Unidad 5. SQL y SQL Avanzado

Tema 1. Introducción a SQL y Definición de Datos (DDL)

El **Lenguaje de Definición de Datos (DDL)** de SQL es como el arquitecto de la base de datos. Permite crear, modificar y eliminar la estructura de los objetos de la base de datos, como tablas, índices y vistas. Es fundamental porque define cómo se almacenarán los datos y qué reglas deben cumplir, asegurando la integridad desde la cimentación.

Sintaxis

CREATE TABLE

Se usa para crear una nueva tabla en la base de datos. Define el nombre de la tabla, sus columnas, los tipos de datos para cada columna y las restricciones.

```
-- Crear la tabla Departamentos
CREATE TABLE Departamentos (
    id_departamento INT PRIMARY KEY, -- id_departamento es un entero y es la clave primaria (identificador único)
    nombre_departamento VARCHAR(50) UNIQUE NOT NULL, -- Nombre del departamento, texto de hasta 50 caracteres, debe ser único y no puede ser nulo
    ubicacion VARCHAR(50) -- Ubicación del departamento, texto de hasta 50 caracteres
);

-- Crear la tabla Empleados
CREATE TABLE Empleados (
    id_empleado INT PRIMARY KEY, -- id_empleado es un entero y es la clave primaria
    nombre VARCHAR(50) NOT NULL, -- Nombre del empleado, no puede ser nulo
    apellido VARCHAR(50) NOT NULL, -- Apellido del empleado, no puede ser nulo
    id_departamento INT, -- id_departamento para relacionar con la tabla Departamentos
    salario DECIMAL(10, 2), -- Salario del empleado, número decimal con 10 dígitos en total y 2 después del punto
    fecha_contratacion DATE, -- Fecha de contratación del empleado
    email VARCHAR(100) UNIQUE, -- Email del empleado, debe ser único
    FOREIGN KEY (id_departamento) REFERENCES Departamentos(id_departamento) -- Define id_departamento como clave foránea que referencia a la clave primaria de Departamentos
);
```

ALTER TABLE

Permite modificar la estructura de una tabla existente, como añadir o eliminar columnas, o agregar/quitar restricciones.

```sql
-- Añadir una nueva columna 'telefono' a la tabla Empleados
ALTER TABLE Empleados
ADD COLUMN telefono VARCHAR(15);

-- Eliminar la columna 'ubicacion' de la tabla Departamentos
ALTER TABLE Departamentos
DROP COLUMN ubicacion;

-- Añadir una restricción CHECK para asegurar que el salario sea positivo
ALTER TABLE Empleados
ADD CONSTRAINT CHK_Salario CHECK (salario > 0);
```

DROP TABLE

Elimina una tabla completa de la base de datos, incluyendo todos sus datos, índices y restricciones. ¡Cuidado con su uso! ⚠️

```sql
-- Eliminar la tabla Proyectos
DROP TABLE Proyectos;
```

Casos de Uso Comunes y Mejores Prácticas

- **Casos de Uso**: Definir el esquema inicial de una base de datos, adaptar la estructura a nuevos requisitos de negocio, migrar datos entre sistemas.
- **Mejores Prácticas**:
    - **Nombrado Consistente**: Usa nombres claros y consistentes para tablas y columnas (ej., `snake_case`).
    - **Tipos de Datos Correctos**: Elige el tipo de dato más apropiado para cada columna para optimizar el almacenamiento y el rendimiento.
    - **Restricciones desde el Diseño**: Define las restricciones (`PRIMARY KEY`, `FOREIGN KEY`, `NOT NULL`, `UNIQUE`, `CHECK`) desde la creación de la tabla para asegurar la integridad de los datos.
- **Trampas Comunes**:
    - Olvidar las claves foráneas, rompiendo la integridad referencial.
    - Usar tipos de datos demasiado grandes o pequeños, desperdiciando espacio o causando truncamiento de datos.
    - Borrar tablas con `DROP TABLE` sin un respaldo, perdiendo todos los datos.

Relación con Otros Temas

El DDL es la base para toda la base de datos. Las **restricciones de integridad** (Unidad 3 y tema 11 de esta unidad) se implementan principalmente a través del DDL. Sin una estructura sólida creada con DDL, el **DML (Manipulación de Datos)** no tendría dónde operar ni reglas que seguir.

Tema 2. Estructura Básica de las Consultas SQL (DML - SELECT)

El **Lenguaje de Manipulación de Datos (DML)**, específicamente la sentencia `SELECT`, es tu herramienta para **consultar y recuperar información** de la base de datos. Imagina tu base de datos como una biblioteca: el DDL organiza los estantes y los libros, mientras que `SELECT` te permite encontrar los libros que necesitas, filtrarlos y ordenarlos.

Sintaxis

La estructura básica de una consulta `SELECT` es `SELECT <columnas> FROM <tablas> WHERE <condiciones>`.

```
SQL
-- Recuperar el nombre y apellido de todos los empleados
SELECT
    nombre,          -- Selecciona la columna 'nombre'
    apellido         -- Selecciona la columna 'apellido'
FROM
    Empleados;       -- De la tabla 'Empleados'

-- Recuperar todos los datos de los empleados del departamento con ID 1
SELECT
    * -- Selecciona todas las columnas
FROM
    Empleados        -- De la tabla 'Empleados'
WHERE
    id_departamento = 1; -- Donde el id_departamento sea igual a 1

-- Recuperar nombres de departamentos únicos en orden alfabético
SELECT DISTINCT
    nombre_departamento -- Selecciona nombres de departamentos, eliminando duplicados
FROM
    Departamentos
ORDER BY
    nombre_departamento ASC; -- Ordena los resultados por nombre_departamento de forma
ascendente
```

LIMIT/TOP

Controla el número de filas devueltas. `LIMIT` es común en MySQL, PostgreSQL; `TOP` en SQL Server.

```sql
SQL
-- Recuperar los 3 empleados con el salario más alto (ejemplo MySQL/PostgreSQL)
SELECT
    nombre,
    apellido,
    salario
FROM
    Empleados
ORDER BY
    salario DESC -- Ordena por salario de forma descendente
LIMIT 3;        -- Limita los resultados a 3 filas
```

Casos de Uso Comunes y Mejores Prácticas

- **Casos de Uso**: Generar informes, buscar información específica, preparar datos para aplicaciones o análisis.
- **Mejores Prácticas**:
    - **Especificar Columnas**: En lugar de `SELECT *`, lista las columnas específicas que necesitas. Esto mejora el rendimiento y la legibilidad.
    - **Usar Alias**: Para nombres de columnas o tablas largos, usa `AS` para crear alias cortos y legibles.
    - **Indentar Código**: Ayuda enormemente a la legibilidad de consultas complejas.
- **Trampas Comunes**:
    - Olvidar la cláusula `WHERE` y recuperar accidentalmente todos los datos de una tabla grande.
    - Confundir `DISTINCT` con `GROUP BY`. `DISTINCT` es para filas únicas; `GROUP BY` es para agrupar y agregar.

Relación con Otros Temas

Las consultas `SELECT` son la base para todas las operaciones de recuperación de datos. Se combinan con **JOINs** para recuperar datos de múltiples tablas, con **funciones de agregación** para resúmenes, y forman la base de las **Vistas** y **Subconsultas Anidadas**.

Tema 3. Operaciones sobre Conjuntos

Las operaciones sobre conjuntos (`UNION`, `INTERSECT`, `EXCEPT`) te permiten **combinar los resultados de dos o más consultas** `SELECT` como si fueran conjuntos matemáticos. Son útiles cuando necesitas obtener información de diferentes fuentes que comparten una estructura similar. Piensa en ellas como fusionar listas de elementos con reglas específicas para duplicados y elementos únicos.

Sintaxis

Para que estas operaciones funcionen, las consultas `SELECT` deben tener el **mismo número de columnas** y las columnas correspondientes deben tener **tipos de datos compatibles**.

`UNION`

Combina los resultados de dos o más consultas, eliminando las filas duplicadas. `UNION ALL` incluye todas las filas, incluso los duplicados.

```
SQL
-- Obtener los nombres y apellidos de empleados y el nombre de los departamentos (como una lista combinada)
SELECT nombre, apellido FROM Empleados -- Selecciona nombre y apellido de Empleados
UNION                                  -- Une los resultados, eliminando duplicados
SELECT nombre_departamento, ubicacion FROM Departamentos; -- Selecciona nombre_departamento y ubicacion de Departamentos (ejemplo para mostrar compatibilidad de columnas)
-- Nota: La segunda columna 'ubicacion' se alinea con 'apellido' solo por tipo de dato, no por significado lógico.

-- Un ejemplo más práctico sería:
SELECT nombre, apellido FROM Empleados
UNION ALL
SELECT nombre_empleado_temp, apellido_empleado_temp FROM Empleados_Temporales; -- Si tuviéramos una tabla de empleados temporales
```

`INTERSECT`

Devuelve las filas que son comunes a ambos conjuntos de resultados (es decir, que aparecen en ambas consultas).

```
SQL
-- Asumamos que tenemos una tabla 'Empleados_Activos_Ultimo_Mes'
-- Obtener los empleados que están en ambos conjuntos (Empleados y Empleados_Activos_Ultimo_Mes)
SELECT id_empleado, nombre, apellido FROM Empleados
INTERSECT -- Devuelve las filas comunes
SELECT id_empleado, nombre, apellido FROM Empleados_Activos_Ultimo_Mes;
```

`EXCEPT` (o `MINUS`)

Devuelve las filas que están en el primer conjunto de resultados pero no en el segundo.

```
SQL
-- Obtener los empleados que están en la tabla Empleados pero no en Empleados_Activos_Ultimo_Mes
SELECT id_empleado, nombre, apellido FROM Empleados
EXCEPT -- Devuelve filas del primer SELECT que no están en el segundo
SELECT id_empleado, nombre, apellido FROM Empleados_Activos_Ultimo_Mes;
```

Casos de Uso Comunes y Mejores Prácticas

- **Casos de Uso**: Combinar listas de clientes de diferentes fuentes, identificar elementos comunes entre dos conjuntos de datos, encontrar diferencias entre versiones de datos.
- **Mejores Prácticas**:
    - **Compatibilidad de Columnas**: Asegúrate de que las consultas tienen el mismo número de columnas y tipos de datos compatibles.
    - **`UNION ALL` por Defecto**: Si no necesitas eliminar duplicados, usa `UNION ALL` para un mejor rendimiento, ya que `UNION` implica una operación de ordenamiento y eliminación de duplicados.
- **Trampas Comunes**:
    - Errores de compatibilidad de tipos o número de columnas, lo que causa fallos en la consulta.
    - Uso ineficiente de `UNION` cuando `UNION ALL` sería suficiente, impactando el rendimiento.

Relación con Otros Temas

Las operaciones de conjunto complementan las **consultas** `SELECT` y pueden ser parte de **subconsultas** o la base para la creación de **vistas** que combinan datos de diversas fuentes.

Tema 4. Funciones de Agregación y Agrupamiento

Las **funciones de agregación** te permiten realizar cálculos sobre un conjunto de filas y devolver un único valor. Son ideales para obtener resúmenes de tus datos. Imagina que en una planilla de cálculo usas `SUMA()`, `PROMEDIO()` o `CONTAR()`. En SQL, estas funciones se aplican a grupos de datos definidos por la cláusula `GROUP BY`.

Sintaxis

Las funciones de agregación más comunes son `COUNT`, `SUM`, `AVG`, `MIN`, y `MAX`.

```
SQL
-- Contar el número total de empleados
SELECT COUNT(*) AS TotalEmpleados -- Cuenta todas las filas en la tabla Empleados y le
da un alias
FROM Empleados;

-- Calcular el salario promedio de todos los empleados
SELECT AVG(salario) AS SalarioPromedio -- Calcula el promedio de la columna salario
FROM Empleados;

-- Obtener el salario mínimo y máximo
SELECT
    MIN(salario) AS SalarioMinimo, -- Obtiene el valor mínimo de la columna salario
    MAX(salario) AS SalarioMaximo  -- Obtiene el valor máximo de la columna salario
```

```
FROM
    Empleados;
```

GROUP BY y HAVING

GROUP BY agrupa filas que tienen los mismos valores en las columnas especificadas, permitiendo que las funciones de agregación operen sobre cada grupo. HAVING filtra estos grupos basándose en una condición de agregación.

```
SQL
-- Calcular el salario promedio por departamento
SELECT
    d.nombre_departamento, -- Selecciona el nombre del departamento
    AVG(e.salario) AS SalarioPromedioDepartamento -- Calcula el salario promedio para
cada grupo de departamento
FROM
    Empleados e
JOIN
    Departamentos d ON e.id_departamento = d.id_departamento -- Une Empleados y
Departamentos
GROUP BY
    d.nombre_departamento; -- Agrupa los resultados por nombre_departamento

-- Encontrar departamentos con un salario promedio superior a 60000
SELECT
    d.nombre_departamento,
    AVG(e.salario) AS SalarioPromedioDepartamento
FROM
    Empleados e
JOIN
    Departamentos d ON e.id_departamento = d.id_departamento
GROUP BY
    d.nombre_departamento
HAVING
    AVG(e.salario) > 60000; -- Filtra los grupos donde el salario promedio sea mayor a
60000
```

Casos de Uso Comunes y Mejores Prácticas

- **Casos de Uso**: Generar informes de ventas totales por producto, promedio de calificaciones por curso, conteo de usuarios activos por región, etc.
- **Mejores Prácticas**:
    - **Distinción WHERE vs HAVING**: Recuerda que WHERE filtra *filas individuales* antes del agrupamiento, mientras que HAVING filtra *grupos* después de que se han calculado las agregaciones.
    - **COUNT(*) vs COUNT(columna)**: COUNT(*) cuenta todas las filas, COUNT(columna) cuenta las filas donde la columna no es NULL.
- **Trampas Comunes**:
    - Intentar usar funciones de agregación directamente en la cláusula WHERE.

- No incluir en la cláusula `GROUP BY` todas las columnas que no están siendo agregadas en la cláusula `SELECT`.

Relación con Otros Temas

Las funciones de agregación son fundamentales para el **análisis de datos**. Se usan frecuentemente con **JOINs** para agregar datos de múltiples tablas y son clave en muchas **subconsultas** y **vistas**.

Tema 5. Valores Nulos

Un **valor** `NULL` en SQL representa la ausencia de un valor. No es lo mismo que cero, un espacio en blanco o una cadena vacía; simplemente significa "desconocido", "no aplicable" o "sin valor". Es como una casilla en un formulario que se dejó en blanco porque no hay información disponible o no es relevante.

Sintaxis

Para comprobar si un valor es `NULL`, se usan los operadores `IS NULL` o `IS NOT NULL`.

```
SQL
-- Encontrar empleados cuyo email no ha sido registrado
SELECT
    nombre,
    apellido,
    email
FROM
    Empleados
WHERE
    email IS NULL; -- Busca filas donde la columna 'email' es NULL

-- Encontrar empleados con un email registrado
SELECT
    nombre,
    apellido,
    email
FROM
    Empleados
WHERE
    email IS NOT NULL; -- Busca filas donde la columna 'email' NO es NULL
```

Impacto de `NULL`

- **Comparaciones**: Cualquier comparación con `NULL` (ej., `salario = NULL`) siempre resulta en `UNKNOWN` (desconocido), no en `TRUE` ni `FALSE`. Por eso se usa `IS NULL`.
- **Funciones de Agregación**: La mayoría de las funciones de agregación (excepto `COUNT(*)`) ignoran los valores `NULL`.

```SQL
-- Calcular el promedio de salario, ignorando los empleados con salario NULL
-- Si hubiera un empleado con salario NULL, no afectaría el AVG
SELECT AVG(salario) FROM Empleados;

-- Contar todos los empleados, incluyendo aquellos con columnas NULL (COUNT(*))
SELECT COUNT(*) FROM Empleados;

-- Contar solo los empleados con un email registrado (COUNT(columna))
SELECT COUNT(email) FROM Empleados;
```

Casos de Uso Comunes y Mejores Prácticas

- **Casos de Uso**: Representar datos faltantes, información no aplicable (ej., fecha de fin de un proyecto en curso).
- **Mejores Prácticas**:
  - **Ser Consciente**: Siempre considera la posibilidad de `NULL` al escribir consultas, especialmente en cláusulas `WHERE`, `JOIN` y funciones de agregación.
  - **`COALESCE`**: Utiliza la función `COALESCE(exp1, exp2, ...)` para reemplazar `NULL` con un valor por defecto si es necesario (ej., `COALESCE(email, 'No disponible')`).
- **Trampas Comunes**:
  - Usar `=` o `!=` para comparar con `NULL` (ej., `WHERE email = NULL`), lo cual nunca funcionará como se espera.
  - Asumir que `NULL` significa cero en cálculos numéricos.

Relación con Otros Temas

Los valores `NULL` son críticos para la **integridad de los datos** y se relacionan directamente con las **restricciones `NOT NULL`** en el DDL. Su manejo es fundamental en **consultas complejas** y en la lógica de las **funciones y procedimientos almacenados**.

Tema 6. Subconsultas Anidadas y Consultas Complejas

Una **subconsulta (o consulta anidada)** es una consulta `SELECT` dentro de otra sentencia SQL (como otro `SELECT`, `INSERT`, `UPDATE`, `DELETE`). Son como mini-consultas que proporcionan un resultado que la consulta externa puede utilizar. Imagina que para cocinar un plato complejo, primero necesitas preparar un ingrediente específico que luego usarás en la receta principal.

Sintaxis

Las subconsultas pueden aparecer en varias cláusulas:

Subconsultas en `WHERE`

Se utilizan para filtrar el conjunto de resultados de la consulta externa.

```
SQL
-- Encontrar empleados que trabajan en el departamento 'Ventas'
SELECT
    nombre,
    apellido
FROM
    Empleados
WHERE
    id_departamento IN (SELECT id_departamento FROM Departamentos WHERE
nombre_departamento = 'Ventas');
    -- La subconsulta devuelve el id_departamento de 'Ventas'
    -- La consulta externa selecciona empleados cuyo id_departamento esté en ese
resultado
```

Subconsultas con `EXISTS`/`NOT EXISTS`

Se utilizan para verificar la existencia de filas en el resultado de la subconsulta. Son útiles para correlacionar.

```
SQL
-- Encontrar departamentos que tienen al menos un empleado
SELECT
    nombre_departamento
FROM
    Departamentos d
WHERE
    EXISTS (SELECT 1 FROM Empleados e WHERE e.id_departamento = d.id_departamento);
    -- Para cada departamento, verifica si existe ALGÚN empleado asignado a ese
departamento
```

Subconsultas en `FROM` (Tablas Derivadas)

El resultado de una subconsulta se utiliza como una tabla temporal en la cláusula `FROM`.

```
SQL
-- Calcular el salario promedio de los empleados de cada departamento, pero solo para
departamentos con más de 2 empleados
SELECT
    dep.nombre_departamento,
    dept_stats.SalarioPromedio
FROM
    (SELECT id_departamento, AVG(salario) AS SalarioPromedio, COUNT(*) AS TotalEmpleados
     FROM Empleados
     GROUP BY id_departamento
     HAVING COUNT(*) > 2) AS dept_stats -- Subconsulta que actúa como una tabla temporal
'dept_stats'
JOIN
    Departamentos dep ON dept_stats.id_departamento = dep.id_departamento;
```

Subconsultas Correlacionadas

Dependen de la consulta externa para su ejecución. Se evalúan una vez por cada fila procesada por la consulta externa.

```
SQL
-- Encontrar empleados cuyo salario es mayor que el salario promedio de su propio
departamento
SELECT
    e.nombre,
    e.apellido,
    e.salario
FROM
    Empleados e
WHERE
    e.salario > (SELECT AVG(salario) FROM Empleados WHERE id_departamento =
e.id_departamento);
    -- La subconsulta se re-ejecuta para cada 'e.id_departamento' de la consulta externa
```

Casos de Uso Comunes y Mejores Prácticas

- **Casos de Uso**: Filtrar datos basados en valores dinámicos, realizar cálculos intermedios, comparar valores con promedios grupales.
- **Mejores Prácticas**:
    - **Legibilidad**: Indentar las subconsultas para mejorar la legibilidad del código.
    - **Rendimiento**: Para subconsultas grandes o correlacionadas, a veces un `JOIN` puede ser más eficiente. Analiza el plan de ejecución.
    - **Evitar `SELECT *`**: En subconsultas, selecciona solo las columnas estrictamente necesarias.
- **Trampas Comunes**:
    - Subconsultas que devuelven múltiples columnas cuando se espera una sola.
    - Errores de rendimiento por subconsultas correlacionadas en tablas muy grandes.

Relación con Otros Temas

Las subconsultas extienden el poder de las **consultas `SELECT`** y las **funciones de agregación**. Son una alternativa a los **JOINs** en ciertos escenarios y pueden ser la base para la definición de **vistas** complejas.

Tema 7. Vistas

Una **vista** es una tabla virtual basada en el resultado de una consulta `SELECT`. No almacena datos por sí misma, sino que es una definición lógica de cómo ver los datos de una o más tablas base. Imagina una vista como un filtro personalizado o una ventana a través de la cual miras tus datos.

Sintaxis

```
SQL
-- Crear una vista que muestre información básica de empleados y su departamento
CREATE VIEW Empleados_Vista_Basica AS
SELECT
    e.id_empleado,
    e.nombre,
    e.apellido,
    d.nombre_departamento AS Departamento,
    e.salario
FROM
    Empleados e
JOIN
    Departamentos d ON e.id_departamento = d.id_departamento
WHERE
    e.salario > 50000; -- La vista solo mostrará empleados con salario mayor a 50000

-- Consultar la vista como si fuera una tabla
SELECT * FROM Empleados_Vista_Basica WHERE Departamento = 'Ventas';

-- Eliminar una vista
DROP VIEW Empleados_Vista_Basica;
```

Casos de Uso Comunes y Mejores Prácticas

- **Casos de Uso**:
  - **Simplificación de Consultas**: Ocultar la complejidad de `JOINs` o subconsultas.
  - **Seguridad**: Restringir el acceso de los usuarios a ciertas filas o columnas, sin dar acceso directo a las tablas base.
  - **Consistencia**: Presentar los datos de una manera consistente para diferentes aplicaciones.
- **Mejores Prácticas**:
  - **Nombres Descriptivos**: Dale a tus vistas nombres que reflejen su propósito.
  - **Evitar Vistas de Vistas**: Anidar demasiadas vistas puede dificultar el mantenimiento y el rendimiento.
- **Limitaciones**:
  - **Actualización**: No todas las vistas son actualizables (modificables con `INSERT`, `UPDATE`, `DELETE`). Generalmente, solo las vistas basadas en una única tabla, sin funciones de agregación, `GROUP BY`, `DISTINCT` o `JOINs` complejos, son actualizables.
  - **Rendimiento**: Aunque son virtuales, las vistas se expanden a su consulta subyacente cada vez que se consultan, lo que puede afectar el rendimiento si son muy complejas.

Relación con Otros Temas

Las vistas se construyen a partir de **consultas** `SELECT` (a menudo con **JOINs** y **subconsultas**). También son un componente clave en la **seguridad y autorización** de la base de datos, ya que controlan qué datos pueden ver los usuarios.

Tema 8. Modificación de la Base de Datos (DML - INSERT, DELETE, UPDATE)

El **Lenguaje de Manipulación de Datos (DML)** no solo es para consultar (`SELECT`), sino también para **modificar los datos** almacenados en las tablas. Estas operaciones son cruciales para mantener la base de datos al día con la información cambiante del mundo real.

Sintaxis

`INSERT`

Añade nuevas filas a una tabla.

```
SQL
-- Insertar una nueva fila con valores específicos
INSERT INTO Departamentos (id_departamento, nombre_departamento, ubicacion)
VALUES (3, 'Recursos Humanos', 'Edificio C');
-- Especifica las columnas y luego los valores en el mismo orden

-- Insertar una nueva fila, dejando algunas columnas con su valor por defecto o NULL
INSERT INTO Empleados (id_empleado, nombre, apellido, id_departamento)
VALUES (101, 'Ana', 'Gomez', 1); -- Salario y fecha_contratacion serán NULL por defecto
```

`UPDATE`

Modifica los valores de las columnas en filas existentes.

```
SQL
-- Aumentar el salario del empleado con ID 101 en un 10%
UPDATE Empleados
SET
    salario = salario * 1.10 -- El nuevo salario es el actual + 10%
WHERE
    id_empleado = 101; -- Aplica la actualización solo al empleado con ID 101

-- Cambiar la ubicación del departamento de 'Ventas'
UPDATE Departamentos
SET
    ubicacion = 'Edificio B'
WHERE
    nombre_departamento = 'Ventas';
```

`DELETE`

Elimina filas de una tabla.

```
SQL
-- Eliminar al empleado con ID 101
DELETE FROM Empleados
WHERE
    id_empleado = 101; -- Elimina solo la fila del empleado con ID 101

-- Eliminar todos los empleados del departamento de 'Ventas'
DELETE FROM Empleados
WHERE
    id_departamento = (SELECT id_departamento FROM Departamentos WHERE
nombre_departamento = 'Ventas');
    -- Utiliza una subconsulta para identificar el id_departamento de 'Ventas'
```

Casos de Uso Comunes y Mejores Prácticas

- **Casos de Uso**: Registrar nuevas transacciones, actualizar el estado de pedidos, eliminar registros obsoletos.
- **Mejores Prácticas**:
    - **Siempre `WHERE`**: ¡Cuidado! Siempre usa la cláusula `WHERE` en `UPDATE` y `DELETE` a menos que quieras modificar/eliminar *todas* las filas de la tabla.
    - **Prueba con `SELECT`**: Antes de ejecutar un `UPDATE` o `DELETE` complejo, ejecuta un `SELECT` con la misma cláusula `WHERE` para verificar qué filas serán afectadas.
    - **Transacciones**: Para operaciones críticas, usa transacciones (`BEGIN TRANSACTION`, `COMMIT`, `ROLLBACK`) para poder deshacer cambios si algo sale mal.
- **Trampas Comunes**:
    - Ejecutar `DELETE FROM Tabla;` sin `WHERE` y borrar todos los datos.
    - Actualizar un valor incorrectamente debido a una cláusula `WHERE` mal construida.

Relación con Otros Temas

Las operaciones DML están intrínsecamente ligadas a las **restricciones de integridad** (definidas con DDL). Un `INSERT` o `UPDATE` fallará si viola una restricción de clave primaria, unicidad o clave foránea. Las **subconsultas** se usan con frecuencia en `INSERT` (para insertar resultados de una consulta), `UPDATE` y `DELETE` para identificar las filas afectadas.

Tema 9. Reunión de Relaciones (Combinaciones - JOINs)

Los **JOINs** son la columna vertebral de las consultas en bases de datos relacionales. Permiten **combinar filas de dos o más tablas** basándose en una columna relacionada entre ellas. Piensa en esto como juntar información dispersa en diferentes archivos (tablas) que comparten un identificador común, para obtener una vista completa y unificada.

Sintaxis

`INNER JOIN`

Devuelve sólo las filas donde hay coincidencias en ambas tablas, basándose en la condición especificada.

```
SQL
-- Obtener el nombre del empleado y el nombre del departamento al que pertenece
SELECT e.nombre, e.apellido, d.nombre_departamento -- Selecciona columnas de Empleados
(e) y Departamentos (d)
FROM Empleados e             -- Alias 'e' para la tabla Empleados
INNER JOIN Departamentos d ON e.id_departamento = d.id_departamento; -- Une donde
id_departamento coincida en ambas tablas
```

`LEFT JOIN` (o `LEFT OUTER JOIN`)

Devuelve todas las filas de la tabla izquierda (la primera en `FROM`) y las filas coincidentes de la tabla derecha. Si no hay coincidencia en la tabla derecha, los valores para sus columnas serán `NULL`.

```
SQL
-- Obtener todos los departamentos y, si tienen empleados, el nombre de
los empleados asociados.
-- Los departamentos sin empleados también aparecerán.
SELECT
    d.nombre_departamento,
    e.nombre,
    e.apellido
FROM
    Departamentos d             -- Departamentos es la tabla 'izquierda'
LEFT JOIN
    Empleados e ON d.id_departamento = e.id_departamento; -- Une con
Empleados
```

`RIGHT JOIN` (o `RIGHT OUTER JOIN`)

Funciona de forma opuesta a `LEFT JOIN`: devuelve todas las filas de la tabla derecha y las filas coincidentes de la tabla izquierda. Si no hay coincidencia en la tabla izquierda, los valores serán `NULL`.

```
SQL
-- Obtener todos los empleados y, si tienen departamento, el nombre del departamento.
-- (Similar a INNER JOIN si todos los empleados tienen departamento, pero ilustra el
punto)
SELECT
    e.nombre,
    e.apellido,
    d.nombre_departamento
FROM
    Empleados e             -- Empleados es la tabla 'izquierda'
RIGHT JOIN
    Departamentos d ON e.id_departamento = d.id_departamento; -- Departamentos es la
tabla 'derecha'
```

`FULL OUTER JOIN` (Soporte varía entre SGBD, ej. PostgreSQL, SQL Server)

Devuelve todas las filas cuando hay una coincidencia en una de las tablas. Si no hay coincidencia, devuelve `NULL` para la tabla sin coincidencia. Combina los resultados de `LEFT` y `RIGHT JOIN`.

```
SQL
-- (Ejemplo conceptual, sintaxis puede variar)
-- Obtener todos los departamentos y todos los empleados, mostrando las
correspondencias.
-- Se mostrarán departamentos sin empleados y empleados sin departamento (si
existieran).
SELECT
    d.nombre_departamento,    e.nombre,
    e.apellido
FROM
    Departamentos d
FULL OUTER JOIN
    Empleados e ON d.id_departamento = e.id_departamento;
```

`NATURAL JOIN` (Precaución)

Une tablas automáticamente basándose en columnas que tienen el mismo nombre en ambas tablas. Es conveniente, pero menos explícito y puede llevar a uniones inesperadas si los nombres de columna coinciden por casualidad.

```
SQL
-- Une Empleados y Departamentos automáticamente por 'id_departamento'
SELECT
    nombre,
    apellido,
    nombre_departamento
FROM
    Empleados
NATURAL JOIN
    Departamentos;
```

ON vs. USING

- **ON**: Especifica explícitamente la condición de unión (la más común y recomendada).
- **USING**: Se usa cuando las columnas de unión tienen el mismo nombre en ambas tablas y quieres unirlas por esa columna. Más concisa que ON pero menos flexible.

```
SQL
-- Ejemplo de JOIN con USING (equivalente al INNER JOIN con ON e.id_departamento =
d.id_departamento)
SELECT
    e.nombre,
    e.apellido,
    d.nombre_departamento
FROM
    Empleados e
JOIN
    Departamentos d USING (id_departamento); -- Se une por la columna 'id_departamento'
```

Casos de Uso Comunes y Mejores Prácticas

- **Casos de Uso**: Recuperar información que está distribuida en varias tablas relacionadas.
- **Mejores Prácticas**:
    - **Siempre ON**: Prefiere la cláusula ON para definir tus condiciones de unión; es explícita y menos propensa a errores que NATURAL JOIN.
    - **Alias de Tablas**: Usa alias cortos para las tablas (ej., e para Empleados) para hacer las consultas más concisas y legibles.
    - **Entender el Tipo de JOIN**: Elige el tipo de JOIN adecuado según si necesitas incluir filas sin coincidencia.
- **Trampas Comunes**:
    - Olvidar la condición ON en un JOIN y terminar con un **producto cartesiano** (todas las combinaciones de filas), lo cual puede ser catastrófico en tablas grandes.
    - Confundir LEFT y RIGHT JOIN.

Relación con Otros Temas

Los JOINs son esenciales para la mayoría de las **consultas SELECT** complejas. A menudo se combinan con **funciones de agregación** y **GROUP BY** para análisis inter-tabla, y son fundamentales para construir **vistas** significativas.

Tema 10. Tipos de Datos y Esquemas

Tipos de Datos

Los **tipos de datos** definen el tipo de valores que puede almacenar una columna (ej., números, texto, fechas). Elegir el tipo de dato correcto es crucial para la eficiencia del almacenamiento, la validez de los datos y el rendimiento de las operaciones. Es como decidir qué tipo de recipiente usarás para guardar algo: un vaso para líquidos, una caja para objetos sólidos.

Esquemas (Schemas)

Un **esquema** es una colección lógica de objetos dentro de una base de datos (tablas, vistas, índices, funciones, etc.), agrupados bajo un nombre. Actúa como un contenedor o un espacio de nombres para organizar los objetos de la base de datos y gestionar permisos. Piensa en un esquema como una carpeta dentro de tu base de datos para organizar tus archivos.

Sintaxis

Tipos de Datos Comunes

```
SQL
-- Ejemplos de tipos de datos en la creación de tablas
CREATE TABLE Productos (
    id_producto INT PRIMARY KEY,           -- Número entero
    nombre_producto VARCHAR(100) NOT NULL, -- Cadena de caracteres de
longitud variable, máximo 100
    precio DECIMAL(8, 2),                  -- Número decimal con 8
dígitos en total, 2 después del punto (ej. 123456.78)
    fecha_lanzamiento DATE,                -- Fecha (año, mes, día)
    disponible BOOLEAN,                    -- Valor booleano (TRUE/FALSE
o 1/0)
    descripcion TEXT                       -- Texto largo sin límite de
longitud definido
);
```

Esquemas

La creación de esquemas y su uso varía ligeramente entre SGBD (ej. SQL Server vs. PostgreSQL).

```
SQL
-- Crear un nuevo esquema para módulos de RRHH
CREATE SCHEMA RRHH_Schema;

-- Crear una tabla dentro del nuevo esquema
CREATE TABLE RRHH_Schema.Beneficios (
    id_beneficio INT PRIMARY KEY,
    nombre_beneficio VARCHAR(50)
);
```

```
-- Consultar una tabla dentro de un esquema específico
SELECT * FROM RRHH_Schema.Beneficios;
```

Casos de Uso Comunes y Mejores Prácticas

- **Casos de Uso (Tipos de Datos)**: Almacenar números de teléfono, fechas de nacimiento, descripciones de productos, precios.
- **Mejores Prácticas (Tipos de Datos)**:
    - **Ser Específico**: Elige el tipo de dato más restrictivo que se ajuste a tus necesidades (ej., `SMALLINT` en lugar de `INT` si sabes que los números serán pequeños).
    - **`VARCHAR` vs `CHAR`**: Usa `VARCHAR` para cadenas de longitud variable, `CHAR` para cadenas de longitud fija.
    - **Fechas y Horas**: Usa los tipos de datos de fecha/hora nativos (`DATE`, `TIME`, `DATETIME`, `TIMESTAMP`) para evitar problemas de formato.
- **Casos de Uso (Esquemas)**:
    - **Organización**: Agrupar objetos relacionados lógicamente.
    - **Seguridad**: Simplificar la gestión de permisos, otorgando o revocando permisos a nivel de esquema.
    - **Multi-tenant**: Permitir que múltiples aplicaciones o clientes compartan una misma base de datos, pero con sus propios esquemas aislados.
- **Mejores Prácticas (Esquemas)**:
    - **Planificación**: Diseña tu estructura de esquemas desde el principio si tu base de datos crecerá.
    - **Separación de Responsabilidades**: Usa esquemas para separar el código de la aplicación de las tablas de datos, o diferentes módulos de negocio.
- **Trampas Comunes**:
    - Elegir tipos de datos demasiado generales, lo que puede llevar a problemas de almacenamiento o rendimiento.
    - No usar esquemas y tener una base de datos con miles de objetos en el esquema por defecto, dificultando la gestión.

Relación con Otros Temas

Los tipos de datos son la base de la **definición de tablas (DDL)**. Las **restricciones de integridad** como `CHECK` a menudo dependen de los tipos de datos. Los esquemas son fundamentales para la **autorización** y la organización de los objetos de la base de datos.

Tema 11. Restricciones de Integridad (Profundización)

Las **restricciones de integridad** son reglas que se imponen a los datos en una base de datos para garantizar su exactitud y consistencia. Son como las leyes de tu base de datos: aseguran que solo los datos válidos y coherentes puedan ser almacenados.

Sintaxis

Las restricciones se definen típicamente al crear tablas o modificarlas.

PRIMARY KEY

Garantiza que la columna o conjunto de columnas contenga valores únicos y no nulos, identificando de forma única cada fila.

```
SQL
CREATE TABLE Productos (
    id_producto INT PRIMARY KEY, -- id_producto es la clave primaria
    nombre VARCHAR(100)
);
```

FOREIGN KEY

Crea un vínculo entre datos en dos tablas, asegurando la integridad referencial. Un valor en la columna de clave foránea debe existir en la clave primaria de la tabla referenciada o ser `NULL`.

```
SQL
CREATE TABLE Pedidos (
    id_pedido INT PRIMARY KEY,
    id_cliente INT,
    fecha_pedido DATE,
    FOREIGN KEY (id_cliente) REFERENCES Clientes(id_cliente) -- id_cliente referencia a
la tabla Clientes
);
```

UNIQUE

Asegura que todos los valores en una columna (o conjunto de columnas) sean distintos. A diferencia de `PRIMARY KEY`, puede aceptar un valor `NULL` (solo uno).

```
SQL
CREATE TABLE Usuarios (
    id_usuario INT PRIMARY KEY,
    nombre_usuario VARCHAR(50) UNIQUE, -- nombre_usuario debe ser único
    email VARCHAR(100) UNIQUE          -- email debe ser único
);
```

NOT NULL

Impide que una columna contenga valores NULL.

```
SQL
CREATE TABLE Tareas (
    id_tarea INT PRIMARY KEY,
    descripcion VARCHAR(255) NOT NULL, -- descripcion no puede ser nula
    fecha_vencimiento DATE
);
```

CHECK

Define una condición que debe ser verdadera para todos los valores en una columna.

```
SQL
CREATE TABLE Empleados (
    ...
    salario DECIMAL(10, 2) CHECK (salario >= 0), -- El salario debe ser mayor o igual a
cero
    edad INT CHECK (edad >= 18 AND edad <= 65)  -- La edad debe estar entre 18 y 65
);
```

Acciones Referenciales (ON DELETE, ON UPDATE)

Definen cómo reaccionar cuando se intenta eliminar o actualizar una fila a la que otras filas hacen referencia.

- **CASCADE**: Elimina o actualiza las filas dependientes.
- **SET NULL**: Establece la clave foránea en NULL en las filas dependientes.
- **RESTRICT**: Impide la eliminación o actualización si existen filas dependientes.
- **NO ACTION**: Similar a RESTRICT, pero la comprobación se realiza al final de la transacción.

```
SQL
CREATE TABLE Empleados (
    id_empleado INT PRIMARY KEY,
    id_departamento INT,
    FOREIGN KEY (id_departamento) REFERENCES Departamentos(id_departamento)
        ON DELETE CASCADE -- Si un departamento se elimina, todos sus empleados
asociados se eliminan. ¡CUIDADO!
        ON UPDATE CASCADE  -- Si el id_departamento cambia, se actualiza automáticamente
en Empleados
);
```

ASSERTION (Concepto Avanzado, soporte variable)

Define una restricción de integridad a nivel de base de datos que puede involucrar múltiples tablas.

```
SQL
-- (Ejemplo conceptual, sintaxis específica depende del SGBD)
-- ASEGURAR QUE EL SALARIO TOTAL DE TODOS LOS EMPLEADOS NO EXCEDA UN LÍMITE
-- CREATE ASSERTION MaxSalarioTotal CHECK
-- (SELECT SUM(salario) FROM Empleados) <= 1000000;
```

Casos de Uso Comunes y Mejores Prácticas

- **Casos de Uso**: Asegurar que cada cliente tiene un ID único, que cada pedido está vinculado a un cliente existente, que las edades son válidas.
- **Mejores Prácticas**:
    - **Define desde el Inicio**: Incorpora las restricciones de integridad en la fase de diseño (DDL) para evitar datos inconsistentes desde el principio.
    - **RESTRICT por Defecto**: Si no estás seguro de la acción referencial, RESTRICT o NO ACTION son las opciones más seguras, ya que te obligan a manejar manualmente las dependencias.
- **Trampas Comunes**:
    - Olvidar NOT NULL en columnas críticas, permitiendo datos incompletos.
    - Usar CASCADE a la ligera, lo que puede llevar a la eliminación masiva e involuntaria de datos.
    - Confundir UNIQUE con PRIMARY KEY.

Relación con Otros Temas

Las restricciones de integridad son la forma de implementar las reglas de negocio en la **definición de datos (DDL)**. Afectan directamente las operaciones de **modificación de datos (DML: INSERT, UPDATE, DELETE)**, ya que cualquier violación detendrá la operación. También son clave para la **normalización** (Unidad 4), ya que las claves primarias y foráneas son fundamentales en la teoría de formas normales.

Tema 12. Autorización (Control)

La **autorización** en SQL se refiere al proceso de controlar qué usuarios o roles tienen permiso para realizar qué operaciones sobre qué objetos dentro de la base de datos. Es como la seguridad en un edificio: no todos tienen acceso a todas las habitaciones o pueden realizar cualquier acción.

Sintaxis

El SQL estándar utiliza las sentencias `GRANT` y `REVOKE` para gestionar los privilegios.

GRANT

Otorga permisos a usuarios o roles.

```
SQL
-- Otorgar permiso para seleccionar datos de la tabla Empleados al usuario 'analista'
GRANT SELECT ON Empleados TO analista;

-- Otorgar permiso para insertar y actualizar datos en la tabla Departamentos al usuario
'admin_rh'
GRANT INSERT, UPDATE ON Departamentos TO admin_rh;

-- Otorgar todos los permisos sobre la tabla Proyectos al usuario 'gestor_proyectos'
GRANT ALL PRIVILEGES ON Proyectos TO gestor_proyectos;

-- Otorgar permiso para crear tablas en el esquema 'RRHH_Schema' a 'desarrollador_rh'
GRANT CREATE TABLE ON SCHEMA :: RRHH_Schema TO desarrollador_rh; -- Sintaxis puede
variar entre SGBD
```

REVOKE

Retira permisos previamente otorgados.

```
SQL-- Revocar el permiso de seleccionar datos de la tabla Empleados al usuario
'analista'
REVOKE SELECT ON Empleados FROM analista;

-- Revocar todos los permisos sobre la tabla Proyectos a 'gestor_proyectos'
REVOKE ALL PRIVILEGES ON Proyectos FROM gestor_proyectos;
```

Concepto de Roles

Un **rol** es una colección de privilegios que se pueden asignar a uno o varios usuarios. Esto simplifica la gestión de permisos, ya que se otorgan permisos al rol, y luego se asigna el rol a los usuarios.

```
SQL
-- Crear un rol para usuarios de solo lectura
CREATE ROLE read_only_role;

-- Otorgar permiso de SELECT a todas las tablas del esquema 'public' al rol
GRANT SELECT ON ALL TABLES IN SCHEMA public TO read_only_role;

-- Asignar el rol 'read_only_role' al usuario 'reportero'
GRANT read_only_role TO reportero;
```

Casos de Uso Comunes y Mejores Prácticas

- **Casos de Uso**: Controlar quién puede ver datos sensibles (salarios), quién puede modificar la estructura de la base de datos, quién puede añadir/borrar registros.
- **Mejores Prácticas**:
    - **Principio de Mínimo Privilegio**: Otorga solo los permisos estrictamente necesarios para que un usuario realice su tarea.
    - **Usar Roles**: Siempre que sea posible, gestiona los permisos a través de roles en lugar de directamente a usuarios individuales. Esto facilita la administración y el mantenimiento.
    - **Documentar Permisos**: Mantén un registro claro de quién tiene qué permisos.
- **Trampas Comunes**:
    - Otorgar demasiados permisos por comodidad, creando agujeros de seguridad.
    - Olvidar revocar permisos cuando un usuario cambia de rol o deja la organización.

Relación con Otros Temas

La autorización es esencial para la **seguridad de la base de datos**. Se basa en los **objetos creados con DDL** (tablas, vistas) y protege el acceso a los datos manipulados con **DML**. Las **vistas** también juegan un papel importante en la seguridad al permitir exponer solo un subconjunto de datos.

Tema 13. SQL Incorporado y SQL Dinámico

Cuando desarrollamos aplicaciones, a menudo necesitamos que interactúen con una base de datos. **SQL Incorporado** y **SQL Dinámico** son dos enfoques para lograr esta interacción, permitiendo que el código de la aplicación ejecute sentencias SQL.

- **SQL Incorporado (Embedded SQL)**
  Se refiere a la inclusión de sentencias SQL estáticas directamente dentro del código de un lenguaje de programación (ej., C++, Java, Python, C#). Estas sentencias son fijas en tiempo de compilación y no cambian durante la ejecución del programa. Piensa

en ello como tener las instrucciones de la base de datos "hardcodeadas" en tu aplicación.

- **SQL Dinámico (Dynamic SQL)**
  Permite construir y ejecutar sentencias SQL en tiempo de ejecución. Esto es útil cuando no conoces la consulta exacta hasta que el programa se está ejecutando (ej., consultas generadas por el usuario, herramientas de administración de bases de datos). Es como tener un constructor de frases que te permite crear la pregunta sobre la marcha.

Sintaxis

La sintaxis específica varía mucho según el lenguaje de programación y el framework/driver de base de datos utilizado (ej., JDBC para Java, SQLAlchemy para Python, ADO.NET para C#). Sin embargo, los conceptos subyacentes son similares.

SQL Incorporado (Pseudo-código)

```
SQL
-- Supongamos una aplicación en C/Java con una conexión a la DB

EXEC SQL SELECT nombre, salario
         INTO :var_nombre, :var_salario -- Variables del lenguaje host
         FROM Empleados
         WHERE id_empleado = :input_id; -- Variable de entrada de la aplicación
         -- Esto se preprocesa y se convierte en llamadas a la API de la DB

-- Uso de Cursores para procesar múltiples filas:
EXEC SQL DECLARE CursorEmpleados CURSOR FOR
    SELECT nombre, apellido, salario
    FROM Empleados
    WHERE id_departamento = :input_depto_id;

EXEC SQL OPEN CursorEmpleados;

LOOP
    EXEC SQL FETCH CursorEmpleados INTO :emp_nombre, :emp_apellido, :emp_salario;
    -- Procesar cada fila aquí
END LOOP;

EXEC SQL CLOSE CursorEmpleados;
```

SQL Dinámico (Pseudo-código)

```
Python
-- En un lenguaje de programación (ej. Python con un conector genérico)

-- 1. Se construye la sentencia SQL como una cadena de texto en tiempo de ejecución
sql_query = "SELECT * FROM Empleados WHERE salario > ?" # Consulta parametrizada
min_salario = 50000

-- 2. Se prepara la sentencia (opcional, para mayor eficiencia si se ejecuta múltiples
veces)
prepared_statement = connection.prepare(sql_query)

-- 3. Se ejecuta la sentencia con los parámetros
results = prepared_statement.execute(min_salario)

-- O una ejecución directa de una cadena construida (menos segura si no se parametrizan)
user_input_column = "nombre"
user_input_table = "Empleados"
dynamic_sql = f"SELECT {user_input_column} FROM {user_input_table} WHERE id_empleado =
1;"
connection.execute(dynamic_sql)
```

Casos de Uso Comunes y Mejores Prácticas

- **Casos de Uso (Incorporado)**: Aplicaciones de escritorio o empresariales donde las consultas son bien conocidas y repetitivas (ej., búsqueda de un empleado por ID).
- **Casos de Uso (Dinámico)**: Herramientas de generación de informes donde el usuario define los criterios de la consulta, interfaces de administración de bases de datos, sistemas de búsqueda avanzados.
- **Mejores Prácticas**:
    - **Parametrización**: SIEMPRE usa parámetros de consulta (`?` o `:nombre_parametro`) en SQL dinámico para evitar ataques de **inyección SQL**.
    - **Manejo de Errores**: Implementa un robusto manejo de errores para capturar excepciones de base de datos.
    - **Cursores (para Incorporado)**: Úsalos para procesar grandes conjuntos de resultados fila por fila, evitando cargar todo en memoria.
- **Trampas Comunes**:
    - **Inyección SQL**: La mayor trampa del SQL dinámico no parametrizado, permitiendo que código malicioso se ejecute.
    - **Rendimiento**: La preparación de sentencias dinámicas puede tener una sobrecarga, pero el rendimiento mejora en ejecuciones repetidas.

Relación con Otros Temas

Estos temas demuestran cómo las **consultas SQL (DML)** y la **definición de datos (DDL)** se utilizan en el contexto de aplicaciones reales. El manejo de **valores nulos** y el **control de transacciones** (no

cubierto en detalle aquí, pero relevante) son cruciales al escribir código de aplicación que interactúa con la base de datos.

Tema 14. Funciones y Procedimientos Almacenados (Introducción)

Las **funciones** y **procedimientos almacenados** son bloques de código SQL (a menudo extendidos con lógica procedimental como bucles y condicionales) que se almacenan directamente en la base de datos. Una vez creados, pueden ser ejecutados repetidamente por diferentes aplicaciones o usuarios. Piensa en ellos como subrutinas o métodos dentro de tu base de datos, que encapsulan lógica de negocio.

- **Funciones Almacenadas**: Siempre devuelven un único valor y se pueden usar en expresiones SQL (ej., en `SELECT`, `WHERE`).
- **Procedimientos Almacenados**: Realizan un conjunto de operaciones (pueden insertar, actualizar, eliminar datos), pueden aceptar parámetros de entrada/salida y pueden o no devolver un conjunto de resultados.

Sintaxis

La sintaxis exacta varía significativamente entre los SGBD (ej., PL/SQL para Oracle, T-SQL para SQL Server, o SQL estándar en MySQL/PostgreSQL). Presentaremos un pseudo-código/sintaxis general.

`CREATE FUNCTION` (Ejemplo General)

```
SQL
-- Función para calcular la antigüedad de un empleado en años
CREATE FUNCTION CalcularAntiguedadAnios (
    fecha_contratacion DATE
)
RETURNS INT
AS
BEGIN
    RETURN YEAR(CURRENT_DATE) - YEAR(fecha_contratacion);
END;

-- Uso de la función en una consulta
SELECT
    nombre,
    apellido,
    CalcularAntiguedadAnios(fecha_contratacion) AS Antiguedad
FROM
    Empleados;
```

`CREATE PROCEDURE` **(Ejemplo General)**

```
SQL
-- Procedimiento para añadir un nuevo empleado
CREATE PROCEDURE InsertarEmpleado (
    IN p_nombre VARCHAR(50),
    IN p_apellido VARCHAR(50),
    IN p_id_departamento INT,
    IN p_salario DECIMAL(10, 2)
)
AS
BEGIN
    INSERT INTO Empleados (nombre, apellido, id_departamento, salario,
fecha_contratacion)
    VALUES (p_nombre, p_apellido, p_id_departamento, p_salario, CURRENT_DATE);
END;

-- Ejecutar el procedimiento
EXEC InsertarEmpleado('Carlos', 'Ruiz', 1, 75000.00); -- O CALL InsertarEmpleado(...)
dependiendo del SGBD
```

Casos de Uso Comunes y Mejores Prácticas

- **Casos de Uso**: Validar datos complejos, realizar cálculos comerciales, implementar lógica de negocio crítica, auditoría, automatizar tareas.
- **Ventajas**:
    - **Rendimiento**: Compilados y almacenados, se ejecutan más rápido.
    - **Modularidad y Reutilización**: La lógica se escribe una vez y se usa en múltiples aplicaciones.
    - **Seguridad**: Se pueden otorgar permisos para ejecutar un procedimiento sin dar acceso directo a las tablas subyacentes.
    - **Consistencia**: Asegura que la lógica de negocio se aplica de la misma manera en toda la aplicación.
- **Mejores Prácticas**:
    - **Modularización**: Dividir lógica compleja en procedimientos y funciones más pequeños y manejables.
    - **Manejo de Errores**: Incluir manejo de excepciones y errores dentro del código del procedimiento.
    - **Evitar la Lógica de Aplicación**: No poner toda la lógica de negocio en la base de datos; equilibrar dónde reside la lógica.
- **Trampas Comunes**:
    - Sobrecargar los procedimientos con demasiada lógica, dificultando el mantenimiento.
    - Problemas de permisos si no se configuran correctamente al crear/ejecutar.

Relación con Otros Temas

Las funciones y procedimientos almacenados utilizan ampliamente las sentencias **DML (`SELECT`, `INSERT`, `UPDATE`, `DELETE`)** y pueden interactuar con **vistas** y respetar **restricciones de integridad**. Se ejecutan a menudo a través de **SQL Incorporado o Dinámico** desde las aplicaciones. Son una forma avanzada de aplicar y reforzar las reglas del modelo relacional.

Esquema de Base de Datos de Ejemplo

Para los ejemplos a lo largo de esta guía, utilizaremos las siguientes tablas sencillas:

- **`Empleados`**:
  - `id_empleado` (INT, PK)
  - `nombre` (VARCHAR(50))
  - `apellido` (VARCHAR(50))
  - `id_departamento` (INT, FK a `Departamentos`)
  - `salario` (DECIMAL(10, 2))
  - `fecha_contratacion` (DATE)
  - `email` (VARCHAR(100), UNIQUE)
- **`Departamentos`**:
  - `id_departamento` (INT, PK)
  - `nombre_departamento` (VARCHAR(50), UNIQUE)
  - `ubicacion` (VARCHAR(50))
- **`Proyectos`**:
  - `id_proyecto` (INT, PK)
  - `nombre_proyecto` (VARCHAR(100))
  - `presupuesto` (DECIMAL(12, 2))
- **`Asignaciones`**:
  - `id_empleado` (INT, FK a `Empleados`, PK)
  - `id_proyecto` (INT, FK a `Proyectos`, PK)
  - `horas_trabajadas` (INT)
  - `fecha_asignacion` (DATE)