje de control de datos**
   - **Propósito**: Controla el acceso a los datos y gestiona los permisos de los usuarios. Se utiliza para definir quién puede acceder o modificar los datos de la base de datos.
   - **Comandos principales**:
     - **GRANT**: Otorga permisos a los usuarios para realizar ciertas acciones (como leer o modificar datos).
       - Ejemplo:

```
GRANT SELECT ON empleados TO usuario1;
```

     - **REVOKE**: Revoca permisos previamente otorgados a los usuarios.
       - Ejemplo:

```
REVOKE SELECT ON empleados FROM usuario1;
```

   - **Utilidad**: DCL se usa para gestionar la seguridad de la base de datos y controlar quién puede acceder o modificar los datos en un entorno multiusuario.
4. **TCL (Transaction Control Language)**: **Lenguaje de control de transacciones**
   - **Propósito**: Gestiona las transacciones dentro de una base de datos, permitiendo que las operaciones se confirmen (commit) o se reviertan (rollback) en bloque.
   - **Comandos principales**:
     - **COMMIT**: Confirma una transacción, haciendo permanentes los cambios realizados en la base de datos.
       - Ejemplo:

```
COMMIT;
```

     - **ROLLBACK**: Deshace los cambios realizados durante una transacción, devolviendo la base de datos a su estado anterior.
       - Ejemplo:

```
ROLLBACK;
```

     - **SAVEPOINT**: Crea puntos intermedios dentro de una transacción, permitiendo deshacer parte de los cambios realizados sin afectar toda la transacción.
       - Ejemplo:

```
SAVEPOINT punto_guardado;
```

- **Utilidad**: TCL es esencial para mantener la **integridad de los datos**, especialmente en sistemas donde varias operaciones necesitan completarse en conjunto. Si una parte de la operación falla, TCL permite deshacerla sin afectar el resto de la base de datos.

¿Por qué son importantes los sublenguajes en SQL?

Cada sublenguaje de SQL está diseñado para cumplir una función específica, permitiendo a los administradores y desarrolladores de bases de datos gestionar, manipular y controlar el acceso a los datos de manera eficiente. En conjunto, proporcionan una **estructura completa** para interactuar con una base de datos, asegurando que tanto la estructura de los datos como las operaciones con ellos se realicen de manera controlada, eficiente y segura.

Al dominar estos sublenguajes, un profesional de bases de datos puede no solo gestionar grandes volúmenes de información, sino también garantizar la **seguridad, integridad y disponibilidad de los datos** dentro de un entorno de múltiples usuarios y transacciones.

Este enfoque modular permite que SQL sea lo suficientemente flexible para adaptarse a diferentes tareas, desde consultas simples hasta transacciones complejas, sin perder su simplicidad y accesibilidad como lenguaje.

Definición y Tipos de Datos en SQL

En SQL, definir los datos correctamente es crucial para garantizar que la base de datos sea capaz de almacenar la información de manera eficiente y coherente. SQL permite definir diferentes tipos de datos y aplicar restricciones para asegurar que los datos ingresados sigan las reglas establecidas.

Tipos de Datos en SQL

Los **tipos de datos** son una parte fundamental de la definición de cualquier columna en una tabla. Definir correctamente los tipos de datos es clave para evitar errores y optimizar el rendimiento de la base de datos. Los tipos de datos se dividen en las siguientes categorías principales:

1. **Tipos de datos numéricos**:
   - **INT**: Almacena números enteros. Dependiendo del sistema, puede haber variantes como `TINYINT`, `SMALLINT`, `BIGINT`, que indican diferentes tamaños de almacenamiento.
     - Ejemplo: `edad INT`

- **DECIMAL(p,s)** o **NUMERIC(p,s)**: Almacenan números decimales con precisión y escala. `p` indica la precisión total (número de dígitos) y `s` la escala (número de dígitos después del punto decimal).
  - Ejemplo: `salario DECIMAL(10,2)`

2. **Tipos de datos de caracteres**:
   - **CHAR(n)**: Almacena cadenas de longitud fija de `n` caracteres. Es útil cuando se sabe que los datos siempre tendrán el mismo tamaño.
     - Ejemplo: `codigo_postal CHAR(5)`
   - **VARCHAR(n)**: Almacena cadenas de longitud variable, hasta un máximo de `n` caracteres. Solo ocupa el espacio necesario para los datos ingresados.
     - Ejemplo: `nombre VARCHAR(50)`
3. **Tipos de datos de fecha y hora**:
   - **DATE**: Almacena una fecha en el formato `AAAA-MM-DD`.
     - Ejemplo: `fecha_nacimiento DATE`
   - **TIME**: Almacena solo la hora en el formato `HH:MM:SS`.
     - Ejemplo: `hora_registro TIME`
   - **TIMESTAMP**: Almacena una combinación de fecha y hora, incluyendo segundos fraccionados en algunos casos.
     - Ejemplo: `fecha_modificacion TIMESTAMP`
4. **Tipos de datos binarios**:
   - **BLOB**: Almacena grandes volúmenes de datos binarios (imágenes, archivos, etc.).
   - **CLOB**: Almacena grandes cantidades de texto.
5. **Tipos de datos booleanos**:
   - **BOOLEAN**: Almacena valores lógicos `TRUE`, `FALSE` o `UNKNOWN`.

Esquema en SQL

Un **esquema** en SQL es una **colección lógica** de objetos dentro de una base de datos que están relacionados entre sí. Estos objetos pueden ser:

- **Tablas**
- **Vistas**
- **Índices**
- **Procedimientos almacenados**
- **Funciones**

El esquema se usa principalmente para organizar los objetos de la base de datos de manera ordenada y estructurada. Piensa en un esquema como una **carpeta** dentro de una base de datos que agrupa varios archivos (objetos) relacionados. Esto ayuda a mantener la base de datos organizada, especialmente cuando tienes muchos objetos.

Propósito del esquema:

- **Organización**: Un esquema ayuda a agrupar objetos relacionados. Por ejemplo, puedes tener un esquema para todos los objetos relacionados con "Recursos Humanos" y otro para "Ventas".
- **Seguridad**: Los permisos y accesos pueden ser asignados a nivel de esquema, lo que facilita el control de quién puede acceder o modificar ciertos objetos de la base de datos.

Ejemplo:

Si tienes una base de datos con dos departamentos diferentes (Recursos Humanos y Ventas), puedes crear un esquema para cada uno:

```
CREATE SCHEMA RecursosHumanos;
CREATE SCHEMA Ventas;
```

Dentro del esquema **RecursosHumanos**, podrías tener tablas como "Empleados" y "Salarios", mientras que en el esquema **Ventas** podrías tener tablas como "Clientes" y "Pedidos". Esto permite mantener los objetos separados y mejor organizados.

Catálogo en SQL

Un **catálogo** en SQL es una colección de **esquemas**. Es el siguiente nivel en la jerarquía de la base de datos. Si los esquemas son como "carpetas" que organizan objetos dentro de una base de datos, el catálogo sería como una "carpeta más grande" que organiza múltiples esquemas.

Un catálogo agrupa varios esquemas y generalmente contiene **metadatos** sobre los objetos en esos esquemas (información como nombres de tablas, tipos de columnas, índices, etc.). Cada base de datos puede tener uno o más catálogos.

En la práctica, muchos sistemas de gestión de bases de datos utilizan un catálogo por defecto, y no necesitas preocuparte tanto por este nivel a menos que trabajes con múltiples catálogos.

Relación entre Catálogo, Esquema y Objetos:

- **Catálogo**: Contiene uno o más esquemas.
- **Esquema**: Contiene varios objetos (tablas, vistas, etc.).
- **Objetos**: Los elementos reales que se almacenan en la base de datos (como tablas, vistas, procedimientos almacenados).

Ejemplo gráfico de la jerarquía:

```
Catálogo
    ├── Esquema 1
    │     ├── Tabla A
    │     ├── Vista B
    │     └── Índice C
    ├── Esquema 2
    │     ├── Tabla D
```

```
│   └── Procedimiento Almacenado E
└── Esquema 3
    ├── Tabla F
    └── Vista G
```

Ejemplo práctico en SQL

Supón que tienes un esquema llamado **Ventas** y dentro de este esquema tienes una tabla llamada **Pedidos**. Para referirte a la tabla de manera completa, puedes usar el nombre cualificado:

```
Ventas.Pedidos
```

Esto es útil si tienes varias tablas con el mismo nombre en diferentes esquemas (por ejemplo, si tienes otra tabla **Pedidos** en el esquema **Compras**).

Catálogo en SQL Server

En **SQL Server**, un **catálogo** se refiere principalmente a la **base de datos** en sí misma. Cada base de datos es, en realidad, un catálogo que contiene múltiples esquemas y objetos relacionados. En la práctica, SQL Server no expone la noción de múltiples catálogos para los usuarios de manera explícita como lo hacen algunos otros sistemas de bases de datos, ya que cada base de datos individual se considera su propio catálogo.

El **catálogo del sistema** de SQL Server también contiene metadatos acerca de las bases de datos, esquemas y objetos, y se puede acceder a estos metadatos usando vistas del sistema, como `sys.tables`, `sys.schemas`, `sys.databases`, entre otras.

Ejemplo de uso del catálogo en SQL Server:

- Si quieres consultar todas las tablas dentro de una base de datos (o catálogo), puedes usar una vista de sistema:

  ```
  SELECT * FROM sys.tables;
  ```

- Para obtener una lista de todos los esquemas dentro de una base de datos:

  ```
  SELECT * FROM sys.schemas;
  ```

Cada base de datos en SQL Server actúa como un **catálogo** independiente que contiene esquemas y objetos asociados. No necesitas gestionar explícitamente múltiples catálogos, ya que SQL Server organiza todo dentro de bases de datos individuales.

Ejemplo Práctico Completo en SQL Server:

1. **Crear una base de datos (catálogo)**:

   ```
   CREATE DATABASE EmpresaDB;
   GO
   ```

2. **Crear un esquema en esa base de datos**:

```
USE EmpresaDB;
GO
CREATE SCHEMA Ventas;
GO
```

3. **Crear una tabla dentro del esquema**:

```
CREATE TABLE Ventas.Clientes (
    Id INT PRIMARY KEY,
    Nombre NVARCHAR(50),
    Email NVARCHAR(100)
);
```

4. **Consultar la tabla utilizando el nombre cualificado**:

```
SELECT * FROM Ventas.Clientes;
```

El **nombre cualificado** en SQL es un nombre completo que incluye tanto el nombre del objeto (como una tabla, vista o columna) como su contexto, es decir, su esquema y/o base de datos. Este tipo de nomenclatura se utiliza para identificar un objeto de forma única dentro de una base de datos o incluso entre múltiples bases de datos, especialmente cuando existen varios objetos con el mismo nombre en diferentes esquemas o bases de datos.

En **SQL Server**, un nombre cualificado generalmente tiene tres niveles (base de datos, esquema y objeto) y sigue esta estructura:

Estructura de un nombre cualificado:

```
[base_de_datos].[esquema].[objeto]
```

- **base_de_datos**: El nombre de la base de datos donde se encuentra el objeto.
- **esquema**: El nombre del esquema al que pertenece el objeto dentro de la base de datos.
- **objeto**: El nombre del objeto en sí, como una tabla, vista o procedimiento.

Ejemplo de nombre cualificado:

```
SELECT * FROM MiBaseDeDatos.dbo.Empleados;
```

En este caso:

- **`MiBaseDeDatos`** es el nombre de la base de datos.
- **`dbo`** es el nombre del esquema (por defecto, SQL Server usa `dbo` para objetos creados sin un esquema especificado).
- **`Empleados`** es el nombre de la tabla que estás consultando.

¿Por qué usar nombres cualificados?

1. **Claridad y precisión**: En bases de datos grandes o con múltiples esquemas, puede haber tablas con el mismo nombre. Usar nombres cualificados asegura que SQL Server sepa exactamente a qué tabla, vista u objeto te estás refiriendo.
   - Ejemplo: Podrías tener una tabla llamada `Clientes` en el esquema `Ventas` y otra tabla con el mismo nombre en el esquema `Compras`. Al usar nombres cualificados, puedes especificar a cuál de ellas te refieres:

     ```
     SELECT * FROM Ventas.Clientes;  -- Se refiere a la tabla
     Clientes en el esquema Ventas
     SELECT * FROM Compras.Clientes; -- Se refiere a la tabla
     Clientes en el esquema Compras
     ```

2. **Acceso a objetos en otras bases de datos**: En un servidor SQL Server, puedes usar nombres cualificados para consultar o modificar datos en otra base de datos dentro del mismo servidor.
   - Ejemplo: Si tienes dos bases de datos en el mismo servidor, `MiBaseDeDatos1` y `MiBaseDeDatos2`, puedes acceder a una tabla en `MiBaseDeDatos2` desde `MiBaseDeDatos1`:

     ```
     SELECT * FROM MiBaseDeDatos2.dbo.Productos;
     ```

Componentes de un nombre cualificado:

1. **Base de datos** (opcional): El nombre de la base de datos en la que se encuentra el objeto.
   - Si omites el nombre de la base de datos, SQL Server asume que el objeto está en la base de datos actual que estás utilizando.
2. **Esquema** (opcional): El nombre del esquema dentro de la base de datos.
   - Si omites el nombre del esquema, SQL Server asume que estás trabajando dentro del esquema predeterminado del usuario, que generalmente es `dbo`.
3. **Nombre del objeto** (obligatorio): El nombre del objeto (tabla, vista, etc.) que estás consultando o manipulando.

Ejemplos de nombres cualificados en SQL Server:

1. **Nombre completo con base de datos, esquema y objeto**:

   ```
   SELECT * FROM MiBaseDeDatos.dbo.Empleados;
   ```

2. **Nombre con esquema y objeto (sin especificar la base de datos)**:

   ```
   SELECT * FROM dbo.Empleados;
   ```

3. **Solo el nombre del objeto (sin especificar esquema ni base de datos)**:

   ```
   SELECT * FROM Empleados;
   ```

- En este caso, SQL Server buscará la tabla `Empleados` en el esquema predeterminado de la base de datos actual.

Ventajas del uso de nombres cualificados:

- **Evitar ambigüedad**: Si tienes objetos con el mismo nombre en diferentes esquemas o bases de datos, los nombres cualificados evitan confusiones.
- **Acceso a objetos en diferentes bases de datos**: Permite realizar consultas entre diferentes bases de datos sin cambiar el contexto de la base de datos actual.
- **Mejora de seguridad y control**: Al usar esquemas y nombres cualificados, puedes organizar mejor los objetos y controlar los accesos a nivel de esquema o base de datos.

Resumen:

Un **nombre cualificado** es una manera completa de identificar un objeto en SQL Server usando el nombre de la base de datos, esquema y el objeto en sí. Proporciona claridad y precisión al referirse a objetos en bases de datos grandes o en múltiples bases de datos dentro de un servidor SQL.

CREATE

En **SQL Server**, el comando **`CREATE`** permite crear una amplia variedad de objetos dentro de una base de datos. A continuación te proporciono un **listado de los objetos** que se pueden crear con el comando `CREATE`:

Listado de Objetos que se pueden crear con `CREATE` en SQL Server

1. **`CREATE DATABASE`**: Crea una nueva base de datos.
   - **Uso**: Define una nueva base de datos en el servidor.
   - **Ejemplo**:

     ```
     CREATE DATABASE MiBaseDeDatos;
     ```

2. **`CREATE SCHEMA`**: Crea un nuevo esquema dentro de una base de datos.
   - **Uso**: Organiza objetos (tablas, vistas, etc.) dentro de un contenedor lógico llamado esquema.
   - **Ejemplo**:

     ```
     CREATE SCHEMA Ventas;
     ```

3. **`CREATE TABLE`**: Crea una nueva tabla en una base de datos.
   - **Uso**: Define una tabla con sus columnas y restricciones.
   - **Ejemplo**:

     ```
     CREATE TABLE Empleados (
         Id INT PRIMARY KEY,
         Nombre NVARCHAR(50)
     );
     ```

4. **`CREATE VIEW`**: Crea una vista (tabla virtual basada en una consulta SQL).
   - **Uso**: Proporciona una manera de ver los datos resultantes de una consulta como una tabla virtual.
   - **Ejemplo**:

```
CREATE VIEW VistaEmpleados AS
SELECT Nombre, Salario FROM Empleados;
```

5. **`CREATE INDEX`**: Crea un índice para mejorar el rendimiento de las consultas.
   - **Uso**: Acelera las consultas que involucran búsquedas o filtrados en grandes conjuntos de datos.
   - **Ejemplo**:

```
CREATE INDEX idx_nombre ON Empleados(Nombre);
```

6. **`CREATE PROCEDURE`**: Crea un procedimiento almacenado.
   - **Uso**: Define un conjunto de instrucciones SQL que se pueden ejecutar como una única operación.
   - **Ejemplo**:

```
CREATE PROCEDURE sp_AumentarSalario
@Porcentaje DECIMAL(5,2)
AS
BEGIN
    UPDATE Empleados SET Salario = Salario * (1 +
@Porcentaje / 100);
END;
```

7. **`CREATE FUNCTION`**: Crea una función definida por el usuario.
   - **Uso**: Define una función que puede devolver un valor único o una tabla.
   - **Ejemplo**:

```
CREATE FUNCTION ObtenerEdad (@FechaNacimiento DATE)
RETURNS INT
AS
BEGIN
    RETURN DATEDIFF(YEAR, @FechaNacimiento, GETDATE());
END;
```

8. **`CREATE TRIGGER`**: Crea un disparador (trigger).
   - **Uso**: Define un conjunto de acciones que se ejecutan automáticamente en respuesta a eventos como `INSERT`, `UPDATE` o `DELETE` en una tabla.
   - **Ejemplo**:

```
CREATE TRIGGER trg_AuditEmpleado
ON Empleados
AFTER INSERT, UPDATE
AS
BEGIN
    INSERT INTO AuditLog (EmpleadoId, Fecha) SELECT Id,
GETDATE() FROM inserted;
END;
```

9. **`CREATE USER`**: Crea un nuevo usuario en la base de datos.
   - **Uso**: Define un usuario para acceder y operar dentro de la base de datos.
   - **Ejemplo**:

```
CREATE USER Juan FOR LOGIN JuanLogin;
```

10. **`CREATE ROLE`**: Crea un nuevo rol en la base de datos.
    - **Uso**: Define un rol de usuario para agrupar permisos y asignarlos a usuarios.
    - **Ejemplo**:

```
CREATE ROLE AdministradorVentas;
```

11. **`CREATE LOGIN`**: Crea un inicio de sesión (login) para el acceso al servidor.
    - **Uso**: Define un login que puede autenticarse en la instancia de SQL Server.
    - **Ejemplo**:

```
CREATE LOGIN JuanLogin WITH PASSWORD = 'contraseñaSegura';
```

12. **`CREATE SEQUENCE`**: Crea una secuencia de números.
    - **Uso**: Genera números consecutivos automáticamente, que pueden ser usados para claves primarias o numeración.
    - **Ejemplo**:

```
CREATE SEQUENCE NumeroFactura
START WITH 1
INCREMENT BY 1;
```

13. **`CREATE SYNONYM`**: Crea un sinónimo (alias) para un objeto.
    - **Uso**: Define un nombre alternativo para un objeto como una tabla, vista, etc.
    - **Ejemplo**:

```
CREATE SYNONYM EmpleadosAlt AS dbo.Empleados;
```

14. **`CREATE TYPE`**: Crea un tipo de datos definido por el usuario.
    - **Uso**: Define un nuevo tipo de datos que puede ser usado en las tablas.
    - **Ejemplo**:

```
CREATE TYPE Telefono AS VARCHAR(15);
```

15. **`CREATE ASSEMBLY`**: Crea un ensamblado .NET que puede ser referenciado y utilizado en SQL Server.
    - **Uso**: Permite la integración de código .NET dentro de SQL Server.
    - **Ejemplo**:

```
CREATE ASSEMBLY MiEnsamblado FROM
'C:\ruta\a\miensamblado.dll';
```

16. **`CREATE XML SCHEMA COLLECTION`**: Crea una colección de esquemas XML.
    - **Uso**: Define un conjunto de esquemas XML que se pueden usar para validar datos XML en SQL Server.
    - **Ejemplo**:

```
CREATE XML SCHEMA COLLECTION MiEsquemaXML AS
'<schema>...</schema>';
```

17. **`CREATE AGGREGATE`**: Crea una función de agregación definida por el usuario.
    - **Uso**: Permite crear funciones de agregación personalizadas como `SUM` o `AVG`.
    - **Ejemplo**:

```
CREATE AGGREGATE MiAgregado (@Input INT) RETURNS INT;
```

18. **`CREATE FULLTEXT CATALOG`**: Crea un catálogo de texto completo.
    - **Uso**: Define un catálogo para búsqueda de texto completo en SQL Server.
    - **Ejemplo**:

```
CREATE FULLTEXT CATALOG MiCatalogoTextoCompleto;
```

19. **`CREATE FULLTEXT INDEX`**: Crea un índice de texto completo para habilitar búsquedas de texto completo en una tabla.
    - **Uso**: Permite búsquedas rápidas sobre grandes volúmenes de texto.
    - **Ejemplo**:

```
CREATE FULLTEXT INDEX ON Empleados(Nombre)
KEY INDEX PK_Empleados;
```

Objetos adicionales que se pueden crear con `CREATE` en SQL Server:

20. **`CREATE PARTITION FUNCTION`**: Crea una función de partición que define cómo las filas de una tabla o índice se distribuyen entre particiones.

- **Uso**: Para distribuir filas en particiones según un valor de columna.
- **Ejemplo**:

```
CREATE PARTITION FUNCTION MiFuncionParticion (INT)
AS RANGE LEFT FOR VALUES (1000, 2000, 3000);
```

21. **`CREATE PARTITION SCHEME`**: Crea un esquema de partición que especifica los archivos en los que se almacenan las particiones definidas por una función de partición.

- **Ejemplo**:

```
CREATE PARTITION SCHEME MiEsquemaParticion
AS PARTITION MiFuncionParticion TO (FileGroup1, FileGroup2,
FileGroup3);
```

22. **`CREATE SPATIAL INDEX`**: Crea un índice espacial para realizar búsquedas y consultas eficientes sobre datos espaciales.

- **Uso**: Para optimizar consultas sobre datos geoespaciales (puntos, polígonos, etc.).
- **Ejemplo**:

```
CREATE SPATIAL INDEX idx_spatial
ON Geometria(ColumnaGeografica);
```

23. **`CREATE DEFAULT`** (obsoleto, pero aún soportado): Crea un valor predeterminado independiente que se puede enlazar a una columna.

- **Ejemplo**:

```
CREATE DEFAULT MiDefault AS 'Desconocido';
```

24. **`CREATE BINDING`**: Asocia un tipo de datos definido por el usuario a una regla o valor predeterminado.

- **Ejemplo**:

```
CREATE BINDING MiBinding;
```

25. **`CREATE RULE`** (obsoleto, pero aún soportado): Crea una regla que puede enlazarse a una columna para restringir los valores que puede almacenar.

- **Ejemplo**:

```
CREATE RULE MiRegla AS @valor BETWEEN 1 AND 100;
```

26. **`CREATE CONTRACT`**: Crea un contrato para servicios de mensajería en el contexto de Service Broker.

- **Uso**: Define el tipo de mensajes que pueden ser enviados entre servicios.
- **Ejemplo**:

```
CREATE CONTRACT MiContrato
(MiMensaje SENT BY INITIATOR);
```

27. **`CREATE MESSAGE TYPE`**: Define un tipo de mensaje que se puede usar en una conversación de Service Broker.

- **Ejemplo**:

```
CREATE MESSAGE TYPE MiMensaje
VALIDATION = WELL_FORMED_XML;
```

28. **`CREATE SERVICE`**: Crea un servicio para usar con el motor de mensajería de Service Broker.

- **Ejemplo**:

```
CREATE SERVICE MiServicio
ON QUEUE MiCola (MiContrato);
```

29. **`CREATE QUEUE`**: Crea una cola para almacenar mensajes en el contexto de Service Broker.

- **Ejemplo**:

```
CREATE QUEUE MiCola;
```

30. **`CREATE EVENT NOTIFICATION`**: Crea una notificación de eventos para enviar mensajes cuando ocurren ciertos eventos en la base de datos o servidor.

- **Ejemplo**:

```
CREATE EVENT NOTIFICATION MiNotificacion
ON DATABASE
FOR CREATE_TABLE
TO SERVICE 'MiServicio', 'MiContrato';
```

31. **`CREATE EVENT SESSION`**: Crea una sesión de eventos extendidos (Extended Events) para capturar eventos de diagnóstico.

- **Ejemplo**:

```
CREATE EVENT SESSION MiSesionEventos
ON SERVER
ADD EVENT sqlserver.lock_acquired;
```

32. **`CREATE ASYMMETRIC KEY`**: Crea una clave asimétrica para cifrado.

- **Ejemplo**:

```
CREATE ASYMMETRIC KEY MiClaveAsimetrica
WITH ALGORITHM = RSA_2048;
```

33. **`CREATE CERTIFICATE`**: Crea un certificado que se puede usar para cifrar datos o autenticarse en conexiones.

- **Ejemplo**:

```
CREATE CERTIFICATE MiCertificado
WITH SUBJECT = 'Certificado para Encriptación';
```

34. **`CREATE SYMMETRIC KEY`**: Crea una clave simétrica para cifrado.

- **Ejemplo**:

```
CREATE SYMMETRIC KEY MiClaveSimetrica
```

```
WITH ALGORITHM = AES_256;
```

35. **`CREATE MASTER KEY`**: Crea una clave maestra de base de datos para proteger otras claves dentro de una base de datos.

- **Ejemplo**:

```
CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'ContraseñaSegura';
```

36. **`CREATE DATABASE AUDIT`**: Crea un objeto de auditoría a nivel de base de datos.

- **Ejemplo**:

```
CREATE DATABASE AUDIT EsquemaAuditoria
WITH (QUEUE_DELAY = 1000);
```

37. **`CREATE SERVER AUDIT`**: Crea un objeto de auditoría a nivel de servidor.

- **Ejemplo**:

```
CREATE SERVER AUDIT AuditoriaServidor
TO FILE ('C:\Auditorias');
```

38. **`CREATE LOGIN AUDIT`**: Crea un objeto de auditoría para auditorías específicas de inicio de sesión.

- **Ejemplo**:

```
CREATE LOGIN AUDIT AuditoriaLogin
TO FILE ('C:\AuditoriasLogin');
```

Ejemplo de creación de una tabla con diferentes tipos de datos:

```
CREATE TABLE Empleados (
    IdEmpleado INT PRIMARY KEY,
                -- Entero como clave primaria
    Nombre NVARCHAR(50) NOT NULL,
                -- Cadena de texto de longitud variable
    FechaNacimiento DATE,
                -- Fecha (año, mes, día)
    Salario DECIMAL(10, 2),
                -- Número decimal con 10 dígitos en total, 2 después
                del punto decimal
    Genero CHAR(1),
                -- Un solo carácter ('M' para masculino, 'F' para
                femenino)
    Telefono VARCHAR(15),
                -- Número de teléfono como texto (longitud variable)
    CorreoElectronico VARCHAR(100) UNIQUE,
                -- Correo electrónico con restricción de unicidad
    FechaContratacion DATETIME DEFAULT GETDATE(),
                -- Fecha y hora con valor predeterminado
    Estado BIT DEFAULT 1,
                -- Booleano (0 para falso, 1 para verdadero)
    FotoPerfil VARBINARY(MAX),
                -- Almacena datos binarios como una imagen
    Direccion NVARCHAR(200),
                -- Cadena de texto con hasta 200 caracteres
    Comentarios TEXT,
                -- Texto largo (almacenar comentarios o notas)
    UltimaModificacion TIMESTAMP
            -- Marca temporal (cambia automáticamente con cada
            modificación)
);
```

Descripción de los tipos de datos utilizados:

1. **`INT`**: Entero. En este caso, `IdEmpleado` es la clave primaria de la tabla.
   - Ejemplo: `IdEmpleado INT PRIMARY KEY`
2. **`NVARCHAR(n)`**: Cadena de texto de longitud variable que soporta caracteres Unicode (por ejemplo, nombres con acentos). `n` especifica la longitud máxima.
   - Ejemplo: `Nombre NVARCHAR(50)`
3. **`DATE`**: Almacena solo la fecha (año, mes, día).
   - Ejemplo: `FechaNacimiento DATE`
4. **`DECIMAL(p, s)`**: Número decimal, donde `p` es la precisión total (cantidad de dígitos) y `s` es la escala (cantidad de dígitos después del punto decimal).
   - Ejemplo: `Salario DECIMAL(10, 2)`
5. **`CHAR(n)`**: Cadena de longitud fija, útil para almacenar códigos o valores de longitud constante.
   - Ejemplo: `Genero CHAR(1)` (almacena un solo carácter)
6. **`VARCHAR(n)`**: Cadena de texto de longitud variable que no soporta Unicode. `n` es el número máximo de caracteres.
   - Ejemplo: `Telefono VARCHAR(15)`
7. **`DATETIME`**: Almacena la fecha y la hora.

   - Ejemplo: `FechaContratacion DATETIME DEFAULT GETDATE()` (por defecto, se asigna la fecha y hora actual)
8. **`BIT`**: Representa un valor booleano. En SQL Server, `BIT` se utiliza para representar valores de `0` (falso) y `1` (verdadero).
   - Ejemplo: `Estado BIT DEFAULT 1`
9. **`VARBINARY(MAX)`** : Almacena datos binarios de longitud variable, como imágenes o archivos.
   - Ejemplo: `FotoPerfil VARBINARY(MAX)`
10. **`TEXT`**: Almacena grandes cantidades de texto (útil para comentarios, descripciones largas, etc.). Aunque es una opción válida, en versiones modernas de SQL Server es preferible utilizar `VARCHAR(MAX)`.

    - Ejemplo: `Comentarios TEXT`

11. **`TIMESTAMP`**: Cada vez que se realiza una modificación en una fila, la columna de tipo `TIMESTAMP` se actualiza automáticamente. Se utiliza principalmente para el control de versiones.

- Ejemplo: `UltimaModificacion TIMESTAMP`

12. **`UNIQUE`**: Se aplica a la columna `CorreoElectronico` para asegurar que todos los valores en esa columna sean únicos dentro de la tabla.

- Ejemplo: `CorreoElectronico VARCHAR(100) UNIQUE`

Explicación de las restricciones:

- **`PRIMARY KEY`**: Define la clave primaria, que debe ser única y no nula. En este caso, la columna `IdEmpleado` es la clave primaria.
- **`NOT NULL`**: Asegura que la columna no puede tener valores nulos. En este ejemplo, `Nombre` es obligatorio.
- **`DEFAULT`**: Establece un valor predeterminado para una columna. Por ejemplo, `FechaContratacion` usará la fecha y hora actual si no se proporciona un valor.
- **`UNIQUE`**: Garantiza que los valores en esa columna sean únicos. Por ejemplo, ningún empleado puede tener el mismo `CorreoElectronico`.

Para restringir que la columna **`Genero`** en SQL Server solo acepte los valores `'M'` (Masculino) o `'F'` (Femenino), puedes utilizar una **restricción `CHECK`**. Esta restricción permite definir una condición que los valores deben cumplir antes de ser insertados o actualizados en la tabla.

Ejemplo de cómo aplicar la restricción `CHECK`:

```
CREATE TABLE Empleados (
    IdEmpleado INT PRIMARY KEY,
    Nombre NVARCHAR(50) NOT NULL,
```

```
    FechaNacimiento DATE,
    Salario DECIMAL(10, 2),
    Genero CHAR(1) NOT NULL CHECK (Genero IN ('M', 'F')),  --
Restricción CHECK
    Telefono VARCHAR(15),
    CorreoElectronico VARCHAR(100) UNIQUE,
    FechaContratacion DATETIME DEFAULT GETDATE(),
    Estado BIT DEFAULT 1
);
```

Explicación:

- **`Genero CHAR(1)`**: Define la columna `Genero` como un carácter de longitud fija (solo permite un carácter: `'M'` o `'F'`).
- **`CHECK (Genero IN ('M', 'F'))`**: La restricción `CHECK` garantiza que los valores que se ingresen en la columna `Genero` solo pueden ser `'M'` o `'F'`. Si se intenta insertar o actualizar cualquier valor distinto a `'M'` o `'F'`, SQL Server lanzará un error y no permitirá la operación.

Ejemplo de inserción correcta:

```
INSERT INTO Empleados (IdEmpleado, Nombre, FechaNacimiento, Salario,
Genero)
VALUES (1, 'Juan Pérez', '1990-01-15', 50000, 'M');
```

Ejemplo de inserción incorrecta (genera error):

```
INSERT INTO Empleados (IdEmpleado, Nombre, FechaNacimiento, Salario,
Genero)
VALUES (2, 'Ana Gómez', '1985-06-30', 55000, 'X');  -- 'X' no es
permitido
```

¿Cómo funciona la restricción `CHECK`?

1. **Valida los valores antes de insertarlos o actualizarlos**: Cuando se intenta insertar o actualizar un valor en la columna `Genero`, SQL Server verifica que el valor cumpla con la condición definida en la restricción `CHECK`.
2. **Rechaza valores inválidos**: Si el valor no es `'M'` ni `'F'`, el motor de la base de datos no permitirá que el cambio se realice y devolverá un error.

Alternativa: Usar una tabla de referencia para más flexibilidad

Si en el futuro necesitas agregar más géneros (por ejemplo, `'O'` para "Otro"), puedes considerar usar una **tabla de referencia** para gestionar los valores permitidos en lugar de una restricción `CHECK`. Esto proporciona más flexibilidad para agregar nuevos valores sin modificar la estructura de la tabla.

Ejemplo con tabla de referencia:

1. **Crear la tabla de géneros**:

```
CREATE TABLE Generos (
    CodigoGenero CHAR(1) PRIMARY KEY,
    Descripcion NVARCHAR(50)
);
```

2. **Insertar los valores permitidos en la tabla `Generos`:**

```
INSERT INTO Generos (CodigoGenero, Descripcion)
VALUES ('M', 'Masculino'), ('F', 'Femenino'), ('O', 'Otro');
```

3. **Modificar la tabla `Empleados` para usar una clave foránea:**

```
CREATE TABLE Empleados (
    IdEmpleado INT PRIMARY KEY,
    Nombre NVARCHAR(50) NOT NULL,
    FechaNacimiento DATE,
    Salario DECIMAL(10, 2),
    Genero CHAR(1) NOT NULL,
    FOREIGN KEY (Genero) REFERENCES Generos(CodigoGenero), --
Restricción con clave foránea
    Telefono VARCHAR(15),
    CorreoElectronico VARCHAR(100) UNIQUE,
    FechaContratacion DATETIME DEFAULT GETDATE(),
    Estado BIT DEFAULT 1
);
```

Con esta alternativa, cualquier valor de género debe estar presente en la tabla `Generos`, lo que te permite agregar o eliminar opciones de manera más flexible.

No es estrictamente necesario usar **`NOT NULL`** junto con la restricción **`CHECK`** para limitar los valores de una columna, como en el caso de los géneros `'M'` o `'F'`. Sin embargo, **es recomendable** si deseas asegurarte de que la columna **no acepte valores nulos**.

¿Qué sucede sin `NOT NULL`?

Si no defines la columna como **`NOT NULL`**, aunque la restricción **`CHECK`** (como `CHECK (Genero IN ('M', 'F'))`) esté presente, la columna aún permitirá **valores nulos (`NULL`)**. Esto se debe a que en SQL, el valor `NULL` no se evalúa como verdadero o falso, sino como "desconocido", y **`NULL` no viola la restricción `CHECK`**.

CONSTRAINT

Un **`CONSTRAINT`** es una forma de nombrar y definir restricciones en SQL Server de manera explícita. Se utiliza para aplicar restricciones a columnas y asegurarse de que los datos cumplan ciertas condiciones, como garantizar la unicidad, definir claves foráneas, y en este caso, garantizar que el valor de la columna **`Genero`** sea solo `'M'` o `'F'`.

En el caso de la columna **`Genero`**, puedes usar un **`CONSTRAINT CHECK`** para asegurarte de que los valores en esa columna sean solo `'M'` (Masculino) o `'F'` (Femenino), y opcionalmente, también puedes nombrar el `CONSTRAINT` para facilitar su identificación en caso de depuración o mantenimiento.

Ejemplo de cómo usar `CONSTRAINT` para la columna `Genero`:

```
CREATE TABLE Empleados (
    IdEmpleado INT PRIMARY KEY,
    Nombre NVARCHAR(50) NOT NULL,
    Genero CHAR(1) NOT NULL CONSTRAINT chk_genero CHECK (Genero IN
('M', 'F')),  -- CONSTRAINT CHECK
    FechaNacimiento DATE,
    Salario DECIMAL(10, 2)
);
```

Explicación:

1. **`Genero CHAR(1)`**: Define la columna `Genero` como un carácter de longitud fija, que almacenará `'M'` o `'F'`.
2. **`CONSTRAINT chk_genero`**: Esta parte del código **nombra la restricción** como `chk_genero`. Darle un nombre explícito a la restricción te ayuda a identificarla fácilmente si alguna vez necesitas modificarla o eliminarla.
3. **`CHECK (Genero IN ('M', 'F'))`**: Especifica la condición de que los valores en la columna `Genero` solo pueden ser `'M'` o `'F'`.

Ventajas de usar un `CONSTRAINT`:

- **Facilidad de mantenimiento**: Al asignarle un nombre a la restricción (en este caso, `chk_genero`), puedes referenciarla fácilmente si necesitas modificarla o eliminarla en el futuro.
- **Claridad**: Nombrar los `CONSTRAINTS` proporciona una documentación adicional sobre el propósito de la restricción, lo cual es útil para otros desarrolladores o administradores de bases de datos.

Cómo eliminar un `CONSTRAINT`:

Si en algún momento decides eliminar la restricción, puedes hacerlo utilizando el comando **`ALTER TABLE`** seguido de **`DROP CONSTRAINT`**.

Ejemplo de cómo eliminar el `CONSTRAINT`:

```
ALTER TABLE Empleados
DROP CONSTRAINT chk_genero;
```

Insertar datos con la restricción:

Inserción válida:

```
INSERT INTO Empleados (IdEmpleado, Nombre, Genero, FechaNacimiento,
Salario)
VALUES (1, 'Juan Pérez', 'M', '1990-01-01', 50000);
```

Inserción inválida (genera error):

```
INSERT INTO Empleados (IdEmpleado, Nombre, Genero, FechaNacimiento,
Salario)
VALUES (2, 'Ana Gómez', 'X', '1985-06-15', 55000);  -- 'X' no es
permitido por el CONSTRAINT
```

Genero CHAR(1) NOT NULL CHECK (Genero IN ('M', ‘F’, ‘O’)),

es lo mismo que

Genero CHAR(1) NOT NULL CONSTRAINT chk_genero CHECK (Genero IN ('M', ‘F’, ‘O’)),

La única **diferencia** entre las dos formas es que, en la segunda opción, con **`CONSTRAINT`**, da un **nombre explícito** a la restricción, lo cual facilita su **identificación** y **mantenimiento** en el futuro. Sin un nombre explícito, se tendría que buscar el nombre generado por el sistema, lo que puede ser más complicado. Cuando no especificas un nombre para la restricción mediante **`CONSTRAINT`**, **SQL Server** automáticamente le asigna un nombre generado por el sistema. Este nombre será una cadena alfanumérica, que no es fácil de recordar o identificar.

¿Cómo encontrar el nombre de la restricción generada automáticamente?

Puedes utilizar una consulta a las **vistas del sistema** de SQL Server para obtener el nombre de la restricción:

Consulta para encontrar el nombre de la restricción:

```
SELECT name
FROM sys.check_constraints
WHERE parent_object_id = OBJECT_ID('Empleados');
```

Esta consulta busca todas las restricciones `CHECK` asociadas a la tabla `Empleados`. El resultado incluirá el nombre generado automáticamente por SQL Server para la restricción.

Una vez que encuentres el nombre de la restricción, puedes eliminarla utilizando el comando **`ALTER TABLE`**:

Ejemplo de cómo eliminar una restricción sin nombre explícito:

```
ALTER TABLE Empleados
DROP CONSTRAINT nombre_generado_por_el_sistema;
```

La sentencia **`FOREIGN KEY (Genero) REFERENCES Generos(CodigoGenero)`** crea una **clave foránea** (foreign key) en la tabla que estás definiendo, en este caso, para la columna **`Genero`**, y establece una relación con la columna **`CodigoGenero`** de otra tabla llamada **`Generos`**.

Explicación detallada:

1. **`FOREIGN KEY (Genero)`**:
   - **`Genero`** es la columna en la tabla actual (la que estás creando) que será la clave foránea. Esto significa que los valores en esta columna deben corresponder a los valores de una columna en otra tabla.
   - En otras palabras, la columna `Genero` de esta tabla debe contener valores que ya existan en otra tabla, lo que garantiza **integridad referencial**.
2. **`REFERENCES Generos(CodigoGenero)`**:
   - **`Generos`** es el nombre de la tabla a la que se hace referencia, que contiene los valores válidos para la columna `Genero` en la tabla actual.
   - **`CodigoGenero`** es la columna de la tabla `Generos` que se usa como **clave primaria o única**. Los valores de esta columna deben coincidir con los valores de la columna `Genero` en la tabla actual.

   En otras palabras, la columna `Genero` en la tabla actual puede contener solo los valores que ya existen en la columna `CodigoGenero` de la tabla `Generos`.

¿Qué es una clave foránea (`FOREIGN KEY`)?

Una clave foránea es una restricción que garantiza que los valores en una columna o conjunto de columnas de una tabla deben coincidir con los valores de una columna o conjunto de columnas en otra tabla. Esto se usa para **enlazar las tablas** y asegurar la **integridad referencial**.

- **Integridad referencial**: Asegura que no se pueda insertar un valor en la columna `Genero` de la tabla actual si ese valor no existe en la tabla referenciada `Generos`.

Ejemplo práctico de uso:

Supongamos que tienes dos tablas:

1. **Tabla `Generos`**: Define los géneros permitidos.

```
CREATE TABLE Generos (
    CodigoGenero CHAR(1) PRIMARY KEY,      -- Clave primaria
    Descripcion NVARCHAR(50)
);

-- Insertar algunos valores en la tabla Generos
INSERT INTO Generos (CodigoGenero, Descripcion)
VALUES ('M', 'Masculino'), ('F', 'Femenino'), ('O', 'Otro');
```

2. **Tabla `Empleados`**: Contiene información de empleados y la columna `Genero` que se relaciona con la tabla `Generos`.

```
CREATE TABLE Empleados (
    IdEmpleado INT PRIMARY KEY,
    Nombre NVARCHAR(50) NOT NULL,
```

```
    Genero CHAR(1) NOT NULL,
    FOREIGN KEY (Genero) REFERENCES Generos(CodigoGenero) --
Clave foránea
);
```

Explicación del ejemplo:

- La columna `CodigoGenero` en la tabla `Generos` contiene los valores válidos para los géneros (`'M'`, `'F'`, `'O'`).
- La tabla `Empleados` tiene una columna llamada `Genero`, que será una clave foránea que hace referencia a la columna `CodigoGenero` en la tabla `Generos`.
- Esto significa que solo se podrán insertar en la columna `Genero` de la tabla `Empleados` los valores `'M'`, `'F'` o `'O'`, porque son los valores permitidos en la tabla `Generos`.

Ejemplo de inserción correcta:

```
INSERT INTO Empleados (IdEmpleado, Nombre, Genero)
VALUES (1, 'Juan Pérez', 'M');  -- 'M' existe en la tabla Generos
```

Ejemplo de inserción incorrecta (genera error):

```
INSERT INTO Empleados (IdEmpleado, Nombre, Genero)
VALUES (2, 'Ana Gómez', 'X');  -- 'X' no existe en la tabla Generos,
genera error
```

Beneficios de usar claves foráneas:

1. **Garantizar consistencia**: La clave foránea asegura que los valores en la columna `Genero` de la tabla `Empleados` deben estar siempre presentes en la tabla `Generos`. Esto evita errores como la inserción de valores no permitidos.
2. **Integridad referencial**: Si intentas eliminar un valor de la tabla `Generos` que está siendo referenciado por la tabla `Empleados`, SQL Server impedirá la operación (a menos que definas acciones específicas como `ON DELETE CASCADE`).

Diferencias clave entre `VARCHAR` y `NVARCHAR`:

1. **Codificación de caracteres**:
   - **`VARCHAR`**: Almacena texto usando el **conjunto de caracteres no Unicode**, lo que significa que utiliza una codificación de un solo byte para cada carácter. Es más eficiente en términos de espacio de almacenamiento si solo necesitas manejar caracteres ASCII o conjuntos de caracteres limitados.
   - **`NVARCHAR`**: Almacena texto usando **Unicode**, lo que significa que cada carácter ocupa 2 bytes. Esto permite almacenar caracteres de múltiples alfabetos y conjuntos de símbolos, como chino, japonés, árabe, cirílico, etc.
2. **Espacio de almacenamiento**:

- **`VARCHAR(n)`**: Cada carácter ocupa **1 byte**. El espacio de almacenamiento es más eficiente cuando se trabaja solo con caracteres ASCII o alfabéticos de lenguajes como el inglés.
- **`NVARCHAR(n)`**: Cada carácter ocupa **2 bytes**, lo que significa que los datos ocupan el doble de espacio en disco. Es ideal cuando se espera manejar caracteres internacionales o cualquier tipo de texto Unicode.

3. **Capacidad de almacenamiento**:
   - **`VARCHAR(n)`**: Puede almacenar hasta **8,000 caracteres** cuando `n` es 8,000.
   - **`NVARCHAR(n)`**: Puede almacenar hasta **4,000 caracteres** cuando `n` es 4,000. Aunque el número de caracteres almacenados es menor en comparación con `VARCHAR`, cada carácter puede ser cualquier símbolo Unicode, lo que lo hace más flexible.
4. **Compatibilidad con Unicode**:
   - **`VARCHAR`**: No admite caracteres Unicode, por lo que es adecuado para lenguajes con caracteres simples (como el inglés y otros lenguajes que no requieren un conjunto de caracteres extendido).
   - **`NVARCHAR`**: Admite caracteres Unicode, lo que lo hace esencial para almacenar texto que puede contener caracteres internacionales, como chino, árabe, o caracteres especiales.
5. **Uso en consultas y procesamiento**:
   - **`VARCHAR`**: Generalmente es más eficiente en términos de rendimiento y uso de espacio cuando se trabaja solo con texto ASCII o con conjuntos de caracteres no Unicode.
   - **`NVARCHAR`**: Se debe usar cuando existe la posibilidad de trabajar con caracteres internacionales o si tu base de datos necesita admitir múltiples lenguajes.

Ejemplo:

Usando `VARCHAR`:

```
CREATE TABLE Clientes (
    Nombre VARCHAR(50)
);
```

- Aquí se está diciendo que la columna `Nombre` puede almacenar hasta 50 caracteres ASCII, como los caracteres alfabéticos en inglés. Es eficiente en espacio, pero no se podrá almacenar correctamente texto que contenga caracteres especiales o de otros alfabetos (Unicode).

Usando `NVARCHAR`:

```
CREATE TABLE Clientes (
    Nombre NVARCHAR(50)
);
```

- Con `NVARCHAR`, la columna `Nombre` puede almacenar hasta 50 caracteres Unicode, lo que permite usar cualquier alfabeto internacional (caracteres chinos,

árabes, cirílicos, etc.). Sin embargo, se requerirán más bytes de almacenamiento para cada carácter.

Resumen de diferencias:

Característica	VARCHAR	NVARCHAR
Codificación	No Unicode (ASCII u otro)	Unicode
Tamaño por carácter	1 byte	2 bytes
Número máximo de caracteres (n)	Hasta 8,000 caracteres	Hasta 4,000 caracteres
Almacenamiento máximo (MAX)	2^31-1 bytes (~2 GB)	2^30-1 bytes (~1 GB)
Soporte multilingüe	No, solo conjuntos de caracteres limitados	Sí, admite caracteres internacionales

¿Cuándo usar cada uno?

- **Usa VARCHAR** cuando:
  - Solo necesitas manejar texto simple en inglés o idiomas similares (como español, francés, etc.) que no requieren caracteres Unicode.
  - Quieres ahorrar espacio y mejorar el rendimiento al trabajar con grandes cantidades de texto simple.
- **Usa NVARCHAR** cuando:
  - Necesitas almacenar caracteres internacionales (Unicode) o tienes la posibilidad de tener texto en diferentes idiomas o con caracteres especiales.
  - Trabajas en aplicaciones que pueden manejar diferentes lenguajes o juegos de caracteres.

SQL Server, si no especificas **NULL** o **NOT NULL** para una columna, el sistema asume que, por defecto, la columna puede aceptar **valores NULL**. Es decir:

- Si no defines explícitamente que una columna es **NOT NULL**, SQL Server asume que es **NULL**.
- Por lo tanto, **omitir NULL** es lo mismo que **escribir `NULL`**, ya que el sistema automáticamente lo considera.

Ejemplos:

1. **Declaración explícita con NULL**:

```
Telefono VARCHAR(15) NULL
```

2. **Sin especificar NULL ni NOT NULL** (se asume `NULL` por defecto):

```
    Telefono VARCHAR(15)
```

Ambas declaraciones son **equivalentes**, ya que, al no especificar **`NOT NULL`**, SQL Server permite valores nulos en esa columna.

Detalles de cada tipo de dato:

1. **`DATE`**:
   - Almacena únicamente la fecha en el formato **`YYYY-MM-DD`** (año-mes-día).
   - Rango de valores: desde **'0001-01-01'** hasta **'9999-12-31'**.
   - Precisión: Solo almacena la fecha, sin información de la hora.

   Ejemplo de uso de `DATE`:

   ```
   CREATE TABLE Eventos (
       IdEvento INT PRIMARY KEY,
       Fecha DATE  -- Solo almacena la fecha
   );

   -- Insertar una fecha
   INSERT INTO Eventos (IdEvento, Fecha)
   VALUES (1, '2024-09-18');
   ```

   - El valor almacenado en la columna `Fecha` será `'2024-09-18'`, sin ninguna hora asociada.
2. **`TIME`**:
   - Almacena solo la hora, en el formato **`HH:MM:SS[.nnnnnnn]`** (hora, minutos, segundos y opcionalmente fracciones de segundo).
   - Rango de valores: desde **'00:00:00.0000000'** hasta **'23:59:59.9999999'**.
   - Precisión: Puedes especificar el número de decimales de los segundos (hasta 7 dígitos de precisión).

   Ejemplo de uso de `TIME`:

   ```
   CREATE TABLE Turnos (
       IdTurno INT PRIMARY KEY,
       HoraInicio TIME(3)  -- Almacena solo la hora con 3 decimales
   en los segundos
   );

   -- Insertar una hora
   INSERT INTO Turnos (IdTurno, HoraInicio)
   VALUES (1, '08:30:15.123');  -- Hora con fracción de segundos
   ```

   - El valor almacenado en la columna `HoraInicio` será `'08:30:15.123'`, con precisión hasta los milisegundos.

Diferencia con el tipo `DATETIME`:

- **DATETIME** almacena tanto la fecha como la hora en un único campo, mientras que DATE y TIME permiten trabajar con **fechas y horas de manera independiente**.
    - **DATETIME**: Combina fecha y hora, con un rango desde **'1753-01-01'** hasta **'9999-12-31'**.
    - **DATE** y **TIME**: Separan la información, lo que puede ser más eficiente si solo necesitas almacenar una parte.

Ejemplo de tabla con ambos tipos de datos (DATE y TIME):

```
CREATE TABLE Reuniones (
    IdReunion INT PRIMARY KEY,
    FechaReunion DATE,          -- Solo la fecha
    HoraInicio TIME(0),         -- Solo la hora (sin fracciones de
segundos)
    HoraFin TIME(0)             -- Hora de fin
);

-- Insertar una reunión con fecha y horas
INSERT INTO Reuniones (IdReunion, FechaReunion, HoraInicio, HoraFin)
VALUES (1, '2024-09-18', '09:00', '10:00');
```

En este caso, puedes almacenar la **fecha de la reunión** y las **horas de inicio y fin** de manera independiente.

Características principales de TIMESTAMP/ROWVERSION:

1. **Generado automáticamente**: SQL Server asigna automáticamente un valor a la columna de tipo TIMESTAMP o ROWVERSION cuando se inserta o actualiza una fila en la tabla. El valor de la columna es un número binario único que cambia cada vez que se realiza una modificación en la fila.
2. **No es una fecha ni una hora**: A pesar del nombre "TIMESTAMP", este tipo de dato **no almacena información de fecha o tiempo**. Es un número único binario que cambia con cada operación de modificación de la fila. Si necesitas almacenar una marca de tiempo real (fecha y hora), deberías usar el tipo de dato **DATETIME** o **DATETIME2**.
3. **No se puede modificar manualmente**: No puedes insertar, actualizar o establecer manualmente un valor en una columna de tipo TIMESTAMP o ROWVERSION. SQL Server maneja esto de manera automática.
4. **Uso principal**: Se utiliza comúnmente para detectar cambios o concurrencias en filas de una tabla. Por ejemplo, puedes usar TIMESTAMP o ROWVERSION para identificar si una fila ha sido modificada desde la última vez que se leyó.

Ejemplo de uso:

```
CREATE TABLE Empleados (
    IdEmpleado INT PRIMARY KEY,
    Nombre NVARCHAR(50),
    Salario DECIMAL(10, 2),
    UltimaModificacion ROWVERSION  -- Columna de tipo ROWVERSION
(anteriormente TIMESTAMP)
```

```
);
```

En este ejemplo:

- Cada vez que se inserte o actualice una fila en la tabla `Empleados`, la columna `UltimaModificacion` se actualizará automáticamente con un nuevo valor único.
- No puedes modificar manualmente el valor de la columna `ROWVERSION`.

Intento de modificación manual (genera error):

```
INSERT INTO Empleados (IdEmpleado, Nombre, Salario,
UltimaModificacion)
VALUES (1, 'Juan Pérez', 50000, 0x00000000000007D3);  -- Esto generará
un error
```

Resultado: SQL Server genera un error porque no se permite modificar o insertar manualmente valores en una columna de tipo `TIMESTAMP` o `ROWVERSION`.

Diferencias clave entre `DATETIME` y `DATETIME2`:

Característica	`DATETIME`	`DATETIME2`
Rango de fechas	Desde **1753-01-01** hasta **9999-12-31**	Desde **0001-01-01** hasta **9999-12-31**
Rango de horas	Desde **00:00:00** hasta **23:59:59.997** (con precisión hasta milisegundos)	Desde **00:00:00.0000000** hasta **23:59:59.9999999** (con precisión de 100 nanosegundos)
Precisión de la hora	Precisión de **3 milisegundos**	Precisión ajustable: hasta **7 dígitos** fraccionarios de segundo
Tamaño de almacenamiento	**8 bytes**	Entre **6 y 8 bytes**, dependiendo de la precisión
Precisión predeterminada	No ajustable	**7 dígitos** (100 nanosegundos), pero se puede ajustar

Explicación detallada:

1. **Rango de fechas**:
   - `DATETIME`: Solo puede manejar fechas desde **1 de enero de 1753** hasta **31 de diciembre de 9999**. Esto se debe a que SQL Server utiliza el calendario gregoriano completo, y la fecha de 1 de enero de 1753 es cuando este calendario entró en vigencia en el Reino Unido y sus colonias.
   - `DATETIME2`: Amplía el rango de fechas, comenzando desde el **1 de enero del año 0001** hasta **31 de diciembre de 9999**, lo que permite manejar fechas históricas más antiguas y más precisas.

2. **Rango de horas**:
   - **`DATETIME`**: Almacena la hora con una precisión de hasta **3 milisegundos**, lo que significa que los valores se redondean al milisegundo más cercano.
   - **`DATETIME2`**: Puede almacenar tiempos con una precisión de hasta **100 nanosegundos**. Además, puedes especificar cuántos decimales de segundos necesitas, ajustando la precisión hasta **7 dígitos**.
3. **Tamaño de almacenamiento**:
   - **`DATETIME`**: Siempre ocupa **8 bytes**.
   - **`DATETIME2`**: El tamaño de almacenamiento es más eficiente y varía entre **6 y 8 bytes**, dependiendo del nivel de precisión especificado. Si necesitas menos precisión, puede usar menos espacio.
4. **Precisión de la hora**:
   - **`DATETIME`**: Tiene una precisión fija de **3 milisegundos**, por lo que las horas se redondean a los valores más cercanos en incrementos de 0, 3, 7... milisegundos.
   - **`DATETIME2`**: Tiene una precisión mucho mayor, permitiendo hasta **7 dígitos fraccionarios** de segundos. Esto significa que puede ser tan preciso como **100 nanosegundos**.
5. **Ajustabilidad de la precisión** (solo en `DATETIME2`):
   - En **`DATETIME2`**, puedes ajustar la precisión fraccionaria del componente de la hora al especificar el número de dígitos después del punto decimal:
     - `DATETIME2(0)`: Solo segundos, sin fracción de segundos.
     - `DATETIME2(3)`: Milisegundos.
     - `DATETIME2(7)`: Nanosegundos.

Ejemplo:

```
-- DATETIME2 con precisión de 3 (milisegundos)
DECLARE @Fecha DATETIME2(3) = '2024-09-18 12:34:56.123';
```

Ejemplos:

1. **Usando `DATETIME`**:

```
CREATE TABLE Eventos (
    IdEvento INT PRIMARY KEY,
    FechaEvento DATETIME
);

INSERT INTO Eventos (IdEvento, FechaEvento)
VALUES (1, '2024-09-18 14:35:45.123');
```

El valor de la hora se almacenará con una precisión de 3 milisegundos, lo que significa que los segundos se redondearán a los valores más cercanos: **'2024-09-18 14:35:45.123'**.

2. **Usando `DATETIME2` con precisión de 7 dígitos**:

```
CREATE TABLE Reuniones (
```

```
    IdReunion INT PRIMARY KEY,
    FechaReunion DATETIME2(7)
);

INSERT INTO Reuniones (IdReunion, FechaReunion)
VALUES (1, '2024-09-18 14:35:45.1234567');
```

En este caso, `DATETIME2(7)` almacena la fecha y la hora con una precisión de **7 dígitos** en los segundos, lo que significa que los segundos serán almacenados con hasta **100 nanosegundos** de precisión.

¿Cuándo usar `DATETIME2` en lugar de `DATETIME`?

- **`DATETIME2`** es generalmente **recomendado** sobre **`DATETIME`** porque:
  - Tiene un **rango más amplio** de fechas.
  - Ofrece **mayor precisión** en el manejo de horas.
  - Es **más eficiente en términos de almacenamiento**, especialmente cuando ajustas la precisión a tus necesidades.

Resumen:

Característica	**`DATETIME`**	**`DATETIME2`**
Rango de fechas	1753-01-01 a 9999-12-31	0001-01-01 a 9999-12-31
Rango de horas	Hasta 3 milisegundos	Hasta 100 nanosegundos
Tamaño de almacenamiento	8 bytes	6 a 8 bytes, dependiendo de la precisión
Precisión de la hora	3 milisegundos	Ajustable, hasta 7 dígitos (100 ns)
Uso recomendado	Situaciones más antiguas, pero con menos precisión	Situaciones que requieren fechas históricas o alta precisión de tiempo

En general, **`DATETIME2`** es una mejora sobre **`DATETIME`**, con un rango más amplio, mayor precisión y menor tamaño de almacenamiento en muchos casos. Si estás comenzando un nuevo proyecto, **`DATETIME2`** es la opción recomendada.

COMANDO ALTER

El comando **`ALTER`** en **SQL Server** se utiliza para **modificar la estructura de objetos existentes** en una base de datos, como tablas, esquemas, vistas, procedimientos almacenados, etc. Es parte del **DDL (Data Definition Language)** y permite hacer ajustes sin tener que eliminar y volver a crear el objeto.

A continuación, te doy una descripción detallada de cómo se usa el comando `ALTER`, especialmente para **tablas** y algunos ejemplos prácticos para que puedas entender su uso.

Sintaxis básica de `ALTER`:

1. **Para modificar una tabla**:

   ```
   ALTER TABLE nombre_tabla
   ```

2. **Para modificar otros objetos (procedimientos almacenados, vistas, etc.)**:

   ```
   ALTER objeto_tipo nombre_objeto
   ```

`ALTER TABLE`:

El comando **`ALTER TABLE`** se utiliza para **modificar** una tabla ya existente. Con `ALTER TABLE`, puedes:

- Agregar o eliminar columnas.
- Modificar las propiedades de las columnas.
- Agregar o eliminar restricciones.
- Cambiar nombres de columnas o de la tabla.

Operaciones comunes con `ALTER TABLE`:

1. Agregar una columna:

Si quieres agregar una nueva columna a una tabla, puedes usar el comando `ADD`:

```
ALTER TABLE nombre_tabla
ADD nueva_columna DATATYPE;
```

Ejemplo: Agregar una columna de correo electrónico a la tabla `Usuarios`:

```
ALTER TABLE Usuarios
ADD CorreoElectronico NVARCHAR(100);
```

2. Modificar una columna existente:

Puedes cambiar el tipo de datos o las propiedades de una columna existente con el comando `ALTER COLUMN`:

```
ALTER TABLE nombre_tabla
ALTER COLUMN nombre_columna NUEVO_DATATYPE;
```

Ejemplo: Cambiar el tipo de dato de la columna `Telefono` en la tabla `Usuarios`:

```
ALTER TABLE Usuarios
ALTER COLUMN Telefono NVARCHAR(20);
```

- **Nota**: No puedes modificar el tipo de dato de una columna si hay datos incompatibles existentes en esa columna.

3. Eliminar una columna:

Si ya no necesitas una columna, puedes eliminarla con `DROP COLUMN`:

```
ALTER TABLE nombre_tabla
DROP COLUMN nombre_columna;
```

Ejemplo: Eliminar la columna `Telefono` de la tabla `Usuarios`:

```
ALTER TABLE Usuarios
DROP COLUMN Telefono;
```

4. Agregar una restricción:

Puedes agregar una restricción como **`CHECK`**, **`UNIQUE`**, **`FOREIGN KEY`**, o **`DEFAULT`** con `ALTER TABLE`.

- **Agregar una restricción `CHECK`**:

  ```
  ALTER TABLE nombre_tabla
  ADD CONSTRAINT nombre_restriccion CHECK (condicion);
  ```

  Ejemplo: Agregar una restricción para que la columna `Edad` no permita valores menores de 18:

  ```
  ALTER TABLE Usuarios
  ADD CONSTRAINT chk_edad CHECK (Edad >= 18);
  ```

- **Agregar una clave foránea**:

  ```
  ALTER TABLE nombre_tabla
  ADD CONSTRAINT nombre_fk FOREIGN KEY (columna) REFERENCES
  tabla_referenciada(columna_referenciada);
  ```

  Ejemplo: Agregar una clave foránea que relacione `Usuarios` con `Departamentos`:

  ```
  ALTER TABLE Usuarios
  ADD CONSTRAINT fk_departamento FOREIGN KEY (DepartamentoId)
  REFERENCES Departamentos(IdDepartamento);
  ```

5. Eliminar una restricción:

Si necesitas eliminar una restricción (como un `CHECK` o `FOREIGN KEY`), puedes hacerlo con el comando `DROP CONSTRAINT`:

```
ALTER TABLE nombre_tabla
DROP CONSTRAINT nombre_restriccion;
```

Ejemplo: Eliminar una clave foránea:

```
ALTER TABLE Usuarios
DROP CONSTRAINT fk_departamento;
```

6. Renombrar una columna:

En SQL Server, no puedes renombrar una columna directamente con `ALTER TABLE`. Para hacerlo, puedes usar el comando `sp_rename`:

```
EXEC sp_rename 'nombre_tabla.nombre_columna', 'nuevo_nombre',
'COLUMN';
```

Ejemplo: Renombrar la columna `Nombre` a `NombreCompleto` en la tabla `Usuarios`:

```
EXEC sp_rename 'Usuarios.Nombre', 'NombreCompleto', 'COLUMN';
```

7. Cambiar el nombre de una tabla:

Para cambiar el nombre de una tabla completa, también usas `sp_rename`:

```
EXEC sp_rename 'nombre_tabla', 'nuevo_nombre';
```

Ejemplo: Renombrar la tabla `Usuarios` a `Clientes`:

```
EXEC sp_rename 'Usuarios', 'Clientes';
```

`ALTER` para otros objetos (procedimientos almacenados, vistas, etc.):

1. Modificar un procedimiento almacenado:

Si necesitas cambiar el código de un procedimiento almacenado, puedes usar `ALTER PROCEDURE`:

```
ALTER PROCEDURE nombre_procedimiento
AS
BEGIN
    -- Nuevas instrucciones
END;
```

Ejemplo:

```
ALTER PROCEDURE sp_ObtenerUsuarios
AS
BEGIN
    SELECT * FROM Usuarios WHERE Activo = 1;
END;
```

2. Modificar una vista:

Para modificar una vista, también se usa `ALTER VIEW`:

```
ALTER VIEW nombre_vista
AS
SELECT columnas
FROM tablas
WHERE condicion;
```

Ejemplo: Cambiar el contenido de una vista llamada `VistaActivos`:

```
ALTER VIEW VistaActivos
AS
SELECT Nombre, Apellido
FROM Usuarios
WHERE Activo = 1;
```

Resumen de comandos principales de `ALTER TABLE`:

Acción	Sintaxis
Agregar una columna	`ALTER TABLE nombre_tabla ADD nombre_columna DATATYPE;`
Modificar una columna	`ALTER TABLE nombre_tabla ALTER COLUMN nombre_columna NUEVO_DATATYPE;`
Eliminar una columna	`ALTER TABLE nombre_tabla DROP COLUMN nombre_columna;`
Agregar una restricción	`ALTER TABLE nombre_tabla ADD CONSTRAINT nombre_restriccion CHECK (...);`
Eliminar una restricción	`ALTER TABLE nombre_tabla DROP CONSTRAINT nombre_restriccion;`
Renombrar una columna	`EXEC sp_rename 'nombre_tabla.nombre_columna', 'nuevo_nombre', 'COLUMN';`
Renombrar una tabla	`EXEC sp_rename 'nombre_tabla', 'nuevo_nombre';`

Tipos de objetos que pueden modificarse con `ALTER` en SQL Server:

1. **`ALTER TABLE`**:
   - Permite modificar la estructura de una tabla existente.
   - Operaciones comunes: agregar, eliminar o modificar columnas, agregar o eliminar restricciones (como claves primarias, foráneas, `CHECK`, etc.).

   Ejemplo:

   ```
   ALTER TABLE Empleados
   ADD FechaNacimiento DATE;
   ```

2. **`ALTER VIEW`**:
   - Modifica una vista existente, que es una consulta predefinida sobre una o más tablas.

   Ejemplo:

   ```
   ALTER VIEW VistaEmpleados
   AS
   SELECT IdEmpleado, Nombre, Salario
   FROM Empleados
   WHERE Activo = 1;
   ```

3. **`ALTER PROCEDURE`** (o `ALTER PROC`):
   - Modifica un procedimiento almacenado existente.

- Puedes cambiar la lógica dentro del procedimiento sin tener que eliminarlo y volver a crearlo.

Ejemplo:

```
ALTER PROCEDURE sp_ActualizarSalario
@IdEmpleado INT,
@NuevoSalario DECIMAL(10,2)
AS
BEGIN
    UPDATE Empleados
    SET Salario = @NuevoSalario
    WHERE IdEmpleado = @IdEmpleado;
END;
```

4. **`ALTER FUNCTION`**:
   - Modifica una función definida por el usuario (puede ser de escalares o de tabla).

Ejemplo:

```
ALTER FUNCTION ObtenerEdad (@FechaNacimiento DATE)
RETURNS INT
AS
BEGIN
    RETURN DATEDIFF(YEAR, @FechaNacimiento, GETDATE());
END;
```

5. **`ALTER TRIGGER`**:
   - Modifica un **disparador** (trigger), que es un conjunto de instrucciones SQL que se ejecutan automáticamente en respuesta a eventos (`INSERT`, `UPDATE`, `DELETE`).

Ejemplo:

```
ALTER TRIGGER trg_Auditoria
ON Empleados
AFTER INSERT, DELETE
AS
BEGIN
    -- Lógica del disparador
END;
```

6. **`ALTER SCHEMA`**:
   - Cambia el esquema de un objeto o transfiere un objeto de un esquema a otro.

Ejemplo:

```
ALTER SCHEMA Ventas
TRANSFER dbo.Productos;
```

7. **`ALTER INDEX`**:

- Modifica un índice existente, por ejemplo, para reorganizarlo, reconstruirlo o deshabilitarlo.

Ejemplo:

```
ALTER INDEX idx_Empleados_Nombre ON Empleados REBUILD;
```

8. **`ALTER DATABASE`**:
   - Cambia la configuración de una base de datos, como el tamaño del archivo, el estado de recuperación, opciones de autocompletar, etc.

Ejemplo:

```
ALTER DATABASE MiBaseDeDatos
SET RECOVERY FULL;
```

9. **`ALTER LOGIN`**:
   - Modifica un inicio de sesión (login), como cambiar su contraseña o habilitar/deshabilitar el login.

Ejemplo:

```
ALTER LOGIN MiLogin WITH PASSWORD = 'NuevaContraseña';
```

10. **`ALTER USER`**:
    - Modifica un usuario de la base de datos, por ejemplo, para cambiar su asociación con un login o su esquema predeterminado.

Ejemplo:

```
ALTER USER MiUsuario WITH DEFAULT_SCHEMA = Produccion;
```

11. **`ALTER ROLE`**:
    - Modifica un rol de la base de datos, como agregar o eliminar miembros del rol.

Ejemplo:

```
ALTER ROLE db_datareader ADD MEMBER JuanPerez;
```

12. **`ALTER SEQUENCE`**:
    - Modifica una secuencia numérica definida por el usuario, que genera números secuenciales.

Ejemplo:

```
ALTER SEQUENCE NumeroFactura
RESTART WITH 100;
```

13. **`ALTER SERVICE`**:

- Modifica un servicio en el contexto de **Service Broker**.

Ejemplo:

```
ALTER SERVICE MiServicio
ON QUEUE MiCola;
```

14. **`ALTER ASSEMBLY`**:
    - Modifica un ensamblado de .NET que se haya cargado en la base de datos.

Ejemplo:

```
ALTER ASSEMBLY MiEnsamblado
WITH PERMISSION_SET = SAFE;
```

15. **`ALTER FULLTEXT INDEX`**:
    - Modifica un índice de texto completo.

Ejemplo:

```
ALTER FULLTEXT INDEX ON Empleados
ADD (Nombre);
```

16. **`ALTER CERTIFICATE`**:
    - Modifica un certificado para cifrado de datos o autenticación.

Ejemplo:

```
ALTER CERTIFICATE MiCertificado
WITH EXPIRY_DATE = '2024-12-31';
```

17. **`ALTER ASYMMETRIC KEY`**:
    - Modifica una clave asimétrica utilizada para cifrar datos.

Ejemplo:

```
ALTER ASYMMETRIC KEY MiClave
ADD COUNTER SIGNATURE BY CERTIFICATE MiCertificado;
```

18. **`ALTER SYMMETRIC KEY`**:
    - Modifica una clave simétrica utilizada para cifrar datos.

Ejemplo:

```
ALTER SYMMETRIC KEY MiClaveSimetrica
WITH ALGORITHM = AES_256;
```

19. **`ALTER AVAILABILITY GROUP`**:
    - Modifica un grupo de disponibilidad en SQL Server para alta disponibilidad y recuperación ante desastres.

Ejemplo:

```
ALTER AVAILABILITY GROUP MiGrupo
ADD DATABASE MiBaseDeDatos;
```

Tipo de datos numéricos

En **SQL Server**, existen varios tipos de datos numéricos que se utilizan para almacenar distintos tipos de números, tanto enteros como decimales, con diferentes rangos y tamaños de almacenamiento. Estos tipos de datos pueden dividirse en dos categorías principales: **enteros** y **de punto flotante** (para decimales).

1. Tipos de datos numéricos enteros:

Estos tipos almacenan números enteros (sin parte decimal).

Tipo de dato	Tamaño (bytes)	Rango de valores
`TINYINT`	1 byte	0 a 255
`SMALLINT`	2 bytes	-32,768 a 32,767
`INT`	4 bytes	-2,147,483,648 a 2,147,483,647
`BIGINT`	8 bytes	-9,223,372,036,854,775,808 a 9,223,372,036,854,775,807

Explicación de los tipos de datos enteros:

- `TINYINT`: Solo almacena números **positivos** y es útil cuando trabajas con valores pequeños (por ejemplo, puntuaciones, conteos pequeños).
- `SMALLINT`: Almacena números enteros pequeños, tanto positivos como negativos.
- `INT`: Es el tipo de dato entero más común y se usa cuando necesitas manejar una mayor cantidad de números.
- `BIGINT`: Se utiliza para almacenar enteros extremadamente grandes, muy por encima de lo que `INT` puede manejar.

---

2. Tipos de datos numéricos decimales y de precisión fija:

Estos tipos almacenan números decimales y permiten controlar la precisión (número de dígitos totales) y la escala (número de dígitos a la derecha del punto decimal).

Tipo de dato	Tamaño (bytes)	Rango de valores (aproximado)
`DECIMAL(p,s)` / `NUMERIC(p,s)`	Depende de la precisión (`p`)	De -10^38 + 1 a 10^38 - 1
`MONEY`	8 bytes	-922,337,203,685,477.5808 a 922,337,203,685,477.5807
`SMALLMONEY`	4 bytes	-214,748.3648 a 214,748.3647

Explicación de los tipos de datos decimales:

- **`DECIMAL(p,s)` y `NUMERIC(p,s)`**: Son equivalentes y se utilizan para almacenar números decimales con precisión fija. El parámetro **`p`** (precisión) define el número total de dígitos (máximo 38), y **`s`** (escala) define cuántos de esos dígitos están a la derecha del punto decimal. Por ejemplo, **`DECIMAL(5,2)`** puede almacenar hasta 5 dígitos, de los cuales 2 estarán a la derecha del punto decimal, como en **123.45**.
- **`MONEY`** y **`SMALLMONEY`**: Se utilizan específicamente para almacenar valores monetarios o financieros, con una precisión predeterminada de hasta 4 decimales.

---

3. Tipos de datos numéricos de punto flotante:

Estos tipos de datos almacenan números decimales con precisión variable, usando notación científica.

Tipo de dato	Tamaño (bytes)	Rango de valores (aproximado)
`FLOAT(n)`	4 u 8 bytes	Depende de la precisión (`n`)
`REAL`	4 bytes	-3.4028235E+38 a 3.4028235E+38

Explicación de los tipos de datos de punto flotante:

- **`FLOAT(n)`**: Se utiliza para almacenar números con **precisión variable**. El parámetro **`n`** especifica la cantidad de bits usados para representar el número:
  - Si **`n`** es entre 1 y 24, **`FLOAT`** usa 4 bytes (similar a `REAL`).
  - Si **`n`** es entre 25 y 53, **`FLOAT`** usa 8 bytes.
- **`REAL`**: Es equivalente a **`FLOAT(24)`**, y se utiliza para representar números de punto flotante de **precisión simple**.

Comparación entre decimales y punto flotante:

- **`DECIMAL`**/**`NUMERIC`**: Son más precisos para cálculos financieros o cuando necesitas control exacto sobre los decimales. Sin embargo, ocupan más espacio en comparación con los tipos de punto flotante.
- **`FLOAT`**/**`REAL`**: Son más adecuados cuando necesitas manejar un rango amplio de números y no es crucial tener precisión exacta (por ejemplo, para cálculos científicos).

DROP

En **SQL Server**, el comando **`DROP`** se utiliza para **eliminar objetos de la base de datos**. Cuando un objeto se elimina con `DROP`, se pierde de forma permanente, junto con todos los datos que contiene, si aplica. El comando `DROP` puede utilizarse con diversos tipos de objetos, desde tablas y vistas hasta restricciones y procedimientos almacenados.

Tipos de objetos que se pueden eliminar con `DROP`:

1. **`DROP TABLE`**: Elimina una tabla de la base de datos.

   ```
   DROP TABLE Empleados;
   ```

2. **`DROP VIEW`**: Elimina una vista.

   ```
   DROP VIEW VistaEmpleados;
   ```

3. **`DROP INDEX`**: Elimina un índice.

   ```
   DROP INDEX idx_Nombre ON Empleados;
   ```

4. **`DROP PROCEDURE`**: Elimina un procedimiento almacenado.

   ```
   DROP PROCEDURE sp_ObtenerEmpleados;
   ```

5. **`DROP FUNCTION`**: Elimina una función definida por el usuario.

   ```
   DROP FUNCTION ObtenerEdad;
   ```

6. **`DROP TRIGGER`**: Elimina un disparador.

   ```
   DROP TRIGGER trg_Auditoria;
   ```

7. **`DROP CONSTRAINT`**: Elimina una restricción, como una clave primaria, clave foránea, o `CHECK`.

8. **`DROP DATABASE`**: Elimina una base de datos completa.

   ```
   DROP DATABASE MiBaseDeDatos;
   ```

9. **`DROP SCHEMA`**: Elimina un esquema.

   ```
   DROP SCHEMA MiEsquema;
   ```

10. **`DROP SYNONYM`**: Elimina un sinónimo (alias de un objeto).

    ```
    DROP SYNONYM MiSinonimo;
    ```

11. **`DROP ROLE`**: Elimina un rol de seguridad.

    ```
    DROP ROLE MiRol;
    ```

12. **`DROP USER`**: Elimina un usuario de la base de datos.

    ```
    DROP USER MiUsuario;
    ```

13. **`DROP LOGIN`**: Elimina un inicio de sesión.

```
DROP LOGIN MiLogin;
```

14. **`DROP ASSEMBLY`**: Elimina un ensamblado de .NET.

```
DROP ASSEMBLY MiEnsamblado;
```

15. **`DROP TYPE`**: Elimina un tipo de dato definido por el usuario.

```
DROP TYPE MiTipo;
```

16. **`DROP FULLTEXT INDEX`**: Elimina un índice de texto completo.

```
DROP FULLTEXT INDEX ON Empleados;
```

17. **`DROP AGGREGATE`**: Elimina una función agregada definida por el usuario.

```
DROP AGGREGATE MiAgregado;
```

18. **`DROP SEQUENCE`**: Elimina una secuencia.

```
DROP SEQUENCE MiSecuencia;
```

Cuándo se usa `DROP` junto con `ALTER`:

El comando `DROP` **no se usa directamente dentro de `ALTER`**, pero se puede utilizar **después de un `ALTER`** cuando necesitas modificar un objeto y luego eliminar alguna parte de él, como una restricción, un índice o una columna. Aquí te doy algunos ejemplos de cómo se combinan:

1. **Eliminar una restricción con `ALTER`**:
   - Cuando usas `ALTER TABLE`, puedes eliminar restricciones como claves primarias, claves foráneas o restricciones `CHECK` usando **`DROP CONSTRAINT`**.
   - **Ejemplo**: Eliminar una clave foránea:

```
ALTER TABLE Empleados
DROP CONSTRAINT fk_Departamento;
```

2. **Eliminar una columna con `ALTER TABLE`**:
   - Puedes usar `ALTER TABLE` para eliminar una columna existente de una tabla.
   - **Ejemplo**: Eliminar la columna `Telefono` de la tabla `Empleados`:

```
ALTER TABLE Empleados
DROP COLUMN Telefono;
```

3. **Eliminar un índice con `DROP INDEX`**:

- Aunque `DROP INDEX` no se usa directamente dentro de un `ALTER`, es común eliminar un índice después de modificar la estructura de una tabla.
- **Ejemplo**: Eliminar un índice después de modificar una tabla:

```
DROP INDEX idx_Nombre ON Empleados;
```

4. **Eliminar una secuencia con `DROP SEQUENCE`:**
   - Puedes usar `DROP SEQUENCE` para eliminar una secuencia que se haya creado previamente para generar valores secuenciales.
   - **Ejemplo**:

```
DROP SEQUENCE MiSecuencia;
```

Resumen de los objetos que se pueden eliminar con `DROP`:

Objeto	Comando `DROP`	Ejemplo
Tabla	`DROP TABLE`	`DROP TABLE Empleados;`
Vista	`DROP VIEW`	`DROP VIEW VistaEmpleados;`
Índice	`DROP INDEX`	`DROP INDEX idx_Nombre ON Empleados;`
Procedimiento almacenado	`DROP PROCEDURE`	`DROP PROCEDURE sp_ObtenerEmpleados;`
Función	`DROP FUNCTION`	`DROP FUNCTION ObtenerEdad;`
Disparador	`DROP TRIGGER`	`DROP TRIGGER trg_Auditoria;`
Restricción	`ALTER TABLE ... DROP CONSTRAINT`	`ALTER TABLE Empleados DROP CONSTRAINT fk_Departamento;`
Base de datos	`DROP DATABASE`	`DROP DATABASE MiBaseDeDatos;`
Esquema	`DROP SCHEMA`	`DROP SCHEMA MiEsquema;`
Sinónimo	`DROP SYNONYM`	`DROP SYNONYM MiSinonimo;`
Rol	`DROP ROLE`	`DROP ROLE MiRol;`
Usuario	`DROP USER`	`DROP USER MiUsuario;`
Inicio de sesión	`DROP LOGIN`	`DROP LOGIN MiLogin;`
Ensamblado	`DROP ASSEMBLY`	`DROP ASSEMBLY MiEnsamblado;`
Tipo de dato	`DROP TYPE`	`DROP TYPE MiTipo;`
Índice de texto completo	`DROP FULLTEXT INDEX`	`DROP FULLTEXT INDEX ON Empleados;`
Función agregada	`DROP AGGREGATE`	`DROP AGGREGATE MiAgregado;`
Secuencia	`DROP SEQUENCE`	`DROP SEQUENCE MiSecuencia;`

El **Lenguaje de Manipulación de Datos (DML)** es una parte fundamental de SQL que se utiliza para interactuar con los datos almacenados en las tablas de una base de datos. A través del DML, puedes insertar, modificar, eliminar y consultar los datos en las tablas. A diferencia de los comandos **DDL** (Data Definition Language), que definen la estructura de la base de datos, los comandos **DML** operan directamente sobre los datos.

Principales comandos de DML:

1. **`SELECT`**: Se utiliza para **consultar** o **recuperar datos** de una o más tablas. Es el comando más común y flexible para obtener información de las tablas.
2. **`INSERT INTO`**: Sirve para **agregar nuevos registros** (filas) a una tabla.
3. **`UPDATE`**: Se utiliza para **modificar** datos existentes en una o más filas de una tabla.
4. **`DELETE`**: Se usa para **eliminar uno o más registros** de una tabla, basándose en una condición.

---

1. `SELECT`: Consultar datos

El comando **`SELECT`** es el más utilizado en SQL y permite **consultar** datos desde una o varias tablas. Puedes obtener datos específicos o todas las filas y columnas de una tabla.

Sintaxis básica:

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

Ejemplo:

Consultar todos los empleados con un salario mayor a 3000:

```
SELECT Nombre, Salario
FROM RecursosHumanos.Empleados
WHERE Salario > 3000;
```

---

2. `INSERT INTO`: Insertar nuevos datos

Este comando se utiliza para **insertar** nuevos registros en una tabla. Puedes especificar los valores de todas o algunas de las columnas de la tabla.

Sintaxis básica:

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

Ejemplo:

Insertar un nuevo empleado en la tabla **Empleados**:

```
INSERT INTO RecursosHumanos.Empleados (IdEmpleado, Nombre, Salario,
IdDepartamento, IdCategoria)
VALUES (1, 'Carlos López', 4000.00, 2, 1);
```

---

3. `UPDATE`: Modificar datos existentes

El comando **`UPDATE`** se utiliza para **modificar** datos existentes en una tabla. Puedes actualizar una o varias columnas de una o más filas en función de una condición.

Sintaxis básica:

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

Ejemplo:

Actualizar el salario de un empleado:

```
UPDATE RecursosHumanos.Empleados
SET Salario = 5000.00
WHERE IdEmpleado = 1;
```

---

4. `DELETE`: Eliminar datos

El comando **`DELETE`** se utiliza para **eliminar registros** de una tabla. Debes proporcionar una condición que especifique qué filas se deben eliminar.

Sintaxis básica:

```
DELETE FROM nombre_tabla
WHERE condición;
```

Ejemplo:

Eliminar al empleado con el ID 1:

```
DELETE FROM RecursosHumanos.Empleados
WHERE IdEmpleado = 1;
```

Características importantes del DML:

1. **Transacciones**: Las operaciones DML suelen formar parte de una transacción. En SQL Server, puedes agrupar comandos DML dentro de una transacción para asegurarte de que todos los cambios se realicen de manera atómica (todos o ninguno).

- **BEGIN TRANSACTION** inicia una transacción.
- **COMMIT** aplica los cambios de la transacción.
- **ROLLBACK** deshace los cambios si ocurre algún error.

2. **Condiciones `WHERE`**: Todos los comandos DML (excepto `INSERT`) pueden incluir una cláusula **`WHERE`** que especifica la condición para seleccionar las filas afectadas. Esto es esencial para evitar actualizar o eliminar todas las filas de una tabla accidentalmente.
3. **Seguridad y acceso**: Las operaciones DML están sujetas a los permisos de seguridad. Los usuarios solo pueden ejecutar comandos DML si tienen los permisos adecuados para las tablas.

INSERT

1. `INSERT INTO`: Inserción de datos en una tabla

El comando **`INSERT INTO`** se utiliza para **insertar nuevos registros** en una tabla. Puedes especificar los valores para cada columna o solo algunas columnas de la tabla, en cuyo caso el resto tomará valores predeterminados o `NULL`, si están permitidos.

Sintaxis básica de `INSERT INTO`:

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

Ejemplo básico:

Insertar un nuevo empleado en la tabla **Empleados**:

```
INSERT INTO RecursosHumanos.Empleados (IdEmpleado, Nombre, Salario,
IdDepartamento, IdCategoria)
VALUES (1, 'Carlos López', 4000.00, 2, 1);
```

2. Expresiones dentro de `INSERT`

En lugar de insertar valores literales (como `'Carlos López'` o `4000.00`), también puedes utilizar **expresiones** en el comando `INSERT INTO`. Las expresiones pueden incluir funciones, subconsultas o cálculos.

Ejemplos de expresiones dentro de `INSERT INTO`:

- **Expresión matemática**: Puedes realizar cálculos matemáticos directamente dentro del `INSERT`.

  Ejemplo:

  ```
  INSERT INTO RecursosHumanos.Empleados (IdEmpleado, Nombre,
  Salario, IdDepartamento, IdCategoria)
  ```

```
VALUES (2, 'Ana Gómez', 2000.00 + 500.00, 3, 2);
```

Aquí, el salario de **Ana Gómez** será el resultado de la suma: `2500.00`.

- **Funciones de SQL**: Puedes utilizar funciones como `GETDATE()`, `LEN()`, o cualquier función SQL estándar para generar valores dinámicos.

  Ejemplo (inserta la fecha actual para la columna `FechaContratacion`):

```
INSERT INTO RecursosHumanos.Empleados (IdEmpleado, Nombre,
Salario, IdDepartamento, IdCategoria, FechaContratacion)
VALUES (3, 'Laura Sánchez', 3500.00, 1, 3, GETDATE());
```

- **Subconsultas**: Puedes usar subconsultas dentro de un `INSERT` para insertar valores que provienen de otra tabla.

  Ejemplo (insertar empleados con el salario promedio de un departamento):

```
INSERT INTO RecursosHumanos.Empleados (IdEmpleado, Nombre,
Salario, IdDepartamento, IdCategoria)
VALUES (4, 'Miguel Fernández', (SELECT AVG(Salario) FROM
RecursosHumanos.Empleados WHERE IdDepartamento = 1), 1, 2);
```

  Aquí, se inserta un nuevo empleado con el salario promedio de los empleados del departamento 1.

3. Inserción en Vistas (`INSERT INTO` con vistas)

Es posible realizar **inserciones** en una **vista** bajo ciertas condiciones. SQL Server permite insertar datos en vistas si cumplen ciertos criterios:

- La vista debe referenciar una sola tabla directamente (no puede ser una combinación de varias tablas o una subconsulta compleja).
- Todas las columnas necesarias en la tabla subyacente deben estar disponibles en la vista.
- Las restricciones como claves primarias y foráneas deben respetarse.

Ejemplo de inserción en una vista:

Primero, creamos una vista que muestre todos los empleados de un departamento específico:

```
CREATE VIEW VistaVentas AS
SELECT IdEmpleado, Nombre, Salario, IdDepartamento
FROM RecursosHumanos.Empleados
WHERE IdDepartamento = 2;
```

Luego, podemos insertar un nuevo empleado a través de la vista:

```
INSERT INTO VistaVentas (IdEmpleado, Nombre, Salario, IdDepartamento)
VALUES (5, 'Daniela Ruiz', 3700.00, 2);
```

En este caso, la inserción se reflejará en la tabla original **`Empleados`**. Sin embargo, no puedes insertar datos en una vista que referencie múltiples tablas o utilice funciones agregadas (`SUM`, `AVG`, etc.).

4. Identificadores de tabla y vista en `INSERT INTO`

- **Identificador de tabla**: En el comando `INSERT INTO`, el identificador de la tabla es simplemente el nombre de la tabla donde deseas insertar los datos.

  Ejemplo:

  ```
  INSERT INTO RecursosHumanos.Empleados (IdEmpleado, Nombre,
  Salario, IdDepartamento)
  VALUES (6, 'Roberto Castillo', 4200.00, 3);
  ```

- **Identificador de vista**: Si estás insertando datos a través de una vista, el identificador será el nombre de la vista. Si la vista es válida para la inserción, el sistema gestionará la inserción en la tabla subyacente.

  Ejemplo:

  ```
  INSERT INTO VistaVentas (IdEmpleado, Nombre, Salario,
  IdDepartamento)
  VALUES (7, 'Victoria Suárez', 3900.00, 2);
  ```

5. Diferencias entre insertar en tablas y vistas

- **Insertar en tablas**: Tienes acceso completo a todas las columnas de la tabla, y puedes insertar valores en cualquiera de ellas, siempre y cuando respetes las restricciones (como claves foráneas o no nulos).
- **Insertar en vistas**: Solo puedes insertar en vistas si:
  - La vista se basa en una única tabla.
  - La vista no utiliza funciones de agregación ni subconsultas.
  - Las restricciones de la tabla subyacente se respetan.

En SQL, al usar el comando **`INSERT INTO`**, es crucial que el número de valores proporcionados coincida con el número de columnas especificadas en la lista de columnas. Además, debes respetar el **orden de las columnas** en la lista y proporcionar un valor para cada columna especificada.

Consideraciones importantes:

1. **Correspondencia entre columnas y valores**:
   - Los valores deben coincidir en cantidad y orden con las columnas especificadas en la instrucción `INSERT INTO`.
2. **Valores `NULL` o valores por defecto**:
   - Si una columna tiene un valor por defecto o permite valores `NULL`, no es necesario incluirla en la lista de columnas del `INSERT INTO`, y se le asignará el valor predeterminado o `NULL` si no se especifica.

3. **Inserción con todas las columnas**:
    - Si no especificas las columnas en el `INSERT INTO`, deberás proporcionar valores para **todas** las columnas de la tabla y en el mismo orden en que fueron definidas en la creación de la tabla.

Ejemplos:

Inserción especificando columnas:

```
INSERT INTO RecursosHumanos.Empleados (IdEmpleado, Nombre, Salario,
IdDepartamento)
VALUES (1, 'Carlos López', 4000.00, 2);
```

En este ejemplo, se especifican solo algunas columnas (`IdEmpleado`, `Nombre`, `Salario`, `IdDepartamento`), por lo que es importante respetar el orden y proporcionar un valor para cada una de ellas.

Inserción con todas las columnas:

Si no se especifican las columnas, debes proporcionar valores para todas las columnas definidas en la tabla, respetando el orden en el que fueron creadas:

```
INSERT INTO RecursosHumanos.Empleados
VALUES (2, 'Ana Gómez', 2500.00, 1, 2);
```

Si faltan valores o si el orden no es el correcto, SQL Server generará un error.

DELETE

Este comando es parte del **Lenguaje de Manipulación de Datos (DML)** y, al igual que los otros comandos de DML, permite modificar el contenido de las tablas.

1. ¿Qué hace el comando `DELETE`?

El comando **`DELETE`** se utiliza para eliminar filas de una tabla. Puedes eliminar una o varias filas a la vez, dependiendo de la condición que especifiques.

- Si no se utiliza una cláusula **`WHERE`**, se **eliminarán todas las filas** de la tabla, pero la estructura de la tabla permanece intacta.

2. Sintaxis básica del comando `DELETE`:

```
DELETE FROM nombre_tabla
WHERE condición;
```

3. Ejemplos básicos de `DELETE`:

Ejemplo 1: Eliminar un solo registro

Supongamos que tienes la tabla **Empleados** en la base de datos **RecursosHumanos**, y deseas eliminar a un empleado con un `IdEmpleado` específico.

```
DELETE FROM RecursosHumanos.Empleados
WHERE IdEmpleado = 1;
```

Este comando eliminará al empleado con **IdEmpleado = 1** de la tabla **Empleados**.

Ejemplo 2: Eliminar varios registros

Puedes eliminar múltiples registros utilizando una condición que afecte a varias filas. Por ejemplo, si deseas eliminar a todos los empleados cuyo salario sea menor a **3000**:

```
DELETE FROM RecursosHumanos.Empleados
WHERE Salario < 3000;
```

Este comando eliminará a todos los empleados con un salario inferior a **3000**.

Ejemplo 3: Eliminar todos los registros

Si deseas eliminar todos los registros de una tabla (pero mantener la estructura de la tabla intacta), puedes usar el comando **`DELETE`** sin una cláusula **`WHERE`**. Esto eliminará **todas las filas** de la tabla:

```
DELETE FROM RecursosHumanos.Empleados;
```

4. `DELETE` VS. `TRUNCATE`

Es importante entender la diferencia entre **`DELETE`** y **`TRUNCATE`**, ya que ambos comandos eliminan filas de una tabla, pero de maneras diferentes:

- **`DELETE`**: Elimina filas de una tabla, pero **registra** cada eliminación individualmente, lo que puede ser más lento si se eliminan muchas filas. También permite eliminar solo algunas filas usando **`WHERE`**. Además, puedes realizar **transacciones** con `DELETE`, lo que te permite **revertir los cambios** si es necesario.
- **`TRUNCATE`**: Elimina **todas las filas** de una tabla, pero lo hace de manera más rápida y eficiente, ya que **no registra** cada eliminación individualmente. No se puede usar **`WHERE`** con **`TRUNCATE`** y no puedes revertir los cambios en caso de error.

---

5. Consideraciones importantes al usar `DELETE`:

- **Cláusula `WHERE`**: Siempre es una buena práctica usar una **condición `WHERE`** cuando deseas eliminar solo ciertas filas. Si olvidas la cláusula **`WHERE`**, todas las filas de la tabla serán eliminadas.
- **Transacciones**: Si estás realizando eliminaciones críticas, es recomendable usar **transacciones** para garantizar que puedas **revertir los cambios** si algo sale mal. Por ejemplo:

```
BEGIN TRANSACTION;

DELETE FROM RecursosHumanos.Empleados
WHERE IdDepartamento = 1;

-- Si todo está correcto
COMMIT;

-- Si algo salió mal
-- ROLLBACK;
```

`BEGIN TRANSACTION` es un comando en **SQL Server** que se utiliza para iniciar una **transacción**. Una **transacción** es un conjunto de operaciones SQL que se ejecutan de manera atómica, es decir, todas las operaciones dentro de la transacción deben completarse correctamente para que los cambios se confirmen en la base de datos. Si ocurre un error en alguna operación, puedes deshacer los cambios realizados hasta ese momento utilizando **`ROLLBACK`**.

Concepto de Transacción:

Las transacciones garantizan las propiedades **ACID**:

- **A**tomidad: Todas las operaciones de la transacción se ejecutan completamente o ninguna se ejecuta.
- **C**onsistencia: La base de datos pasa de un estado consistente a otro estado consistente.
- **I**solación: Los cambios realizados en una transacción no son visibles para otras transacciones hasta que se confirmen (commit).
- **D**urabilidad: Una vez que los cambios de la transacción son confirmados, se mantienen de manera permanente.

Comandos relacionados con transacciones:

1. **`BEGIN TRANSACTION`**: Inicia una transacción.
2. **`COMMIT`**: Confirma todos los cambios realizados durante la transacción.
3. **`ROLLBACK`**: Revierte los cambios realizados durante la transacción.

Flujo básico de una transacción:

1. **`BEGIN TRANSACTION`**: Inicia la transacción.
2. Ejecutas las operaciones de SQL.
3. Si todo sale bien, **`COMMIT`** aplica los cambios de manera permanente.

4. Si hay algún error, puedes ejecutar **ROLLBACK** para deshacer los cambios.

---

6. Ejercicio práctico de DELETE

1. **Eliminar un empleado específico**: Elimina al empleado con **IdEmpleado = 3**.

```
DELETE FROM RecursosHumanos.Empleados
WHERE IdEmpleado = 3;
```

2. **Eliminar empleados de un departamento**: Elimina a todos los empleados que pertenecen al **Departamento 2**.

```
DELETE FROM RecursosHumanos.Empleados
WHERE IdDepartamento = 2;
```

3. **Eliminar empleados con un salario bajo**: Elimina a todos los empleados cuyo salario sea inferior a **3500**.

```
DELETE FROM RecursosHumanos.Empleados
WHERE Salario < 3500;
```

4. **Eliminar todos los registros**: Si deseas vaciar la tabla **Empleados** por completo, sin eliminar la estructura de la tabla:

```
DELETE FROM RecursosHumanos.Empleados;
```

UPDATE

El comando **UPDATE** en SQL se utiliza para **modificar** los datos existentes en una o más filas de una tabla. A diferencia de **INSERT**, que agrega nuevas filas, **UPDATE** se usa para cambiar los valores de columnas en filas ya existentes.

Sintaxis básica del comando UPDATE:

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

- **nombre_tabla**: Es el nombre de la tabla en la que deseas hacer la modificación.
- **SET**: Especifica las columnas que deseas actualizar y los nuevos valores que se les asignarán.
- **WHERE**: (Opcional) Especifica las condiciones bajo las cuales se actualizarán las filas. Si omites `WHERE`, todas las filas de la tabla se actualizarán.

Ejemplo básico:

Actualizar el salario de un empleado en la tabla **Empleados**.

```
UPDATE RecursosHumanos.Empleados
SET Salario = 5000.00
WHERE IdEmpleado = 1;
```

Este comando cambia el salario del empleado con **`IdEmpleado` = 1** a **5000.00**.

Actualización de múltiples columnas:

Puedes actualizar varias columnas al mismo tiempo separándolas por comas.

```
UPDATE RecursosHumanos.Empleados
SET Nombre = 'Carlos López', Salario = 5500.00
WHERE IdEmpleado = 1;
```

Este comando actualiza tanto el nombre como el salario del empleado con **`IdEmpleado` = 1**.

Actualizar múltiples filas:

Si deseas actualizar varias filas al mismo tiempo, puedes utilizar una condición en el **`WHERE`** que seleccione más de una fila.

```
UPDATE RecursosHumanos.Empleados
SET Salario = Salario * 1.10
WHERE IdDepartamento = 2;
```

Este comando incrementa en un 10% el salario de todos los empleados que pertenecen al **departamento 2**.

Actualizar todas las filas (sin `WHERE`):

Si omites la cláusula **`WHERE`**, todas las filas de la tabla serán actualizadas. Ten mucho cuidado con este uso.

```
UPDATE RecursosHumanos.Empleados
SET Salario = 3000.00;
```

Esto establecerá el salario de **todos** los empleados a **3000.00**.

Usar subconsultas en `UPDATE`:

Es posible usar una subconsulta para actualizar valores basados en el resultado de otra consulta.

```
UPDATE RecursosHumanos.Empleados
SET Salario = (SELECT AVG(Salario) FROM RecursosHumanos.Empleados)
WHERE IdDepartamento = 2;
```

Este comando actualiza el salario de todos los empleados del **departamento 2** con el salario promedio de todos los empleados.

Consideraciones importantes con `UPDATE`:

1. **Cuidado con el `WHERE`**: Si omites la cláusula `WHERE`, todas las filas de la tabla serán actualizadas. Siempre verifica las condiciones antes de ejecutar un `UPDATE`.
2. **Actualización condicional**: Es común usar subconsultas o cálculos en el `SET` para actualizar los valores de manera dinámica.
3. **Transacciones**: Si estás haciendo actualizaciones importantes, es recomendable usar **transacciones** para asegurarte de que puedas revertir los cambios en caso de error.

Ejercicio práctico de `UPDATE`:

1. **Incrementar el salario de un empleado**:

   ```
   UPDATE RecursosHumanos.Empleados
   SET Salario = Salario + 500.00
   WHERE IdEmpleado = 3;
   ```

2. **Actualizar el departamento de todos los empleados cuyo salario es mayor a 4000**:

   ```
   UPDATE RecursosHumanos.Empleados
   SET IdDepartamento = 1
   WHERE Salario > 4000;
   ```

3. **Actualizar el salario de los empleados con el salario promedio del departamento**:

   ```
   UPDATE RecursosHumanos.Empleados
   SET Salario = (SELECT AVG(Salario) FROM
   RecursosHumanos.Empleados WHERE IdDepartamento = 1)
   WHERE IdDepartamento = 1;
   ```

Actualizar todas las filas:

```
UPDATE RecursosHumanos.Empleados
```

```
SET Salario = 3000.00;
```

Situaciones comunes para usar `NULL` en el `SET` de un `UPDATE`:

1. **Reiniciar un valor**: Si un campo tiene un valor y deseas eliminarlo para que quede sin datos, puedes usar `NULL` en el `SET`.
2. **Asignar un valor desconocido**: En ocasiones, puede haber datos que no están disponibles temporalmente, por lo que puedes establecer la columna en `NULL` para indicar que el valor es desconocido.

Sintaxis básica para usar `NULL` en `SET`:

```
UPDATE nombre_tabla
SET columna = NULL
WHERE condición;
```

- **`columna = NULL`**: Esto establece el valor de la columna como `NULL` en las filas que cumplan con la condición del `WHERE`.

SELECT

`SELECT` es uno de los comandos más importantes en SQL, utilizado para **consultar datos** de una o varias tablas en una base de datos. A través de **`SELECT`**, puedes recuperar información de las tablas de manera flexible y filtrada según tus necesidades.

1. Sintaxis básica de `SELECT`:

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

- **`SELECT`**: Especifica las columnas que deseas consultar.
- **`FROM`**: Especifica la tabla desde la que se recuperan los datos.
- **`WHERE`**: (Opcional) Filtra las filas que cumplen una determinada condición.

Si deseas consultar **todas** las columnas de una tabla, puedes usar **`*`** en lugar de listar las columnas:

```
SELECT * FROM nombre_tabla;
```

2. Ejemplos básicos de `SELECT`:

Ejemplo 1: Seleccionar todas las columnas de una tabla

```
SELECT * FROM RecursosHumanos.Empleados;
```

Este comando selecciona y devuelve **todas las filas y columnas** de la tabla **Empleados**.

Ejemplo 2: Seleccionar columnas específicas

Si solo deseas consultar algunas columnas de la tabla:

```
SELECT Nombre, Salario
FROM RecursosHumanos.Empleados;
```

Este comando devuelve solo las columnas **Nombre** y **Salario** de la tabla **Empleados**.

Ejemplo 3: Usar `WHERE` para filtrar resultados

Puedes filtrar los resultados usando la cláusula **`WHERE`**, que especifica una condición.

```
SELECT Nombre, Salario
FROM RecursosHumanos.Empleados
WHERE Salario > 3000;
```

Este comando selecciona a los empleados cuyo **Salario** es mayor que **3000**.

---

3. Uso de operadores en `SELECT` con `WHERE`

Operadores de comparación:

- **`=`**: Igual
- **`<>`** o **`!=`**: Distinto
- **`>`**, **`<`**, **`>=`**, **`<=`**: Mayor, menor, mayor o igual, menor o igual.

Ejemplo con operadores de comparación:

```
SELECT Nombre, Salario
FROM RecursosHumanos.Empleados
WHERE Salario >= 3500;
```

Operadores lógicos:

- **`AND`**: Ambas condiciones deben ser verdaderas.
- **`OR`**: Al menos una de las condiciones debe ser verdadera.
- **`NOT`**: La condición debe ser falsa.

Ejemplo con operadores lógicos:

Seleccionar empleados del departamento 2 con un salario mayor a **3000**:

```
SELECT Nombre, Salario
FROM RecursosHumanos.Empleados
WHERE IdDepartamento = 2 AND Salario > 3000;
```

4. Ordenar resultados con `ORDER BY`

Puedes **ordenar los resultados** de una consulta utilizando `ORDER BY`. Por defecto, el orden es ascendente (**ASC**), pero también puedes ordenar de forma descendente (**DESC**).

Ejemplo de `ORDER BY`:

```
SELECT Nombre, Salario
FROM RecursosHumanos.Empleados
ORDER BY Salario DESC;
```

Este comando muestra los empleados ordenados por **Salario** en orden descendente.

---

5. Limitar el número de resultados con `TOP`

Puedes limitar el número de filas devueltas usando la cláusula `TOP` en SQL Server.

Ejemplo con `TOP`:

```
SELECT TOP 5 Nombre, Salario
FROM RecursosHumanos.Empleados
ORDER BY Salario DESC;
```

Esto selecciona los 5 empleados con los salarios más altos.

---

6. Uso de funciones agregadas

Las **funciones agregadas** permiten realizar cálculos sobre varias filas de una tabla y devolver un solo valor.

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

Ejemplo de funciones agregadas:

```
SELECT AVG(Salario) AS SalarioPromedio
FROM RecursosHumanos.Empleados;
```

Esto calcula el **salario promedio** de todos los empleados.

---

7. Agrupar resultados con `GROUP BY`

La cláusula **`GROUP BY`** se utiliza para agrupar filas que tienen valores iguales en columnas específicas y realizar funciones agregadas en esas agrupaciones.

Ejemplo de `GROUP BY`:

```
SELECT IdDepartamento, AVG(Salario) AS SalarioPromedio
FROM RecursosHumanos.Empleados
GROUP BY IdDepartamento;
```

Este comando agrupa a los empleados por **departamento** y calcula el **salario promedio** por departamento.

---

8. Filtrar grupos con `HAVING`

La cláusula **`HAVING`** se utiliza para filtrar grupos después de aplicar `GROUP BY`, generalmente cuando se usa con funciones agregadas.

Ejemplo de `HAVING`:

```
SELECT IdDepartamento, AVG(Salario) AS SalarioPromedio
FROM RecursosHumanos.Empleados
GROUP BY IdDepartamento
HAVING AVG(Salario) > 4000;
```

Este comando devuelve solo los departamentos cuyo **salario promedio** es mayor que **4000**.

---

9. Subconsultas en `SELECT`

Una subconsulta es una consulta dentro de otra consulta, y se puede usar para devolver valores utilizados en la consulta principal.

Ejemplo de subconsulta:

```
SELECT Nombre, Salario
FROM RecursosHumanos.Empleados
WHERE Salario > (SELECT AVG(Salario) FROM RecursosHumanos.Empleados);
```

Este comando selecciona a los empleados cuyo salario es mayor que el salario promedio de todos los empleados.

Ejercicio práctico con `SELECT`:

1. **Seleccionar empleados del departamento de ventas**:

```
SELECT Nombre, Salario
FROM RecursosHumanos.Empleados
WHERE IdDepartamento = 2;
```

2. **Seleccionar los 3 empleados con mayor salario**:

```
SELECT TOP 3 Nombre, Salario
FROM RecursosHumanos.Empleados
ORDER BY Salario DESC;
```

3. **Calcular el salario promedio de los empleados**:

```
SELECT AVG(Salario) AS SalarioPromedio
FROM RecursosHumanos.Empleados;
```

Opciones en la cláusula `SELECT`:

1. **Columnas específicas o todas las columnas**:
   - Puedes seleccionar columnas específicas o todas las columnas de una tabla.

   Ejemplo:

```
SELECT columna1, columna2
SELECT *
```

2. **`DISTINCT`**:
   - Elimina los duplicados y devuelve solo filas únicas.

   Ejemplo:

```
SELECT DISTINCT columna1
```

3. **Expresiones**:
   - Puedes utilizar expresiones matemáticas, lógicas, funciones, concatenaciones, etc.

   Ejemplo:

```
SELECT columna1 * 2, columna2 + columna3
```

4. **Alias con `AS`**:
   - Puedes asignar nombres (alias) a las columnas o expresiones calculadas utilizando **`AS`**.

   Ejemplo:

```
SELECT columna1 AS AliasColumna1, columna2 * 2 AS DobleColumna2
```

Las **funciones de fila** en SQL Server, como `ROW_NUMBER()`, `RANK()`, y `DENSE_RANK()`, se utilizan para asignar un número o rango a cada fila en un conjunto de resultados. Son útiles para la **paginación**, **ranking** o cuando quieres numerar filas en función de un criterio específico.

1. `ROW_NUMBER()`:

- `ROW_NUMBER()` asigna un **número de fila secuencial** único a cada fila en el conjunto de resultados, basado en el orden especificado en la cláusula `ORDER BY`.
- Cada fila tiene un número diferente, independientemente de si los valores de las columnas están repetidos.

Sintaxis:

```
ROW_NUMBER() OVER (ORDER BY columna)
```

Ejemplo con `ROW_NUMBER()`:

Imaginemos que tienes una tabla de **empleados** y deseas numerar a los empleados según su **salario**, ordenando de mayor a menor:

```
SELECT Nombre, Salario,
       ROW_NUMBER() OVER (ORDER BY Salario DESC) AS NumeroFila
FROM Empleados;
```

Salida (ordenados por salario de mayor a menor):

Nombre	Salario	NumeroFila
Juan	5000	1
Ana	4500	2
Carlos	4500	3
Pedro	4000	4

- Aquí, `ROW_NUMBER()` asigna un número de fila secuencial (1, 2, 3, 4...) basado en el salario en orden descendente.
- Si dos empleados tienen el mismo salario (como Ana y Carlos), ambos tienen números de fila diferentes.

---

2. `RANK()`:

- `RANK()` también asigna un rango a cada fila según el `ORDER BY`, pero a diferencia de `ROW_NUMBER()`, asigna el **mismo rango a filas con valores iguales**. Después de los valores repetidos, salta al siguiente número de rango.

- Es útil para saber qué posición ocupa cada fila en función de un valor, sin importar si hay empates.

Sintaxis:

```
RANK() OVER (ORDER BY columna)
```

Ejemplo con `RANK()`:

Si ejecutas la misma consulta, pero usando **`RANK()`** en lugar de **`ROW_NUMBER()`**:

```
SELECT Nombre, Salario,
       RANK() OVER (ORDER BY Salario DESC) AS Rango
FROM Empleados;
```

Salida:

Nombre	Salario	Rango
Juan	5000	1
Ana	4500	2
Carlos	4500	2
Pedro	4000	4

- Los empleados **Ana** y **Carlos** tienen el mismo salario, por lo que tienen el mismo **rango (2)**.
- El siguiente rango es **4**, porque se salta el número 3 debido al empate en el rango 2.

---

3. `DENSE_RANK()`:

- **`DENSE_RANK()`** es similar a **`RANK()`**, pero **no salta números** cuando hay empates. Los empleados con el mismo valor comparten el mismo rango, pero el siguiente valor asignado no salta números.

Sintaxis:

```
DENSE_RANK() OVER (ORDER BY columna)
```

Ejemplo con `DENSE_RANK()`:

```
SELECT Nombre, Salario,
       DENSE_RANK() OVER (ORDER BY Salario DESC) AS RangoDenso
FROM Empleados;
```

Salida:

Nombre	Salario	RangoDenso
Juan	5000	1
Ana	4500	2
Carlos	4500	2
Pedro	4000	3

- Al igual que en **`RANK()`**, Ana y Carlos tienen el mismo rango (2), pero **`DENSE_RANK()`** no salta números. El siguiente valor es el **3**, no el 4.

Diferencias clave entre `ROW_NUMBER()`, `RANK()`, y `DENSE_RANK()`:

1. **`ROW_NUMBER()`**:
   - Asigna números secuenciales sin importar si los valores de las columnas son iguales.
   - No considera empates.
2. **`RANK()`**:
   - Asigna el mismo rango a filas con valores iguales.
   - Salta números después de filas con valores duplicados.
3. **`DENSE_RANK()`**:
   - Asigna el mismo rango a filas con valores iguales.
   - No salta números, continúa secuencialmente.

Uso práctico en paginación:

Las funciones de fila como **`ROW_NUMBER()`** son útiles para implementar paginación de resultados, lo que permite dividir grandes conjuntos de resultados en páginas más pequeñas.

Ejemplo: Paginación con `ROW_NUMBER()`:

```
WITH EmpleadosNumerados AS (
    SELECT Nombre, Salario,
           ROW_NUMBER() OVER (ORDER BY Salario DESC) AS NumeroFila
    FROM Empleados
)
SELECT Nombre, Salario
FROM EmpleadosNumerados
WHERE NumeroFila BETWEEN 11 AND 20;
```

Este ejemplo selecciona los empleados entre las filas **11** y **20**, útil para mostrar datos en una página de resultados.

Resumen:

- **`ROW_NUMBER()`**: Asigna un número de fila secuencial sin importar duplicados.
- **`RANK()`**: Asigna el mismo rango a valores duplicados, pero salta números en los rangos.
- **`DENSE_RANK()`**: Asigna el mismo rango a duplicados sin saltar números en los rangos.

Estas funciones son extremadamente útiles para **numerar filas**, **clasificar resultados** y **paginación** en SQL Server.

Ejemplo con `ROW_NUMBER()` y un `ORDER BY` externo diferente:

Si quieres numerar las filas por **salario** pero mostrar el resultado ordenado por **nombre**, puedes hacerlo así:

```
SELECT Nombre, Salario,
       ROW_NUMBER() OVER (ORDER BY Salario DESC) AS NumeroFila
FROM Empleados
ORDER BY Nombre;
```

- **`ROW_NUMBER() OVER (ORDER BY Salario DESC)`**: Numerará las filas de acuerdo con el salario en orden descendente.
- **`ORDER BY Nombre`**: Organiza los resultados por nombre en orden ascendente en el resultado final, sin afectar la numeración.

FROM

La **cláusula `FROM`** en SQL es fundamental, ya que **especifica la tabla o tablas** desde las cuales se extraen los datos para una consulta. Se encuentra inmediatamente después de la cláusula `SELECT` y es imprescindible en la mayoría de las consultas.

La cláusula **`FROM`** también permite trabajar con combinaciones de tablas a través de **uniones (`JOIN`)**, **subconsultas**, y **alias de tablas**.

Sintaxis básica de la cláusula `FROM`:

```
SELECT columna1, columna2
FROM nombre_tabla;
```

- **`nombre_tabla`**: Es la tabla de la que se extraen los datos.

Funciones y características clave de la cláusula `FROM`:

1. **Especificar una o más tablas**.
2. **Uniones (`JOIN`)** entre tablas para combinar datos.
3. **Subconsultas** en la cláusula `FROM`.
4. **Alias de tablas** para abreviar nombres largos.
5. **Referenciar vistas** o tablas derivadas.

---

1. Especificar una o más tablas en `FROM`:

En la consulta más básica, la cláusula **`FROM`** especifica una sola tabla de la cual se extraen las columnas listadas en el **`SELECT`**.

Ejemplo básico:

```
SELECT Nombre, Salario
FROM Empleados;
```

Este ejemplo consulta las columnas **`Nombre`** y **`Salario`** de la tabla **`Empleados`**.

2. Usar `JOIN` en la cláusula `FROM`:

Puedes combinar datos de varias tablas utilizando diferentes tipos de **uniones (`JOIN`)** en la cláusula **`FROM`**. Esta es una forma común de realizar operaciones de reunión entre tablas.

Ejemplo con `INNER JOIN`:

```
SELECT Empleados.Nombre, Departamentos.NombreDepartamento
FROM Empleados
```

```
INNER JOIN Departamentos ON Empleados.IdDepartamento =
Departamentos.IdDepartamento;
```

En este ejemplo:

- **`FROM Empleados INNER JOIN Departamentos`** combina ambas tablas basándose en la relación entre **`IdDepartamento`**.

3. Subconsultas en la cláusula `FROM`:

Puedes incluir una **subconsulta** en la cláusula **`FROM`**, que crea una tabla temporal o derivada para su uso en la consulta principal.

Ejemplo con subconsulta:

```
SELECT Nombre, SalarioPromedio
FROM (SELECT IdDepartamento, AVG(Salario) AS SalarioPromedio
      FROM Empleados
      GROUP BY IdDepartamento) AS SalariosPorDepartamento;
```

En este caso:

- La subconsulta en **`FROM`** agrupa a los empleados por departamento y calcula el salario promedio para cada departamento.
- La consulta principal selecciona los **nombres** de los empleados y los compara con el **salario promedio** de su departamento.

4. Alias de tablas en la cláusula `FROM`:

Usar alias para tablas te permite acortar los nombres de las tablas y mejorar la legibilidad de la consulta, especialmente cuando trabajas con múltiples tablas o subconsultas.

Ejemplo con alias:

```
SELECT e.Nombre, d.NombreDepartamento
FROM Empleados e
INNER JOIN Departamentos d ON e.IdDepartamento = d.IdDepartamento;
```

Aquí, las tablas **`Empleados`** y **`Departamentos`** se renombraron con los alias **`e`** y **`d`**, respectivamente, para abreviar los nombres de las tablas.

5. Referenciar vistas o tablas derivadas en `FROM`:

La cláusula **`FROM`** también puede referenciar **vistas** o **tablas derivadas** que se han creado previamente.

Ejemplo con vista:

```
SELECT Nombre, Salario
```

```
FROM VistaEmpleadosActivos;
```

En este caso, **`VistaEmpleadosActivos`** es una vista previamente definida que puede representar una consulta compleja sobre la tabla **`Empleados`**.

Ejemplos avanzados de la cláusula `FROM`:

Consulta con múltiples uniones (`JOIN`):

```
SELECT e.Nombre, p.NombreProyecto, d.NombreDepartamento
FROM Empleados e
INNER JOIN Proyectos p ON e.IdProyecto = p.IdProyecto
LEFT JOIN Departamentos d ON e.IdDepartamento = d.IdDepartamento;
```

- Esta consulta combina tres tablas: **`Empleados`**, **`Proyectos`**, y **`Departamentos`** utilizando `INNER JOIN` y `LEFT JOIN`.

Subconsulta compleja en `FROM`:

```
SELECT e.Nombre, e.Salario, s.PromedioSalario
FROM Empleados e
INNER JOIN (SELECT IdDepartamento, AVG(Salario) AS PromedioSalario
            FROM Empleados
            GROUP BY IdDepartamento) s
ON e.IdDepartamento = s.IdDepartamento;
```

- Aquí, la subconsulta dentro de `FROM` calcula el salario promedio por departamento y la tabla derivada es utilizada en la consulta principal para obtener el salario de cada empleado y el salario promedio de su departamento.

WHERE

La **cláusula `WHERE`** en SQL se utiliza para filtrar registros de una consulta, actualización o eliminación, basándose en una condición específica. Es una parte crucial de las consultas SQL porque permite seleccionar solo los registros que cumplen con ciertos criterios, lo que hace que las consultas sean más eficientes y específicas.

Uso de la cláusula `WHERE`:

La cláusula `WHERE` se usa en las siguientes sentencias SQL:

- **`SELECT`**: Para filtrar las filas devueltas en una consulta.
- **`UPDATE`**: Para actualizar solo las filas que cumplen con una condición.
- **`DELETE`**: Para eliminar filas específicas.
- **`INSERT INTO ... SELECT`**: Para filtrar las filas que se van a insertar.

Sintaxis básica:

```
SELECT columna1, columna2
FROM tabla
WHERE condición;
```

Ejemplos de uso de la cláusula `WHERE`:

1. Filtrar registros con `SELECT`:

Supongamos que tienes una tabla de **empleados** y quieres seleccionar solo los empleados cuyo salario sea mayor que 3500.

```
SELECT Nombre, Apellido, Salario
FROM Empleados
WHERE Salario > 3500;
```

Este comando seleccionará a todos los empleados que tienen un salario superior a 3500.

2. Actualizar registros con `UPDATE`:

Supongamos que quieres dar un aumento de salario del 10% a los empleados que ganan menos de 3000.

```
UPDATE Empleados
SET Salario = Salario * 1.10
WHERE Salario < 3000;
```

Este comando actualizará el salario solo de los empleados que ganan menos de 3000.

3. Eliminar registros con `DELETE`:

Si quieres eliminar todos los empleados del departamento de "Ventas":

```
DELETE FROM Empleados
WHERE IdDepartamento = 1;  -- Suponiendo que el Id 1 es el de Ventas
```

Este comando eliminará solo a los empleados que pertenecen al departamento con Id 1 (Ventas).

Condiciones en la cláusula `WHERE`:

La condición en la cláusula `WHERE` puede incluir:

- **Operadores de comparación**: =, >, <, >=, <=, <> (distinto).
- **Operadores lógicos**: `AND`, `OR`, `NOT`.
- **Operadores de rango**: `BETWEEN ... AND`.
- **Patrones de texto**: `LIKE`.

- **Valores nulos**: `IS NULL`, `IS NOT NULL`.

Operadores comunes:

1. **Uso de `AND`**: Para combinar múltiples condiciones:

   ```
   SELECT Nombre, Apellido
   FROM Empleados
   WHERE Salario > 3500 AND IdDepartamento = 2;
   ```

   Selecciona empleados con salario mayor que 3500 **y** que pertenecen al departamento con Id 2.

2. **Uso de `OR`**: Para especificar que solo una de varias condiciones debe cumplirse:

   ```
   SELECT Nombre, Apellido
   FROM Empleados
   WHERE Salario < 3000 OR IdDepartamento = 1;
   ```

   Selecciona empleados cuyo salario sea menor a 3000 **o** que pertenecen al departamento con Id 1 (Ventas).

3. **Uso de `LIKE`**: Para realizar coincidencias de patrones en una cadena de texto. El símbolo `%` representa cualquier cadena de caracteres.

   ```
   SELECT Nombre, Apellido
   FROM Empleados
   WHERE Nombre LIKE 'A%';  -- Selecciona los empleados cuyo nombre
   empieza con 'A'
   ```

   Selecciona empleados cuyo nombre comience con la letra "A".

4. **Uso de `BETWEEN`**: Para seleccionar valores dentro de un rango.

   ```
   SELECT Nombre, Apellido, Salario
   FROM Empleados
   WHERE Salario BETWEEN 3000 AND 5000;
   ```

   Selecciona empleados con un salario entre 3000 y 5000.

5. **Uso de `IS NULL` y `IS NOT NULL`**: Para trabajar con valores nulos.

   ```
   SELECT Nombre, Apellido
   FROM Empleados
   WHERE Telefono IS NULL;  -- Selecciona empleados cuyo campo
   "Telefono" no está definido
   ```

Ejemplo práctico:

Imaginemos que tienes una tabla de **Ventas** y quieres consultar solo aquellas ventas realizadas en marzo de 2023 por empleados que pertenecen al departamento de "Ventas".

```
SELECT e.Nombre, v.FechaVenta, v.Cantidad
FROM Ventas v
JOIN Empleados e ON v.IdEmpleado = e.IdEmpleado
JOIN Departamentos d ON e.IdDepartamento = d.IdDepartamento
WHERE v.FechaVenta BETWEEN '2023-03-01' AND '2023-03-31'
AND d.NombreDepartamento = 'Ventas';
```

Este comando filtra las ventas de marzo de 2023 donde los empleados pertenecen al departamento de **Ventas**.

GROUP BY

La **cláusula `GROUP BY`** en SQL se utiliza para **agrupar filas** que tienen valores comunes en una o más columnas, permitiendo aplicar **funciones de agregación** como `COUNT()`, `SUM()`, `AVG()`, `MAX()`, `MIN()`, etc. Esta cláusula es útil cuando deseas realizar cálculos agregados sobre grupos de datos, en lugar de filas individuales.

Sintaxis básica:

```
SELECT columna1, columna2, función_agregada(columna)
FROM tabla
WHERE condición
GROUP BY columna1, columna2;
```

- **`columna1, columna2`**: Las columnas por las cuales se agrupan los datos.
- **`función_agregada(columna)`**: Las funciones de agregación que se aplican a las columnas.

Ejemplo de uso de `GROUP BY`:

Supongamos que tienes una tabla **Ventas** con las siguientes columnas:

IdVenta	IdEmpleado	IdProducto	Cantidad	FechaVenta
1	1	101	2	2023-03-15
2	2	102	5	2023-03-16
3	1	103	1	2023-03-17
4	2	101	3	2023-03-18

Si quieres saber cuántas ventas hizo cada empleado, puedes agrupar los datos por **IdEmpleado** y contar las ventas que hizo cada uno:

```
SELECT IdEmpleado, COUNT(IdVenta) AS TotalVentas
FROM Ventas
GROUP BY IdEmpleado;
```

Resultado:

IdEmpleado	TotalVentas
1	2
2	2

- La consulta agrupa los registros por el **IdEmpleado**, y luego cuenta el número de ventas (columnas **IdVenta**) que realizó cada empleado.

Funciones de agregación con `GROUP BY`:

La cláusula `GROUP BY` se suele usar en combinación con funciones de agregación como:

- **`COUNT()`**: Cuenta el número de filas en cada grupo.
- **`SUM()`**: Suma los valores en cada grupo.
- **`AVG()`**: Calcula el promedio de los valores en cada grupo.
- **`MAX()`**: Encuentra el valor máximo en cada grupo.
- **`MIN()`**: Encuentra el valor mínimo en cada grupo.

Ejemplo con varias funciones de agregación:

Si quieres saber cuántas ventas hizo cada empleado, la suma total de las cantidades vendidas y el promedio de las cantidades vendidas, puedes usar:

```
SELECT IdEmpleado, COUNT(IdVenta) AS TotalVentas, SUM(Cantidad) AS
TotalCantidad, AVG(Cantidad) AS PromedioCantidad
FROM Ventas
GROUP BY IdEmpleado;
```

Resultado:

IdEmpleado	TotalVentas	TotalCantidad	PromedioCantidad
1	2	3	1.5
2	2	8	4.0

Cláusula `GROUP BY` con varias columnas:

También puedes agrupar por más de una columna. Por ejemplo, si quieres ver cuántas ventas hizo cada empleado **para cada producto**:

```
SELECT IdEmpleado, IdProducto, COUNT(IdVenta) AS TotalVentas
FROM Ventas
GROUP BY IdEmpleado, IdProducto;
```

Resultado:

IdEmpleado	IdProducto	TotalVentas
1	101	1
1	103	1
2	101	1
2	102	1

- Aquí, los resultados están agrupados por el **IdEmpleado** y **IdProducto**, mostrando cuántas ventas de cada producto hizo cada empleado.

Uso de `HAVING` con `GROUP BY`:

La cláusula `HAVING` se utiliza para filtrar los resultados después de agruparlos, similar a cómo se usa `WHERE` para filtrar antes del agrupamiento.

Ejemplo: Si quieres mostrar solo los empleados que realizaron más de 1 venta:

```
SELECT IdEmpleado, COUNT(IdVenta) AS TotalVentas
FROM Ventas
GROUP BY IdEmpleado
HAVING COUNT(IdVenta) > 1;
```

Resultado:

IdEmpleado	TotalVentas
1	2
2	2

Reglas importantes sobre `GROUP BY`:

1. **Las columnas que no usan una función de agregación** deben aparecer en la cláusula `GROUP BY`. Es decir, solo puedes seleccionar columnas que estén en el `GROUP BY` o que sean calculadas mediante una función de agregación.
2. **`GROUP BY` siempre se ejecuta después de `WHERE`** y antes de `HAVING`.
3. **No puedes usar `GROUP BY` directamente con columnas calculadas** en la cláusula `SELECT` (a menos que uses un alias o la función de agregación).

Orden de procesamiento de una consulta SQL:

1. **FROM**: Se seleccionan las tablas involucradas.
2. **WHERE**: Se filtran las filas basadas en las condiciones.
3. **GROUP BY**: Se agrupan los datos.
4. **HAVING**: Se filtran los resultados agrupados.

5. **SELECT**: Se seleccionan y calculan las columnas.
6. **ORDER BY**: Se ordenan los resultados.

ORDER BY

La cláusula **`ORDER BY`** en SQL se utiliza para ordenar los resultados de una consulta **`SELECT`** en **orden ascendente** o **descendente**. Es muy útil cuando necesitas que los resultados se presenten de una forma específica, como ordenar por una columna numérica, alfabética o por fechas.

Sintaxis básica:

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

- **`columna1, columna2, ...`**: Las columnas por las cuales se desea ordenar.
- **`ASC`**: Orden **ascendente** (de menor a mayor o de A a Z). Es el valor predeterminado, por lo que si no lo especificas, SQL asume que el orden es ascendente.
- **`DESC`**: Orden **descendente** (de mayor a menor o de Z a A).

Ejemplo de uso básico de `ORDER BY`:

Supongamos que tienes una tabla de **empleados**:

IdEmpleado	Nombre	Apellido	Salario
1	Juan	Pérez	3500
2	Ana	García	4000
3	Luis	Ramírez	3200
4	María	González	4500

Si quieres ordenar a los empleados por **salario de menor a mayor**, puedes usar `ORDER BY` en la columna `Salario`:

```
SELECT Nombre, Apellido, Salario
FROM Empleados
ORDER BY Salario ASC;
```

Resultado:

Nombre	Apellido	Salario
Luis	Ramírez	3200
Juan	Pérez	3500
Ana	García	4000

Nombre	Apellido	Salario
María	González	4500

Ejemplo de `ORDER BY` en orden descendente:

Si quieres ver a los empleados ordenados por salario de mayor a menor, puedes usar `DESC`:

```
SELECT Nombre, Apellido, Salario
FROM Empleados
ORDER BY Salario DESC;
```

Resultado:

Nombre	Apellido	Salario
María	González	4500
Ana	García	4000
Juan	Pérez	3500
Luis	Ramírez	3200

Ordenar por más de una columna:

Puedes ordenar por varias columnas. Por ejemplo, si dos empleados tienen el mismo salario, puedes decidir ordenar adicionalmente por **Apellido** en orden ascendente.

Ejemplo:

```
SELECT Nombre, Apellido, Salario
FROM Empleados
ORDER BY Salario DESC, Apellido ASC;
```

Resultado:

Nombre	Apellido	Salario
María	González	4500
Ana	García	4000
Juan	Pérez	3500
Luis	Ramírez	3200

Aquí, los empleados están primero ordenados por **Salario** en orden descendente, y luego por **Apellido** en orden ascendente si tienen el mismo salario.

Ordenar por posición de columna:

En lugar de usar los nombres de las columnas, también puedes ordenar usando el **número de la posición** de la columna en la lista de selección. Por ejemplo, si en tu consulta seleccionas tres columnas, puedes referirte a ellas como `1`, `2`, `3` para ordenarlas.

Ejemplo con posición de columna:

```
SELECT Nombre, Apellido, Salario
FROM Empleados
ORDER BY 3 DESC;  -- Ordena por la tercera columna (Salario) en orden descendente
```

Resultado:

Nombre	Apellido	Salario
María	González	4500
Ana	García	4000
Juan	Pérez	3500
Luis	Ramírez	3200

Usar `ORDER BY` con alias:

Puedes ordenar por un alias de una columna definida en el `SELECT` usando la cláusula `ORDER BY`.

Ejemplo con alias:

```
SELECT Nombre, Apellido, Salario * 12 AS SalarioAnual
FROM Empleados
ORDER BY SalarioAnual DESC;
```

Resultado:

Nombre	Apellido	SalarioAnual
María	González	54000
Ana	García	48000
Juan	Pérez	42000
Luis	Ramírez	38400

Aquí, la columna **`SalarioAnual`** es calculada en el `SELECT` y luego se utiliza en el `ORDER BY` para ordenar los resultados en orden descendente.

Combinación con otras cláusulas:

- **`GROUP BY`**: Después de agrupar los resultados con `GROUP BY`, puedes usar `ORDER BY` para ordenar los grupos.

- **`LIMIT` o `TOP`**: En algunos sistemas de bases de datos como SQL Server o MySQL, puedes combinar `ORDER BY` con `LIMIT` (en MySQL) o `TOP` (en SQL Server) para devolver solo un número limitado de resultados ordenados.

Ejemplo en SQL Server con `TOP`:

```
SELECT TOP 5 Nombre, Apellido, Salario
FROM Empleados
ORDER BY Salario DESC;
```

Este ejemplo devolverá solo los **5 empleados con mayor salario**.

VISTAS

Las **vistas** en SQL son una herramienta poderosa que permite a los usuarios simplificar consultas, mejorar la seguridad y crear abstracciones sobre los datos. A continuación, te explicaré los aspectos fundamentales de las vistas.

¿Qué es una vista?

Una **vista** es una consulta guardada que actúa como una tabla virtual. No contiene datos en sí, sino que muestra los resultados de una consulta en el momento en que se accede a la vista. Las vistas se utilizan para simplificar consultas complejas, limitar el acceso a ciertas partes de los datos o dar una estructura diferente a la información.

Sintaxis para crear una vista:

```
CREATE VIEW nombre_vista AS
SELECT columnas
FROM tabla
WHERE condición;
```

Ejemplo básico de una vista:

Supongamos que tienes una tabla de **empleados** y quieres crear una vista que solo muestre a los empleados que trabajan en el departamento de ventas:

```
CREATE VIEW VistaVentas AS
SELECT Nombre, Apellido, Salario
FROM Empleados
WHERE Departamento = 'Ventas';
```

Esta vista te permite hacer consultas más simples a la tabla **Empleados** en futuras consultas, simplemente accediendo a **VistaVentas**.

Consultando una vista:

Las vistas se consultan igual que las tablas:

```
SELECT * FROM VistaVentas;
```

Esto devolverá todos los empleados del departamento de ventas.

Ventajas de las vistas:

1. **Simplicidad**: Puedes crear una vista para simplificar una consulta compleja y luego usar esa vista en lugar de escribir la consulta completa cada vez.

   Ejemplo: Si tienes una consulta compleja que une varias tablas, puedes encapsularla en una vista.

   ```
   CREATE VIEW VistaSalarios AS
   SELECT e.Nombre, e.Apellido, d.NombreDepartamento, s.Salario
   FROM Empleados e
   JOIN Departamentos d ON e.IdDepartamento = d.IdDepartamento
   JOIN Salarios s ON e.IdEmpleado = s.IdEmpleado;
   ```

   Luego, puedes consultar la vista:

   ```
   SELECT * FROM VistaSalarios;
   ```

2. **Seguridad**: Las vistas permiten restringir el acceso a ciertas columnas de las tablas originales. Por ejemplo, podrías permitir a un usuario acceder a una vista que muestre solo información general sobre los empleados, sin permitirle ver datos sensibles como los salarios.
3. **Reutilización**: Puedes reutilizar una vista en múltiples consultas en lugar de repetir la misma consulta compleja varias veces.
4. **Abstracción**: Las vistas permiten cambiar la estructura de la base de datos subyacente sin afectar a los usuarios o aplicaciones que dependen de esa vista.

Tipos de Vistas en SQL

1. Vistas simples:

Son las vistas más comunes. Una **vista simple** está basada en una única tabla y no tiene funciones agregadas, `GROUP BY`, o `DISTINCT`. Estas vistas a menudo son **actualizables** si cumplen ciertas condiciones (que explicaremos más adelante).

Ejemplo:

```
CREATE VIEW VistaEmpleados AS
SELECT Nombre, Apellido, Salario
FROM Empleados
WHERE Departamento = 'Ventas';
```

2. Vistas complejas:

Las **vistas complejas** pueden combinar datos de múltiples tablas, incluir funciones agregadas, subconsultas o `JOINs`. Aunque son útiles para consultas más avanzadas, las vistas complejas generalmente **no son actualizables**, ya que manipulan los datos de una manera más avanzada que impide que las modificaciones directas se propaguen a las tablas subyacentes.

Ejemplo:

```
CREATE VIEW VistaVentas AS
SELECT e.Nombre, e.Apellido, SUM(v.Cantidad) AS TotalVentas
FROM Empleados e
JOIN Ventas v ON e.IdEmpleado = v.IdEmpleado
GROUP BY e.Nombre, e.Apellido;
```

3. Vistas indexadas:

En **SQL Server**, las **vistas indexadas** son un tipo especial de vistas en las que los resultados de la consulta se almacenan físicamente en disco. Esto mejora el rendimiento de las consultas repetitivas sobre la vista, pero también tiene limitaciones, como el requisito de usar **`WITH SCHEMABINDING`**.

Ejemplo:

```
CREATE VIEW VistaSalarios
WITH SCHEMABINDING AS
SELECT IdEmpleado, Salario
FROM dbo.Empleados;
GO

CREATE UNIQUE CLUSTERED INDEX idx_salario
ON VistaSalarios (Salario);
```

4. Vistas con columnas calculadas:

Estas vistas incluyen columnas que no existen en las tablas subyacentes, pero que son **calculadas** a partir de otras columnas en tiempo de ejecución. Son útiles para simplificar consultas con cálculos frecuentes.

Ejemplo:

```
CREATE VIEW VistaSalarioAnual AS
SELECT Nombre, Apellido, Salario, (Salario * 12) AS SalarioAnual
FROM Empleados;
```

Limitaciones en la actualización de vistas

Aunque puedes **actualizar, insertar o eliminar** datos a través de una vista en algunos casos, hay limitaciones importantes a tener en cuenta. Para que una vista sea **actualizable**, debe cumplir con ciertas condiciones.

Requisitos para que una vista sea actualizable:

1. **Debe estar basada en una sola tabla**: Las vistas que combinan varias tablas o usan `JOINs` generalmente no son actualizables.
2. **No debe contener funciones agregadas** (`SUM()`, `COUNT()`, etc.).
3. **No debe usar `DISTINCT`**, `GROUP BY`, o `HAVING`.
4. **No debe contener subconsultas**.
5. **Todas las columnas necesarias** para realizar la actualización (por ejemplo, las claves primarias) deben estar incluidas en la vista.

Ejemplo de una vista actualizable:

```
CREATE VIEW VistaSimple AS
SELECT IdEmpleado, Nombre, Apellido, Salario
FROM Empleados;
```

Esta vista es actualizable porque no contiene ninguna de las limitaciones mencionadas (es simple y se basa en una sola tabla).

Insertar en la vista:

```
INSERT INTO VistaSimple (IdEmpleado, Nombre, Apellido, Salario)
VALUES (6, 'Carlos', 'López', 4500);
```

Limitaciones al actualizar vistas:

Si la vista contiene **elementos complejos**, las actualizaciones pueden fallar o no estar permitidas.

Ejemplo de una vista NO actualizable:

```
CREATE VIEW VistaCompleja AS
SELECT e.Nombre, SUM(v.Cantidad) AS TotalVentas
FROM Empleados e
JOIN Ventas v ON e.IdEmpleado = v.IdEmpleado
GROUP BY e.Nombre;
```

Esta vista **no es actualizable** porque:

- Utiliza la función de agregación `SUM()`.
- Tiene un `GROUP BY`.

Intentar **insertar, actualizar o eliminar** datos a través de esta vista no es posible.

Actualización parcial con vistas:

En algunos casos, aunque la vista sea actualizable, ciertas columnas de la vista pueden no ser modificables. Por ejemplo, las columnas calculadas no pueden ser actualizadas directamente.

Ejemplo con columna calculada (NO actualizable):

```
CREATE VIEW VistaSalarioCalculado AS
SELECT IdEmpleado, Nombre, Salario, (Salario * 12) AS SalarioAnual
FROM Empleados;
```

En este caso, no puedes actualizar directamente la columna **`SalarioAnual`** ya que es una columna calculada. Cualquier intento de actualizar esta columna dará error.

Uso de `WITH CHECK OPTION` en vistas actualizables:

SQL Server permite agregar la cláusula **`WITH CHECK OPTION`** a las vistas actualizables para asegurarse de que las modificaciones a través de la vista cumplan con las condiciones definidas en la vista.

Ejemplo:

```
CREATE VIEW VistaVentas AS
SELECT Nombre, Apellido, Salario
FROM Empleados
WHERE Departamento = 'Ventas'
WITH CHECK OPTION;
```

- **`WITH CHECK OPTION`**: Garantiza que cualquier fila insertada o actualizada a través de la vista siga cumpliendo con la condición **`WHERE Departamento = 'Ventas'`**. Si intentas insertar o actualizar un empleado fuera del departamento de ventas, el sistema dará un error.

Estrategias para el Diseño y Optimización de Vistas Complejas en SQL Server

Las **vistas** en **SQL Server** son una herramienta útil para simplificar consultas complejas, mejorar la seguridad y organizar el acceso a los datos. Sin embargo, al diseñar y optimizar vistas complejas, es importante seguir algunas estrategias clave para mejorar tanto el rendimiento como la facilidad de uso.

1. Simplificar las Consultas Subyacentes

Cuando se trabaja con una vista compleja, es recomendable analizar y simplificar las consultas subyacentes. Si una consulta es difícil de leer o entender, podría convertirse en un punto de fallo. Desglosar la lógica en pasos más simples o utilizar vistas adicionales para encapsular partes de la lógica puede ser una buena estrategia.

2. Indexar las Tablas Subyacentes

En lugar de depender exclusivamente de las vistas, indexar las columnas que se utilizan en las cláusulas **`WHERE`** o **`JOIN`** en las tablas subyacentes es una técnica fundamental para mejorar el rendimiento de las consultas.

Ejemplo:

Si tu vista consulta frecuentemente una columna **`id_categoria`** de la tabla **`productos`**, debes asegurarte de que dicha columna esté indexada:

```
CREATE INDEX idx_productos_categoria_id ON productos (id_categoria);
```

3. Uso de Vistas Indexadas en SQL Server

SQL Server no admite vistas materializadas como Oracle, pero puedes crear **vistas indexadas** para almacenar físicamente los resultados de una vista. Esto mejora el rendimiento de consultas que se ejecutan frecuentemente contra la vista.

Ejemplo:

```
CREATE VIEW VistaVentasConIndice
WITH SCHEMABINDING AS
SELECT c.nombre AS Categoria, SUM(p.venta) AS TotalVentas
FROM dbo.Productos p
JOIN dbo.Categorias c ON p.CategoriaId = c.Id
GROUP BY c.nombre;
GO

CREATE UNIQUE CLUSTERED INDEX idx_vista_ventas ON
VistaVentasConIndice(Categoria);
```

Este tipo de vistas almacenan los resultados de la consulta, lo que acelera las consultas repetidas.

4. Evaluar el Impacto en el Rendimiento

Antes de implementar una vista compleja en un entorno de producción, siempre es recomendable realizar pruebas de rendimiento. Asegúrate de que la vista no cause cuellos de botella ni afecte negativamente el rendimiento general del sistema.

Permisos y Seguridad en Vistas: Control de Acceso a Través de Vistas en SQL Server

Las vistas en **SQL Server** no solo sirven para simplificar consultas, sino también para limitar el acceso a los datos. Puedes usar vistas para controlar qué datos pueden ver y modificar los usuarios sin darles acceso directo a las tablas subyacentes. Esto proporciona un nivel adicional de seguridad.

Control de Acceso Mediante Permisos

Puedes otorgar permisos específicos sobre vistas para restringir qué operaciones pueden realizar ciertos usuarios o roles. Por ejemplo, puedes permitir que ciertos usuarios solo tengan acceso de lectura a través de una vista, mientras que otros pueden tener permisos de lectura y escritura.

Ejemplo de Otorgar Permisos:

```
-- Otorgar permisos de solo lectura a un usuario
GRANT SELECT ON VistaVentasConIndice TO UsuarioLectura;

-- Otorgar permisos de escritura a otro usuario
GRANT INSERT, UPDATE, DELETE ON VistaVentasConIndice TO
UsuarioEscritura;
```

Herencia de Permisos

Los permisos sobre las vistas se heredan de las tablas subyacentes. Es decir, si un usuario tiene permiso para acceder a una vista, pero no tiene permiso para acceder a una tabla subyacente, no podrá ver ni modificar los datos de la tabla a través de la vista.

Mejores Prácticas y Consejos para Trabajar con Vistas en SQL Server

Para sacar el máximo provecho de las vistas en **SQL Server**, sigue estas mejores prácticas:

1. Evitar Vistas Anidadas

El uso excesivo de vistas anidadas (vistas que dependen de otras vistas) puede degradar el rendimiento. Siempre que sea posible, intenta simplificar la estructura de tus vistas para evitar la dependencia de múltiples capas de vistas.

2. Limitar el Número de Columnas en la Vista

Incluye solo las columnas que realmente necesitas en la vista. Evita seleccionar todas las columnas de una tabla (**`SELECT *`**) para reducir la sobrecarga y mejorar el rendimiento.

3. Usar Índices en las Tablas Subyacentes

Para vistas que se consultan con frecuencia, asegúrate de que las tablas subyacentes estén bien indexadas, especialmente en las columnas que se utilizan en **`JOINs`** y **`WHERE`**.

Escenarios Empresariales y Aplicaciones de Vistas en SQL Server

Las vistas tienen una amplia gama de aplicaciones en los negocios, facilitando tanto la consulta como el análisis de datos. A continuación, te mostramos algunos casos donde las vistas pueden ser útiles.

1. Informes Detallados

En lugar de escribir una consulta larga y compleja cada vez que necesites generar un informe, puedes crear una vista que encapsule toda la lógica de la consulta. Esto permite generar informes más rápidamente.

Ejemplo:

```
CREATE VIEW VistaVentasMensuales AS
SELECT c.nombre AS Categoria, SUM(p.venta) AS TotalVentas,
MONTH(p.FechaVenta) AS Mes
FROM Productos p
JOIN Categorias c ON p.CategoriaId = c.Id
GROUP BY c.nombre, MONTH(p.FechaVenta);
```

Esta vista simplifica la generación de informes de ventas mensuales sin tener que reescribir la consulta cada vez.

2. Análisis de Datos Avanzado

Las vistas también son útiles para realizar análisis de datos avanzados. Puedes agregar filtros, realizar cálculos o agrupar datos para preparar la base de un análisis más profundo.

Ejemplo de Vista para Análisis de Tendencias:

```
CREATE VIEW VistaTendenciasVentas AS
SELECT Region, Categoria, SUM(Ventas) AS TotalVentas, YEAR(FechaVenta)
AS Anio
FROM Ventas
GROUP BY Region, Categoria, YEAR(FechaVenta);
```

Caso de Ejemplo para Demostración en SQL Server

Caso: Informe de Ventas por Región y Categoría de Producto

Supongamos que trabajas en una empresa minorista y necesitas un informe que te muestre las ventas totales agrupadas por región y categoría de producto.

1. Crear la Vista

```
CREATE VIEW VistaVentasRegionCategoria AS
SELECT r.nombre AS Region, c.nombre AS Categoria, SUM(p.venta) AS
TotalVentas
FROM Productos p
JOIN Categorias c ON p.CategoriaId = c.Id
JOIN Regiones r ON p.RegionId = r.Id
GROUP BY r.nombre, c.nombre;
```

2. Consultar la Vista

Ahora puedes utilizar esta vista para generar informes fácilmente sin reescribir la consulta.

```
SELECT * FROM VistaVentasRegionCategoria
WHERE TotalVentas > 50000;
```

3. Optimización

Si las consultas sobre la vista son frecuentes, puedes mejorar el rendimiento creando índices en las tablas subyacentes.

```
CREATE INDEX idx_productos_categoria ON Productos (CategoriaId);
CREATE INDEX idx_productos_region ON Productos (RegionId);
```