

Base de Datos

2025

Unidad II

Comandos DML

Unidad 2 – Comandos DML, Consultas Avanzadas, Funciones Agregadas, Joins, Vistas y Restricciones

2.1 Introducción al DML y manipulación de datos

¿Qué es el DML (Data Manipulation Language)?

El DML es un subconjunto del lenguaje SQL que se utiliza para operar sobre los datos ya almacenados en una base de datos. A diferencia del DDL (que define estructuras como tablas), el DML permite insertar, modificar, consultar y eliminar datos.

- ◊ Conceptos necesarios antes de aplicar DML:
 - Conocer la estructura de la tabla: columnas, tipos de datos y claves.
 - Entender la integridad referencial: qué claves foráneas pueden restringir operaciones.
 - Distinguir entre estructuras (DDL) y contenido (DML).
 - Tener noción del concepto de transacción (se verá en otra unidad, pero es importante introducirlo).
-

❖ INSERT: Agregar registros

```
INSERT INTO Productos (Nombre, Precio, Stock)  
VALUES ('Café Molido', 890.50, 120);
```

- El orden de los campos en VALUES debe coincidir con el de las columnas especificadas.
- Es posible omitir columnas con valores DEFAULT o NULL.

⌚ Nota: Si se omite una columna con restricción NOT NULL sin valor por defecto, el sistema generará un error.

❖ UPDATE: Modificar registros existentes

```
UPDATE Productos  
SET Stock = Stock - 10  
WHERE ID_Producto = 3;
```

- La cláusula WHERE es crítica. Si se omite, todos los registros serán modificados.

⚠ Error común: No usar WHERE correctamente y modificar filas no deseadas.

❖ DELETE: Eliminar registros

```
DELETE FROM Productos  
WHERE ID_Producto = 3;
```

- También depende de WHERE.
 - Puede ser bloqueado si el registro está relacionado por una clave foránea.
-

❖ TRUNCATE (mención especial)

```
TRUNCATE TABLE Productos;
```

- Borra todos los datos de una tabla sin posibilidad de filtrar.
 - Más rápido que DELETE, pero no registra detalles en logs ni activa triggers.
 - No es parte del DML estricto, pero se incluye por su efecto.
-

◊ ¿Qué son las transacciones? (introducción)

Las operaciones DML suelen agruparse en transacciones. Una transacción es un conjunto de instrucciones que se ejecutan como una unidad atómica.

```
BEGIN TRANSACTION;  
UPDATE Productos SET Stock = Stock - 1 WHERE ID = 5;  
UPDATE Inventario SET Movimiento = 'Salida' WHERE ID_Producto = 5;  
COMMIT;
```

Si algo falla, se puede hacer ROLLBACK.

⌚ Nota pedagógica: Esto será visto en profundidad en otra unidad, pero se anticipa su existencia por su relevancia en operaciones de múltiples tablas.

2.2 Agrupamiento de datos y su importancia

Antes de ver funciones agregadas, es indispensable comprender:

◊ ¿Qué es una agrupación?

Una agrupación consiste en dividir un conjunto de registros en grupos según uno o más criterios. En SQL se usa con GROUP BY.

```
SELECT ID_Cliente, COUNT(*) AS TotalPedidos  
FROM Pedidos  
GROUP BY ID_Cliente;
```

- Cada grupo es una fila única en el resultado.
 - Solo se pueden seleccionar columnas que están en el GROUP BY o agregadas con funciones como SUM, AVG, etc.
-

◊ ¿Por qué se agrupan datos?

- Para obtener estadísticas (cantidad, promedio, total).
 - Para clasificar información por categorías.
 - Para comparar entre diferentes subconjuntos.
-

◊ Conceptos asociados:

- GROUP BY: agrupa registros.
- HAVING: filtra resultados agrupados (no confundir con WHERE).
- ORDER BY: ordena los resultados finales.

```
SELECT ID_Cliente, COUNT(*) AS TotalPedidos  
FROM Pedidos  
GROUP BY ID_Cliente  
HAVING COUNT(*) > 3  
ORDER BY TotalPedidos DESC;
```

2.3 Funciones Agregadas y Funciones de Grupo

❖ ¿Qué es una función agregada?

Una función agregada toma un conjunto de valores (una columna, típicamente de un grupo) y devuelve un único resultado.

Estas funciones no operan fila por fila, sino sobre conjuntos de registros.

◊ Funciones agregadas básicas

Función	Descripción	Ejemplo básico
COUNT()	Cuenta la cantidad de registros.	COUNT(*), COUNT(ID)
SUM()	Suma los valores de una columna.	SUM(Precio)
AVG()	Calcula el promedio.	AVG(Edad)
MIN()	Devuelve el valor mínimo.	MIN(Salario)
MAX()	Devuelve el valor máximo.	MAX(Salario)

❖ Reglas importantes:

- Las funciones agregadas se suelen usar con GROUP BY, pero pueden usarse sin agrupar si queremos un único valor general.

```
SELECT AVG(Precio) AS Promedio  
FROM Productos;
```

⌚ Nota pedagógica: Esto devuelve un solo valor general. Si queremos promedios por categoría, necesitamos GROUP BY.

◊ Uso combinado con GROUP BY

```
SELECT Categoria, COUNT(*) AS TotalProductos, AVG(Precio) AS Promedio  
FROM Productos  
GROUP BY Categoria;
```

- Cada fila representa una categoría distinta.

- No se puede incluir en el SELECT una columna que no esté en el GROUP BY ni dentro de una función agregada.
-

◊ Filtro sobre resultados agrupados: HAVING

```
SELECT Categoria, COUNT(*) AS TotalProductos  
FROM Productos  
GROUP BY Categoria  
HAVING COUNT(*) > 10;
```

- WHERE filtra filas antes del agrupamiento.
- HAVING filtra después del agrupamiento, es decir, sobre los grupos resultantes.

 Error común: Intentar filtrar por COUNT() en un WHERE, lo cual genera error.

◊ Funciones de agrupación avanzadas (dependen del motor)

- Algunas bases ofrecen funciones como GROUPING SETS, CUBE y ROLLUP para obtener múltiples niveles de agregación en una misma consulta.

Ejemplo de ROLLUP en SQL Server:

```
SELECT Categoria, SubCategoria, SUM(Precio)  
FROM Productos  
GROUP BY ROLLUP (Categoria, SubCategoria);
```

Esto incluye subtotales por Categoria y un total general.

 Nota pedagógica: Estas funciones son útiles en informes gerenciales o dashboards con niveles jerárquicos.

2.4 Consultas Multitablea: JOINS

 ¿Qué es un JOIN?

Un JOIN es una operación que permite combinar datos de múltiples tablas, basándose en condiciones lógicas (típicamente claves foráneas).

◊ Tipos de JOIN

Tipo	Explicación	Resultado
INNER JOIN	Devuelve solo las coincidencias entre tablas.	Intersección.
LEFT JOIN	Devuelve todas las filas de la tabla izquierda y las coincidencias de la derecha.	Incluye NULLs si no hay coincidencia.
RIGHT JOIN	Similar al anterior, pero al revés.	
FULL OUTER JOIN	Todas las filas de ambas tablas, coincidan o no.	
CROSS JOIN	Producto cartesiano.	Todas las combinaciones posibles.

◊ Ejemplo de INNER JOIN

```
SELECT Clientes.Nombre, Pedidos.Fecha  
FROM Clientes  
INNER JOIN Pedidos ON Clientes.ID = Pedidos.ID_Cliente;  
• Une las dos tablas donde coincide el ID del cliente.  
• El resultado contiene columnas de ambas tablas.
```

◊ LEFT JOIN (útil para detectar “faltantes”)

```
SELECT Clientes.Nombre, Pedidos.Fecha  
FROM Clientes  
LEFT JOIN Pedidos ON Clientes.ID = Pedidos.ID_Cliente;  
• Muestra todos los clientes, incluso si no tienen pedidos (con NULL en la columna Fecha).
```

◊ Recomendaciones clave para Joins:

- Siempre especificar las condiciones (ON) correctamente.
 - Usar alias (C, P) para mayor claridad.
 - Para relaciones N:M, usar tablas intermedias.
-

2.5 Subconsultas (Subqueries)

❖ ¿Qué es una subconsulta?

Una subconsulta es una instrucción SELECT dentro de otra consulta, que devuelve un valor o conjunto de valores que se utilizan en la consulta principal.

◊ Tipos de subconsultas

Tipo	Ubicación	Ejemplo
Escalar	En el SELECT o WHERE, devuelve un solo valor.	SELECT (SELECT AVG(Precio) FROM Productos)
De conjunto	Devuelve múltiples valores, se usa con IN, EXISTS, etc.	WHERE ID IN (SELECT ...)
Correlacionada	Se ejecuta por cada fila de la consulta externa.	WHERE Precio > (SELECT AVG(Precio) FROM ...)

◊ Ejemplo práctico:

```
SELECT Nombre  
FROM Productos  
WHERE Precio > (  
    SELECT AVG(Precio)  
    FROM Productos  
)
```

- Muestra productos cuyo precio supera el promedio.

2.6 Vistas en SQL

❖ ¿Qué es una Vista?

Una vista es una consulta almacenada que actúa como una tabla virtual. No guarda los datos, sino la lógica de cómo mostrarlos. Se utiliza para simplificar consultas, controlar el acceso a la información y abstraer la complejidad de las tablas reales.

◊ ¿Por qué usar vistas?

- Seguridad: Podés ocultar columnas sensibles.
- Simplicidad: Reutilizás consultas complejas como si fueran tablas.
- Compatibilidad: Programas externos pueden leer desde una vista en lugar de acceder directamente a varias tablas.

☞ Nota pedagógica: Las vistas son muy útiles en sistemas multiusuario donde se debe limitar el acceso a cierta información sin modificar la lógica del sistema.

◊ Ejemplo básico de vista

```
CREATE VIEW VistaClientesActivos AS
```

```
SELECT ID, Nombre, Email
```

```
FROM Clientes
```

```
WHERE Estado = 'Activo';
```

Esta vista muestra solo los clientes activos y oculta otros datos como DNI o dirección.

◊ Modificación y eliminación

```
ALTER VIEW VistaClientesActivos AS
```

```
SELECT ID, Nombre, Email, Telefono
```

```
FROM Clientes
```

```
WHERE Estado = 'Activo';
```

```
DROP VIEW VistaClientesActivos;
```

☞ Importante: Si cambiás la estructura de las tablas base, la vista puede volverse inválida.

◊ ¿Se pueden actualizar datos desde una vista?

- Sí, pero solo si la vista no incluye:

- Joins múltiples.
- Funciones agregadas.
- DISTINCT, GROUP BY, HAVING.

⚠ Restricción común: En SQL Server y Oracle, si hacés un JOIN dentro de la vista, no podés hacer INSERT directamente, salvo excepciones.

◊ Tipos de vistas

Tipo	Descripción
Vistas simples	Basadas en una sola tabla.
Vistas complejas	Con joins, funciones, subconsultas.
Vistas materializadas (PostgreSQL, Oracle)	Guardan los resultados para mejorar el rendimiento, pero necesitan refresco.
Vistas indexadas (SQL Server)	Se puede crear un índice sobre la vista si cumple ciertos requisitos. Mejora el rendimiento en consultas.

2.7 Restricciones de Integridad

Las restricciones de integridad aseguran que los datos en la base sean coherentes, correctos y válidos.

◊ Tipos de restricciones

Tipo	Qué controla	Ejemplo
NOT NULL	Que un campo no quede vacío.	Nombre VARCHAR(50) NOT NULL
CHECK	Que los valores cumplan una condición.	Edad INT CHECK (Edad >= 18)
DEFAULT	Valor por defecto si no se especifica.	Estado VARCHAR(10) DEFAULT 'Activo'
UNIQUE	Que los valores no se repitan.	CONSTRAINT UQ_Email UNIQUE (Email)
PRIMARY KEY	Identificador único por fila.	ID INT PRIMARY KEY
FOREIGN KEY	Relación entre tablas.	ID_Cliente INT FOREIGN KEY REFERENCES Clientes(ID)

◊ Reglas de claves foráneas

Cuando usamos FOREIGN KEY, debemos decidir qué hacer cuando se elimina o actualiza un registro relacionado:

FOREIGN KEY (ID_Cliente) REFERENCES Clientes(ID)

ON DELETE CASCADE

ON UPDATE CASCADE

- CASCADE: Propaga los cambios.
- SET NULL: Pone el campo en NULL.
- NO ACTION / RESTRICT: No permite el cambio.

⌚ Buena práctica: Activar ON DELETE CASCADE solo si el borrado lógico no es necesario.

En muchos sistemas se marca con un campo “activo/inactivo”.

◊ CHECK y DEFAULT: Reglas locales de validación

Edad INT CHECK (Edad >= 18),

Estado VARCHAR(10) DEFAULT 'Activo'

- CHECK se usa para reglas simples como edades, estados, rangos.
 - DEFAULT simplifica las inserciones.
-

◊ Composición de múltiples restricciones

CREATE TABLE Empleados (

ID INT PRIMARY KEY,

Nombre VARCHAR(100) NOT NULL,

Edad INT CHECK (Edad >= 18 AND Edad <= 65),

Email VARCHAR(100) UNIQUE,

Estado VARCHAR(10) DEFAULT 'Activo'

);

Este ejemplo garantiza:

- ID único.
- Nombre obligatorio.

- Edad entre 18 y 65.
- Email único.
- Estado predeterminado.

2.8 – Funciones Escalares y de Sistema

Introducción

Las funciones escalares y de sistema son herramientas fundamentales del lenguaje SQL. Permiten **transformar, manipular y consultar valores** dentro de sentencias, sin necesidad de alterar los datos de origen. Se aplican a **campos individuales o literales**, y devuelven **un único resultado por fila**.

Estas funciones varían levemente según el motor, tanto en **nombre** como en **sintaxis**, pero la lógica general es compartida. Son esenciales para personalizar salidas, realizar cálculos, aplicar filtros complejos, trabajar con fechas y mucho más.

Clasificación de funciones escalares

Las principales categorías de funciones escalares son:

Categoría	Ejemplos de uso
Funciones de texto	Concatenar, cambiar mayúsculas, extraer subcadenas
Funciones numéricas	Redondear, valor absoluto, raíz cuadrada
Funciones de fecha	Obtener fecha actual, día, mes, diferencia entre fechas
Funciones de conversión	Convertir entre tipos (texto → número, fecha → texto)
Funciones de sistema	Obtener el usuario actual, nombre del servidor, fecha/hora actual

◆ 1. Funciones de texto

SQL Server	MySQL	PostgreSQL	Oracle
UPPER(campo)	UPPER(campo)	UPPER(campo)	UPPER(campo)
LOWER(campo)	LOWER(campo)	LOWER(campo)	LOWER(campo)
LEN(campo)	LENGTH(campo)	LENGTH(campo)	LENGTH(campo)
SUBSTRING(c, i, l)	SUBSTRING(c, i, l)	SUBSTRING(c FROM i FOR l)	SUBSTR(c, i, l)
LTRIM(RTRIM(@))	TRIM@	TRIM@	TRIM@

SQL Server	MySQL	PostgreSQL	Oracle
REPLACE(c, x, y)	REPLACE(c, x, y)	REPLACE(c, x, y)	REPLACE(c, x, y)

◊ *Nota pedagógica:*

- Muchos errores comunes provienen de confundir índices base (algunos empiezan desde 1 y otros desde 0).
- En SQL Server, LEN() **no** cuenta los espacios finales. En PostgreSQL y otros, sí.

◆ **2. Funciones numéricas**

Función	Descripción
ABS(x)	Valor absoluto
ROUND(x, n)	Redondea a n decimales
CEILING(x) / FLOOR(x)	Redondeo hacia arriba / abajo
POWER(x, y)	Potencia
SQRT(x)	Raíz cuadrada
MOD(x, y)	Módulo o resto de la división

◊ *Ejemplo multiplataforma:*

```
SELECT ROUND(123.456, 1); -- Devuelve 123.5
```

```
SELECT POWER(2, 3); -- Devuelve 8
```

◆ **3. Funciones de fecha y hora**

Función común	SQL Server	MySQL	PostgreSQL	Oracle
Fecha actual	GETDATE()	NOW()	NOW()	SYSDATE
Extraer año/mes/día	YEAR(...)	YEAR(...)	EXTRACT(YEAR FROM ...)	EXTRACT(YEAR FROM ...)
Sumar días/meses	DATEADD(day, n, f)	DATE_ADD(f, INTERVAL n DAY)	f + INTERVAL 'n day'	f + n (días) o ADD_MONTHS(f, n)

◊ *Notas útiles:*

- Las fechas pueden tener formatos diferentes según motor o configuración regional.
- Oracle tiene reglas propias para operaciones con fechas, por ejemplo: SYSDATE + 1 suma 1 día.

◆ **4. Funciones de conversión**

Función	Uso principal
CAST(x AS tipo)	Conversión estándar y portable
CONVERT(tipo, x)	Disponible en SQL Server
TO_CHAR(x) / TO_DATE(x)	Específicas de Oracle

◊ *Ejemplo:*

sql

CopiarEditar

SELECT CAST('2024-01-01' AS DATE); -- En general

SELECT TO_DATE('01-ENE-2025', 'DD-MON-YYYY') FROM dual; -- Oracle

◆ 5. Funciones de sistema

Estas funciones son propias de cada motor, útiles para auditorías, control de usuarios o tareas automatizadas.

Motor Funciones frecuentes

SQL Server SUSER_NAME(), GETDATE(), @@SERVERNAME

MySQL CURRENT_USER(), NOW(), DATABASE()

PostgreSQL CURRENT_USER, CURRENT_DATE, VERSION()

Oracle USER, SYSDATE, UID

◆ Consideraciones comunes a todos los motores

Error habitual

Usar función de un motor en otro (e.g., SYSDATE en SQL Server)

Confundir tipos de datos en conversión

Asumir que ROUND se comporta igual en todos

Recomendación pedagógica

Mostrar tabla comparativa al alumno

Trabajar con ejemplos de errores y su solución

Comparar resultados con diferentes parámetros

2.9 Ejercitación

- ◊ Tablas

SQL Server

```
-- =====
```

```
-- CREACIÓN DE TABLAS
```

```
-- =====
```

```
CREATE TABLE Clientes (
```

```
    ID_Cliente INT PRIMARY KEY,  
    Nombre NVARCHAR(50) NOT NULL,  
    Ciudad NVARCHAR(50),  
    Email NVARCHAR(100) UNIQUE
```

```
);
```

```
CREATE TABLE Empleados (
```

```
    ID_Emppleado INT PRIMARY KEY,  
    Nombre NVARCHAR(50),  
    Cargo NVARCHAR(50),  
    Salario DECIMAL(10,2) CHECK (Salario >= 0)
```

```
);
```

```
CREATE TABLE Sucursales (
```

```
    ID_Sucursal INT PRIMARY KEY,  
    Nombre NVARCHAR(50),  
    Ciudad NVARCHAR(50)
```

```
);
```

```
CREATE TABLE Productos (
```

```
    ID_Producto INT PRIMARY KEY,  
    Nombre NVARCHAR(50),  
    Precio DECIMAL(10,2) CHECK (Precio > 0),  
    Stock INT DEFAULT 0
```

```
);
```

```
CREATE TABLE Pedidos (
```

```
    ID_Pedido INT PRIMARY KEY,  
    ID_Cliente INT,  
    ID_Emppleado INT,  
    ID_Sucursal INT,  
    Fecha DATE DEFAULT GETDATE(),
```

```
    FOREIGN KEY (ID_Cliente) REFERENCES Clientes(ID_Cliente),
    FOREIGN KEY (ID_Emppleado) REFERENCES Empleados(ID_Emppleado),
    FOREIGN KEY (ID_Sucursal) REFERENCES Sucursales(ID_Sucursal)
);
```

```
CREATE TABLE DetallePedidos (
    ID_Pedido INT,
    ID_Producto INT,
    Cantidad INT CHECK (Cantidad > 0),
    PRIMARY KEY (ID_Pedido, ID_Producto),
    FOREIGN KEY (ID_Pedido) REFERENCES Pedidos(ID_Pedido),
    FOREIGN KEY (ID_Producto) REFERENCES Productos(ID_Producto)
);
```

```
-- =====
```

```
-- INSERTAR DATOS
```

```
-- =====
```

```
INSERT INTO Clientes VALUES
```

```
(1, 'Juan Pérez', 'Madrid', 'juan.perez@mail.com'),
(2, 'Ana Gómez', 'Barcelona', 'ana.gomez@mail.com'),
(3, 'Luis Martínez', 'Sevilla', 'luis.martinez@mail.com');
```

```
INSERT INTO Empleados VALUES
```

```
(1, 'Carlos Ruiz', 'Vendedor', 1500.00),
(2, 'Marta Sánchez', 'Gerente', 2500.00);
```

```
INSERT INTO Sucursales VALUES
```

```
(1, 'Sucursal Centro', 'Madrid'),
(2, 'Sucursal Norte', 'Barcelona');
```

```
INSERT INTO Productos VALUES
```

```
(1, 'Laptop', 1200.00, 10),
(2, 'Mouse', 20.00, 100),
(3, 'Teclado', 35.00, 50),
(4, 'Monitor', 200.00, 25);
```

```
INSERT INTO Pedidos VALUES
```

```
(1, 1, 1, 1, '2025-08-01'),
(2, 2, 2, 2, '2025-08-15');
```

```
INSERT INTO DetallePedidos VALUES
```

(1, 1, 1),
(1, 2, 2),
(2, 3, 1),
(2, 4, 1);

2. MySQL

```
-- =====
```

```
-- CREACIÓN DE TABLAS
```

```
-- =====
```

```
CREATE TABLE Clientes (
    ID_Cliente INT PRIMARY KEY,
    Nombre VARCHAR(50) NOT NULL,
    Ciudad VARCHAR(50),
    Email VARCHAR(100) UNIQUE
);
```

```
CREATE TABLE Empleados (
    ID_Emppleado INT PRIMARY KEY,
    Nombre VARCHAR(50),
    Cargo VARCHAR(50),
    Salario DECIMAL(10,2) CHECK (Salario >= 0)
);
```

```
CREATE TABLE Sucursales (
    ID_Sucursal INT PRIMARY KEY,
    Nombre VARCHAR(50),
    Ciudad VARCHAR(50)
);
```

```
CREATE TABLE Productos (
    ID_Producto INT PRIMARY KEY,
    Nombre VARCHAR(50),
    Precio DECIMAL(10,2) CHECK (Precio > 0),
    Stock INT DEFAULT 0
) ENGINE=InnoDB;
```

```
CREATE TABLE Pedidos (
    ID_Pedido INT PRIMARY KEY,
    ID_Cliente INT,
```

```
ID_Emppleado INT,  
ID_Sucursal INT,  
Fecha DATE DEFAULT (CURRENT_DATE),  
FOREIGN KEY (ID_Cliente) REFERENCES Clientes(ID_Cliente),  
FOREIGN KEY (ID_Emppleado) REFERENCES Empleados(ID_Emppleado),  
FOREIGN KEY (ID_Sucursal) REFERENCES Sucursales(ID_Sucursal)  
);  
  
CREATE TABLE DetallePedidos (  
    ID_Pedido INT,  
    ID_Producto INT,  
    Cantidad INT CHECK (Cantidad > 0),  
    PRIMARY KEY (ID_Pedido, ID_Producto),  
    FOREIGN KEY (ID_Pedido) REFERENCES Pedidos(ID_Pedido),  
    FOREIGN KEY (ID_Producto) REFERENCES Productos(ID_Producto)  
);  
(los inserts son idénticos a SQL Server salvo por el DEFAULT de fechas → CURRENT_DATE en lugar de GETDATE())  
  
-- =====  
-- INSERTS DE EJEMPLO  
-- =====  
INSERT INTO Clientes VALUES  
(1, 'Juan Pérez', 'Madrid', 'juan.perez@mail.com'),  
(2, 'Ana Gómez', 'Barcelona', 'ana.gomez@mail.com'),  
(3, 'Luis Martínez', 'Sevilla', 'luis.martinez@mail.com');  
  
INSERT INTO Empleados VALUES  
(1, 'Carlos Ruiz', 'Vendedor', 1500.00),  
(2, 'Marta Sánchez', 'Gerente', 2500.00);  
  
INSERT INTO Sucursales VALUES  
(1, 'Sucursal Centro', 'Madrid'),  
(2, 'Sucursal Norte', 'Barcelona');  
  
INSERT INTO Productos VALUES  
(1, 'Laptop', 1200.00, 10),  
(2, 'Mouse', 20.00, 100),  
(3, 'Teclado', 35.00, 50),  
(4, 'Monitor', 200.00, 25);
```

```
INSERT INTO Pedidos VALUES  
(1, 1, 1, 1, '2025-08-01'),  
(2, 2, 2, 2, '2025-08-15');
```

```
INSERT INTO DetallePedidos VALUES  
(1, 1, 1),  
(1, 2, 2),  
(2, 3, 1),  
(2, 4, 1);
```

3. PostgreSQL

```
-- =====
```

```
-- CREACIÓN DE TABLAS
```

```
-- =====
```

```
CREATE TABLE Clientes (  
    ID_Cliente INT PRIMARY KEY,  
    Nombre VARCHAR(50) NOT NULL,  
    Ciudad VARCHAR(50),  
    Email VARCHAR(100) UNIQUE  
);
```

```
CREATE TABLE Empleados (  
    ID_Empelado INT PRIMARY KEY,  
    Nombre VARCHAR(50),  
    Cargo VARCHAR(50),  
    Salario NUMERIC(10,2) CHECK (Salario >= 0)  
);
```

```
CREATE TABLE Sucursales (  
    ID_Sucursal INT PRIMARY KEY,  
    Nombre VARCHAR(50),  
    Ciudad VARCHAR(50)  
);
```

```
CREATE TABLE Productos (  
    ID_Producto INT PRIMARY KEY,  
    Nombre VARCHAR(50),  
    Precio NUMERIC(10,2) CHECK (Precio > 0),  
    Stock INT DEFAULT 0
```

);

```
CREATE TABLE Pedidos (
    ID_Pedido INT PRIMARY KEY,
    ID_Cliente INT REFERENCES Clientes(ID_Cliente),
    ID_Emppleado INT REFERENCES Empleados(ID_Emppleado),
    ID_Sucursal INT REFERENCES Sucursales(ID_Sucursal),
    Fecha DATE DEFAULT CURRENT_DATE
);
```

```
CREATE TABLE DetallePedidos (
    ID_Pedido INT,
    ID_Producto INT,
    Cantidad INT CHECK (Cantidad > 0),
    PRIMARY KEY (ID_Pedido, ID_Producto),
    FOREIGN KEY (ID_Pedido) REFERENCES Pedidos(ID_Pedido),
    FOREIGN KEY (ID_Producto) REFERENCES Productos(ID_Producto)
);
```

-- =====

-- INSERTS DE EJEMPLO

-- =====

INSERT INTO Clientes VALUES

```
(1, 'Juan Pérez', 'Madrid', 'juan.perez@mail.com'),
(2, 'Ana Gómez', 'Barcelona', 'ana.gomez@mail.com'),
(3, 'Luis Martínez', 'Sevilla', 'luis.martinez@mail.com');
```

INSERT INTO Empleados VALUES

```
(1, 'Carlos Ruiz', 'Vendedor', 1500.00),
(2, 'Marta Sánchez', 'Gerente', 2500.00);
```

INSERT INTO Sucursales VALUES

```
(1, 'Sucursal Centro', 'Madrid'),
(2, 'Sucursal Norte', 'Barcelona');
```

INSERT INTO Productos VALUES

```
(1, 'Laptop', 1200.00, 10),
(2, 'Mouse', 20.00, 100),
(3, 'Teclado', 35.00, 50),
(4, 'Monitor', 200.00, 25);
```

```
INSERT INTO Pedidos VALUES  
(1, 1, 1, 1, '2025-08-01'),  
(2, 2, 2, 2, '2025-08-15');
```

```
INSERT INTO DetallePedidos VALUES  
(1, 1, 1),  
(1, 2, 2),  
(2, 3, 1),  
(2, 4, 1);
```

4. Oracle

```
-- =====  
-- CREACIÓN DE TABLAS  
-- =====  
CREATE TABLE Clientes (  
    ID_Cliente NUMBER PRIMARY KEY,  
    Nombre VARCHAR2(50) NOT NULL,  
    Ciudad VARCHAR2(50),  
    Email VARCHAR2(100) UNIQUE  
);  
  
CREATE TABLE Empleados (  
    ID_Empelado NUMBER PRIMARY KEY,  
    Nombre VARCHAR2(50),  
    Cargo VARCHAR2(50),  
    Salario NUMBER(10,2) CHECK (Salario >= 0)  
);  
  
CREATE TABLE Sucursales (  
    ID_Sucursal NUMBER PRIMARY KEY,  
    Nombre VARCHAR2(50),  
    Ciudad VARCHAR2(50)  
);  
  
CREATE TABLE Productos (  
    ID_Producto NUMBER PRIMARY KEY,  
    Nombre VARCHAR2(50),  
    Precio NUMBER(10,2) CHECK (Precio > 0),  
    Stock NUMBER DEFAULT 0
```

);

```
CREATE TABLE Pedidos (
    ID_Pedido NUMBER PRIMARY KEY,
    ID_Cliente NUMBER REFERENCES Clientes(ID_Cliente),
    ID_Emppleado NUMBER REFERENCES Empleados(ID_Emppleado),
    ID_Sucursal NUMBER REFERENCES Sucursales(ID_Sucursal),
    Fecha DATE DEFAULT SYSDATE
);
```

```
CREATE TABLE DetallePedidos (
    ID_Pedido NUMBER,
    ID_Producto NUMBER,
    Cantidad NUMBER CHECK (Cantidad > 0),
    CONSTRAINT pk_detalle PRIMARY KEY (ID_Pedido, ID_Producto),
    CONSTRAINT fk_pedido FOREIGN KEY (ID_Pedido) REFERENCES Pedidos(ID_Pedido),
    CONSTRAINT fk_producto FOREIGN KEY (ID_Producto) REFERENCES
    Productos(ID_Producto)
);
```

-- =====

-- INSERTS DE EJEMPLO

-- =====

```
INSERT INTO Clientes VALUES
(1, 'Juan Pérez', 'Madrid', 'juan.perez@mail.com');
INSERT INTO Clientes VALUES
(2, 'Ana Gómez', 'Barcelona', 'ana.gomez@mail.com');
INSERT INTO Clientes VALUES
(3, 'Luis Martínez', 'Sevilla', 'luis.martinez@mail.com');
```

```
INSERT INTO Empleados VALUES
(1, 'Carlos Ruiz', 'Vendedor', 1500.00);
INSERT INTO Empleados VALUES
(2, 'Marta Sánchez', 'Gerente', 2500.00);
```

```
INSERT INTO Sucursales VALUES
(1, 'Sucursal Centro', 'Madrid');
INSERT INTO Sucursales VALUES
(2, 'Sucursal Norte', 'Barcelona');
```

```
INSERT INTO Productos VALUES
```

```
(1, 'Laptop', 1200.00, 10);
INSERT INTO Productos VALUES
(2, 'Mouse', 20.00, 100);
INSERT INTO Productos VALUES
(3, 'Teclado', 35.00, 50);
INSERT INTO Productos VALUES
(4, 'Monitor', 200.00, 25);

INSERT INTO Pedidos VALUES
(1, 1, 1, 1, TO_DATE('2025-08-01', 'YYYY-MM-DD'));
INSERT INTO Pedidos VALUES
(2, 2, 2, 2, TO_DATE('2025-08-15', 'YYYY-MM-DD'));

INSERT INTO DetallePedidos VALUES
(1, 1, 1);
INSERT INTO DetallePedidos VALUES
(1, 2, 2);
INSERT INTO DetallePedidos VALUES
(2, 3, 1);
INSERT INTO DetallePedidos VALUES
(2, 4, 1);
```

◊ Nivel FÁCIL – 2.1 – SQL Server

Ejercicio 1:

```
INSERT INTO Clientes (Nombre, Email, Telefono)
VALUES ('Carlos López', 'carlos.lopez@email.com', '123456789');
```

Ejercicio 2:

```
UPDATE Clientes
SET Telefono = '987654321'
WHERE Nombre = 'Carlos López';
```

◊ Nivel FÁCIL – 2.1 – MySQL

Ejercicio 1:

```
INSERT INTO Clientes (Nombre, Email, Telefono)
VALUES ('Laura Méndez', 'laura.mendez@email.com', '456789123');
```

Ejercicio 2:

DELETE FROM Clientes
WHERE Nombre = 'Laura Méndez';

◊ Nivel FÁCIL – 2.1 – PostgreSQL

Ejercicio 1:

```
INSERT INTO Clientes (Nombre, Email, Telefono)  
VALUES ('Pedro Gómez', 'pedro.gomez@email.com', '321654987');
```

Ejercicio 2:

```
SELECT * FROM Clientes  
WHERE Email LIKE '%@email.com';
```

◊ Nivel FÁCIL – 2.1 – Oracle

Ejercicio 1:

```
INSERT INTO Clientes (ID_Cliente, Nombre, Email, Telefono)  
VALUES (1001, 'Ana Torres', 'ana.torres@email.com', '789123456');
```

Ejercicio 2:

```
UPDATE Clientes  
SET Telefono = '111222333'  
WHERE ID_Cliente = 1001;
```

◊ Nivel INTERMEDIO – 2.1 – SQL Server

Ejercicio 1:

Insertar múltiples productos en una sola sentencia.

```
INSERT INTO Productos (Nombre, Precio, Stock)  
VALUES  
(‘Teclado’, 3000, 50),  
(‘Mouse’, 1500, 100),  
(‘Monitor’, 25000, 30);
```

Ejercicio 2:

Actualizar precios con una condición numérica.

```
UPDATE Productos  
SET Precio = Precio * 1.1  
WHERE Precio < 2000;
```

◊ Nivel INTERMEDIO – 2.1 – MySQL

Ejercicio 1:

Eliminar productos sin stock.

DELETE FROM Productos

WHERE Stock = 0;

Ejercicio 2:

Modificar el stock de un producto específico.

UPDATE Productos

SET Stock = Stock + 10

WHERE Nombre = 'Mouse';

◊ Nivel INTERMEDIO – 2.1 – PostgreSQL

Ejercicio 1:

Insertar datos usando RETURNING para obtener el ID generado automáticamente.

INSERT INTO Clientes (Nombre, Email)

VALUES ('Marina Ruiz', 'marina@email.com')

RETURNING ID_Cliente;

Ejercicio 2:

Actualizar múltiples columnas de un producto.

UPDATE Productos

SET Precio = 3200, Stock = 60

WHERE Nombre = 'Teclado';

◊ Nivel INTERMEDIO – 2.1 – Oracle

Ejercicio 1:

Usar MERGE para actualizar o insertar registros según corresponda.

MERGE INTO Clientes c

USING (SELECT 101 AS ID_Cliente FROM dual) src

ON (c.ID_Cliente = src.ID_Cliente)

WHEN MATCHED THEN

UPDATE SET Nombre = 'Actualizado Oracle'

WHEN NOT MATCHED THEN

INSERT (ID_Cliente, Nombre)

VALUES (101, 'Nuevo Cliente');

Ejercicio 2:

Eliminar múltiples filas con condición de texto.

DELETE FROM Clientes

WHERE Email LIKE '%@spam.com';

- ◊ Nivel DIFICIL – 2.1 – SQL Server

Ejercicio 1:

Actualizar el stock basado en otra tabla relacionada.

UPDATE P

SET P.Stock = P.Stock - D.Cantidad

FROM Productos P

JOIN DetalleVentas D ON P.ID_Producto = D.ID_Producto

WHERE D.ID_Venta = 5;

Ejercicio 2:

Eliminar todos los pedidos de clientes inactivos.

DELETE P

FROM Pedidos P

JOIN Clientes C ON P.ID_Cliente = C.ID_Cliente

WHERE C.Estado = 'Inactivo';

- ◊ Nivel DIFICIL – 2.1 – MySQL

Ejercicio 1:

Insertar datos solo si no existen (usando INSERT IGNORE).

INSERT IGNORE INTO Clientes (ID_Cliente, Nombre, Email)

VALUES (1002, 'Luis Ramos', 'luis@email.com');

Ejercicio 2:

Eliminar productos no vendidos (requiere subconsulta).

DELETE FROM Productos

WHERE ID_Producto NOT IN (

SELECT DISTINCT ID_Producto FROM DetalleVentas

);

◊ Nivel DIFÍCIL – 2.1 – PostgreSQL

Ejercicio 1:

Actualizar con subconsulta correlacionada.

UPDATE Productos

SET Stock = Stock - (

SELECT SUM(Cantidad)

FROM DetalleVentas

WHERE DetalleVentas.ID_Producto = Productos.ID_Producto

)

WHERE ID_Producto IN (

SELECT ID_Producto FROM DetalleVentas

);

Ejercicio 2:

Insertar múltiples registros usando WITH (CTE).

WITH nuevos_productos AS (

SELECT 'Impresora'::text AS Nombre, 15000 AS Precio, 20 AS Stock

UNION ALL

SELECT 'Tablet', 18000, 15

)

INSERT INTO Productos (Nombre, Precio, Stock)

SELECT * FROM nuevos_productos;

◊ Nivel DIFÍCIL – 2.1 – Oracle

Ejercicio 1:

Eliminar con subconsulta y condiciones avanzadas.

DELETE FROM Productos

WHERE ID_Producto IN (

SELECT ID_Producto

FROM DetalleVentas

GROUP BY ID_Producto

HAVING SUM(Cantidad) < 5

);

Ejercicio 2:

Usar MERGE con condiciones complejas.

MERGE INTO Productos P

USING (SELECT 201 AS ID_Producto, 'Nuevo Producto' AS Nombre FROM dual) NP

ON (P.ID_Producto = NP.ID_Producto)

```
WHEN MATCHED THEN  
UPDATE SET P.Nombre = NP.Nombre  
WHEN NOT MATCHED THEN  
INSERT (ID_Producto, Nombre)  
VALUES (201, 'Nuevo Producto');
```

◊ Nivel FÁCIL – 2.2 - SQL Server

Ejercicio 1:

Obtener la cantidad de productos por categoría.

```
SELECT Categoria, COUNT(*) AS Cantidad  
FROM Productos  
GROUP BY Categoria;
```

Ejercicio 2:

Listar la cantidad de pedidos por cada cliente.

```
SELECT ID_Cliente, COUNT(*) AS TotalPedidos  
FROM Pedidos  
GROUP BY ID_Cliente;
```

◊ Nivel FÁCIL – 2.2 - MySQL

Ejercicio 1:

Cantidad de empleados por departamento.

```
SELECT Departamento, COUNT(*) AS Total  
FROM Empleados  
GROUP BY Departamento;
```

Ejercicio 2:

Total de facturas emitidas por cada mes.

```
SELECT MONTH(Fecha) AS Mes, COUNT(*) AS Facturas  
FROM Facturas  
GROUP BY MONTH(Fecha);
```

◊ Nivel FÁCIL – 2.2 - PostgreSQL

Ejercicio 1:

Clientes agrupados por ciudad.

```
SELECT Ciudad, COUNT(*) AS TotalClientes
```

```
FROM Clientes  
GROUP BY Ciudad;
```

Ejercicio 2:

Cantidad de productos por proveedor.

```
SELECT ID_Proveedor, COUNT(*) AS Productos  
FROM Productos  
GROUP BY ID_Proveedor;
```

◊ Nivel FÁCIL – 2.2 - Oracle

Ejercicio 1:

Cantidad de alumnos por carrera.

```
SELECT Carrera, COUNT(*) AS Total  
FROM Alumnos  
GROUP BY Carrera;
```

Ejercicio 2:

Total de órdenes por tipo de envío.

```
SELECT Tipo_Envio, COUNT(*) AS Ordenes  
FROM Ordenes  
GROUP BY Tipo_Envio;
```

◊ Nivel INTERMEDIO – 2.2 – SQL Server

Ejercicio 1:

Obtener el total de ventas por cliente y filtrar los que superan \$10.000.

```
SELECT ID_Cliente, SUM(Total) AS TotalComprado  
FROM Ventas  
GROUP BY ID_Cliente  
HAVING SUM(Total) > 10000;
```

Ejercicio 2:

Promedio de sueldos por departamento.

```
SELECT Departamento, AVG(Sueldo) AS Promedio  
FROM Empleados  
GROUP BY Departamento;
```

◊ Nivel INTERMEDIO – 2.2 – MySQL

Ejercicio 1:

Mostrar cantidad de productos por marca con más de 5 productos.

```
SELECT Marca, COUNT(*) AS Cantidad  
FROM Productos  
GROUP BY Marca  
HAVING COUNT(*) > 5;
```

Ejercicio 2:

Total vendido por año.

```
SELECT YEAR(Fecha) AS Año, SUM(Total) AS TotalAnual  
FROM Ventas  
GROUP BY YEAR(Fecha);
```

◊ Nivel INTERMEDIO – 2.2 – PostgreSQL

Ejercicio 1:

Clientes que realizaron más de 3 compras.

```
SELECT ID_Cliente, COUNT(*) AS Compras  
FROM Pedidos  
GROUP BY ID_Cliente  
HAVING COUNT(*) > 3;
```

Ejercicio 2:

Valor promedio de facturas por sucursal.

```
SELECT Sucursal, AVG(Total) AS Promedio  
FROM Facturas  
GROUP BY Sucursal;
```

◊ Nivel INTERMEDIO – 2.2 – Oracle

Ejercicio 1:

Ventas por región superiores a \$20.000.

```
SELECT Region, SUM(Monto) AS TotalRegion  
FROM Ventas  
GROUP BY Region  
HAVING SUM(Monto) > 20000;
```

Ejercicio 2:

Número de inscripciones por curso.

```
SELECT ID_Curso, COUNT(*) AS Inscriptos  
FROM Inscripciones  
GROUP BY ID_Curso;
```

◊ Nivel DIFÍCIL – 2.2 – SQL Server

Ejercicio 1:

Mostrar el producto más vendido por cada categoría.

```
SELECT Categoria, Nombre, MAX(CantidadVendida) AS MaxVentas  
FROM Productos  
GROUP BY Categoria, Nombre  
HAVING MAX(CantidadVendida) = (  
SELECT MAX(CantidadVendida)  
FROM Productos AS P2  
WHERE P2.Categoria = Productos.Categoria  
);
```

Ejercicio 2:

Total de ventas mensuales por cliente solo si superó el promedio general.

```
SELECT ID_Cliente, MONTH(Fecha) AS Mes, SUM(Total) AS TotalMensual  
FROM Ventas  
GROUP BY ID_Cliente, MONTH(Fecha)  
HAVING SUM(Total) > (  
SELECT AVG(Total) FROM Ventas  
);
```

◊ Nivel DIFÍCIL – 2.2 – MySQL

Ejercicio 1:

Total vendido por cliente y mes, mostrando solo si superó \$5000.

```
SELECT ID_Cliente, MONTH(Fecha) AS Mes, SUM(Total) AS Total  
FROM Ventas  
GROUP BY ID_Cliente, MONTH(Fecha)  
HAVING Total > 5000;
```

Ejercicio 2:

Mostrar el total de ventas y número de productos distintos por sucursal.

```
SELECT Sucursal, COUNT(DISTINCT ID_Producto) AS ProductosVendidos, SUM(Total) AS  
Total  
FROM DetalleVentas
```

GROUP BY Sucursal;

- ◊ Nivel DIFÍCIL – 2.2 – PostgreSQL

Ejercicio 1:

Total vendido por producto y cliente en el último año.

```
SELECT ID_Producto, ID_Cliente, SUM(Cantidad * Precio) AS Total  
FROM Ventas  
WHERE Fecha >= CURRENT_DATE - INTERVAL '1 year'  
GROUP BY ID_Producto, ID_Cliente;
```

Ejercicio 2:

Empleados con salario mayor al promedio de su área.

```
SELECT ID_Emppleado, Departamento, Sueldo  
FROM Empleados  
WHERE Sueldo > (  
    SELECT AVG(Sueldo)  
    FROM Empleados AS E2  
    WHERE E2.Departamento = Empleados.Departamento  
);
```

- ◊ Nivel DIFÍCIL – 2.2 – Oracle

Ejercicio 1:

Cursos con más inscriptos que el promedio.

```
SELECT ID_Curso, COUNT(*) AS TotalInscriptos  
FROM Inscripciones  
GROUP BY ID_Curso  
HAVING COUNT(*) > (  
    SELECT AVG(Cantidad)  
    FROM (   
        SELECT COUNT(*) AS Cantidad  
        FROM Inscripciones  
        GROUP BY ID_Curso  
    )  
);
```

Ejercicio 2:

Total de órdenes por cliente por cada tipo de pago.

```
SELECT ID_Cliente, Tipo_Pago, COUNT(*) AS Ordenes
```

```
FROM Ordenes  
GROUP BY ID_Cliente, Tipo_Pago;
```

◊ Nivel FÁCIL – 2.3 – SQL Server

Ejercicio 1:

Obtener el total de ventas registradas.

```
SELECT SUM(Total) AS TotalVentas FROM Ventas;
```

Ejercicio 2:

Determinar el número total de productos en la tabla.

```
SELECT COUNT(*) AS TotalProductos FROM Productos;
```

◊ Nivel FÁCIL – 2.3 – MySQL

Ejercicio 1:

Obtener el salario promedio de los empleados.

```
SELECT AVG(Salario) AS PromedioSalario FROM Empleados;
```

Ejercicio 2:

Determinar el precio mínimo y máximo de productos.

```
SELECT MIN(Precio) AS PrecioMinimo, MAX(Precio) AS PrecioMaximo  
FROM Productos;
```

◊ Nivel FÁCIL – 2.3 – PostgreSQL

Ejercicio 1:

Contar la cantidad de inscripciones en total.

```
SELECT COUNT(*) AS TotalInscriptos FROM Inscripciones;
```

Ejercicio 2:

Obtener el promedio de duración de los cursos.

```
SELECT AVG(Duracion) AS PromedioDuracion FROM Cursos;
```

◊ Nivel FACIL – 2.3 – Oracle

Ejercicio 1:

Cantidad total de pedidos realizados.

```
SELECT COUNT(*) AS TotalPedidos FROM Pedidos;
```

Ejercicio 2:

Monto máximo de una factura.

```
SELECT MAX(Monto) AS MayorFactura FROM Facturas;
```

◊ Nivel INTERMEDIO – 2.3 – SQL Server

Ejercicio 1:

Mostrar el total de ventas por sucursal.

```
SELECT Sucursal, SUM(Total) AS TotalSucursal  
FROM Ventas  
GROUP BY Sucursal;
```

Ejercicio 2:

Promedio de calificaciones por curso.

```
SELECT ID_Curso, AVG(Calificacion) AS Promedio  
FROM Evaluaciones  
GROUP BY ID_Curso;
```

◊ Nivel INTERMEDIO – 2.3 – MySQL

Ejercicio 1:

Clientes que han realizado más de 5 compras.

```
SELECT ID_Cliente, COUNT(*) AS CantidadCompras  
FROM Ventas  
GROUP BY ID_Cliente  
HAVING COUNT(*) > 5;
```

Ejercicio 2:

Obtener la suma de sueldos por departamento.

```
SELECT Departamento, SUM(Sueldo) AS TotalSueldos  
FROM Empleados  
GROUP BY Departamento;
```

◊ Nivel INTERMEDIO – 2.3 – PostgreSQL

Ejercicio 1:

Cantidad de alumnos por carrera con más de 10 alumnos.

```
SELECT Carrera, COUNT(*) AS Total  
FROM Alumnos  
GROUP BY Carrera  
HAVING COUNT(*) > 10;
```

Ejercicio 2:

Precio promedio por categoría de productos.

```
SELECT Categoria, AVG(Precio) AS Promedio  
FROM Productos  
GROUP BY Categoria;
```

◊ Nivel INTERMEDIO – 2.3 – Oracle

Ejercicio 1:

Mostrar el total vendido por vendedor.

```
SELECT ID_Vendedor, SUM(Monto) AS Total  
FROM Ventas  
GROUP BY ID_Vendedor;
```

Ejercicio 2:

Promedio de notas por estudiante.

```
SELECT ID_Estudiante, AVG(Nota) AS Promedio  
FROM Calificaciones  
GROUP BY ID_Estudiante;
```

◊ Nivel DIFÍCIL – 2.3 – SQL Server

Ejercicio 1:

Para cada cliente, mostrar su total de ventas y si supera el promedio general.

```
SELECT ID_Cliente, SUM(Total) AS TotalComprado,  
CASE  
WHEN SUM(Total) > (SELECT AVG(Total) FROM Ventas)  
THEN 'Sobre Promedio'  
ELSE 'Bajo Promedio'  
END AS Comparacion
```

```
FROM Ventas  
GROUP BY ID_Cliente;
```

Ejercicio 2:

Obtener la venta más alta de cada vendedor en el último trimestre.

```
SELECT ID_Vendedor, MAX(Total) AS VentaMaxima  
FROM Ventas  
WHERE Fecha >= DATEADD(MONTH, -3, GETDATE())  
GROUP BY ID_Vendedor;
```

◊ Nivel DIFÍCIL – 2.3 – MySQL

Ejercicio 1:

Mostrar el promedio de ventas mensuales por región si superan los \$1000.

```
SELECT Region, MONTH(Fecha) AS Mes, AVG(Total) AS Promedio  
FROM Ventas  
GROUP BY Region, MONTH(Fecha)  
HAVING AVG(Total) > 1000;
```

Ejercicio 2:

Determinar el cliente con la mayor compra por sucursal.

```
SELECT Sucursal, ID_Cliente, MAX(Total) AS MayorCompra  
FROM Ventas  
GROUP BY Sucursal, ID_Cliente;
```

◊ Nivel DIFÍCIL – 2.3 – PostgreSQL

Ejercicio 1:

Cantidad de productos únicos comprados por cliente y sucursal.

```
SELECT ID_Cliente, Sucursal, COUNT(DISTINCT ID_Producto) AS ProductosUnicos  
FROM Ventas  
GROUP BY ID_Cliente, Sucursal;
```

Ejercicio 2:

Promedio de precios y total de productos por marca.

```
SELECT Marca, AVG(Precio) AS Promedio, COUNT(*) AS Cantidad  
FROM Productos  
GROUP BY Marca;
```

◊ Nivel DIFÍCIL – 2.3 – Oracle

Ejercicio 1:

Cursos con inscriptos por encima del promedio general.

```
SELECT ID_Curso, COUNT(*) AS Inscriptos
FROM Inscripciones
GROUP BY ID_Curso
HAVING COUNT(*) > (
    SELECT AVG(Cantidad)
    FROM (
        SELECT COUNT(*) AS Cantidad
        FROM Inscripciones
        GROUP BY ID_Curso
    )
);
```

Ejercicio 2:

Mostrar la venta mínima, máxima y promedio por trimestre.

```
SELECT TO_CHAR(Fecha, 'Q') AS Trimestre,
MIN(Total) AS Minimo,
MAX(Total) AS Maximo,
AVG(Total) AS Promedio
FROM Ventas
GROUP BY TO_CHAR(Fecha, 'Q');
```

◊ Nivel FÁCIL – 2.4 – SQL Server

Ejercicio 1:

Listar los nombres de los clientes y sus pedidos.

```
SELECT C.Nombre, P.ID_Pedido
FROM Clientes C
INNER JOIN Pedidos P ON C.ID_Cliente = P.ID_Cliente;
```

Ejercicio 2:

Mostrar todos los productos y su categoría.

```
SELECT P.Nombre, C.Nombre AS Categoria
FROM Productos P
INNER JOIN Categorias C ON P.ID_Categoría = C.ID_Categoría;
```

◊ Nivel FACIL – 2.4 – MySQL

Ejercicio 1:

Obtener los cursos y sus respectivos profesores.

```
SELECT C.NombreCurso, P.Nombre  
FROM Cursos C  
INNER JOIN Profesores P ON C.ID_Profesor = P.ID_Profesor;
```

Ejercicio 2:

Mostrar nombre de estudiantes y sus carreras.

```
SELECT E.Nombre, C.Nombre AS Carrera  
FROM Estudiantes E  
JOIN Carreras C ON E.ID_Carrera = C.ID_Carrera;
```

◊ Nivel FACIL – 2.4 – PostgreSQL

Ejercicio 1:

Clientes y sus ciudades.

```
SELECT C.Nombre, Ci.Nombre AS Ciudad  
FROM Clientes C  
JOIN Ciudades Ci ON C.ID_Ciudad = Ci.ID_Ciudad;
```

Ejercicio 2:

Productos y proveedores.

```
SELECT P.Nombre, F.Nombre AS Proveedor  
FROM Productos P  
JOIN Proveedores F ON P.ID_Proveedor = F.ID_Proveedor;
```

◊ Nivel FACIL – 2.4 – Oracle

Ejercicio 1:

Empleados y sus departamentos.

sql
CopiarEditar
SELECT E.Nombre, D.Nombre AS Departamento
FROM Empleados E
JOIN Departamentos D ON E.ID_Dpto = D.ID_Dpto;

Ejercicio 2:

Mostrar los libros y sus autores.

sql
CopiarEditar

```
SELECT L.Titulo, A.Nombre  
FROM Libros L  
JOIN Autores A ON L.ID_Autor = A.ID_Autor;
```

◊ Nivel INTERMEDIO – 2.4 – SQL Server

Ejercicio 1:

Obtener los productos, sus ventas y las fechas.

```
SELECT P.Nombre, V.Total, V.Fecha  
FROM Ventas V  
JOIN Productos P ON V.ID_Producto = P.ID_Producto;
```

Ejercicio 2:

Listar clientes que hayan hecho pedidos junto con el monto total.

```
SELECT C.Nombre, SUM(P.Monto) AS Total  
FROM Clientes C  
JOIN Pedidos P ON C.ID_Cliente = P.ID_Cliente  
GROUP BY C.Nombre;
```

◊ Nivel INTERMEDIO – 2.4 – MySQL

Ejercicio 1:

Listar cursos y cantidad de estudiantes inscritos.

```
SELECT C.NombreCurso, COUNT(*) AS Cantidad  
FROM Inscripciones I  
JOIN Cursos C ON I.ID_Curso = C.ID_Curso  
GROUP BY C.NombreCurso;
```

Ejercicio 2:

Clientes que hicieron compras y sus datos de contacto.

```
SELECT C.Nombre, C.Email, V.Fecha  
FROM Ventas V  
JOIN Clientes C ON V.ID_Cliente = C.ID_Cliente;
```

◊ Nivel INTERMEDIO – 2.4 – PostgreSQL

Ejercicio 1:

Cantidad de productos vendidos por vendedor.

```
SELECT V.Nombre, COUNT(*) AS ProductosVendidos  
FROM Ventas Ve
```

```
JOIN Vendedores V ON Ve.ID_Vendedor = V.ID_Vendedor  
GROUP BY V.Nombre;
```

Ejercicio 2:

Estudiantes y cantidad de cursos tomados.

```
SELECT E.Nombre, COUNT(*) AS CursosTomados  
FROM Estudiantes E  
JOIN Inscripciones I ON E.ID_Estudiante = I.ID_Estudiante  
GROUP BY E.Nombre;
```

◊ Nivel INTERMEDIO – 2.4 – Oracle

Ejercicio 1:

Total de horas trabajadas por empleado en diferentes proyectos.

```
SELECT E.Nombre, SUM(T.Horas) AS TotalHoras  
FROM Tareas T  
JOIN Empleados E ON T.ID_Emppleado = E.ID_Emppleado  
GROUP BY E.Nombre;
```

Ejercicio 2:

Productos vendidos con sus precios y fecha de venta.

```
SELECT P.Nombre, P.Precio, V.Fecha  
FROM Ventas V  
JOIN Productos P ON V.ID_Producto = P.ID_Producto;
```

◊ Nivel DIFÍCIL – 2.4 – SQL Server

Ejercicio 1:

Mostrar el total de ventas por producto, incluso si no se vendió.

```
SELECT P.Nombre, ISNULL(SUM(V.Total), 0) AS Total  
FROM Productos P  
LEFT JOIN Ventas V ON P.ID_Producto = V.ID_Producto  
GROUP BY P.Nombre;
```

Ejercicio 2:

Clientes que no han realizado ningún pedido.

```
SELECT C.Nombre  
FROM Clientes C  
LEFT JOIN Pedidos P ON C.ID_Cliente = P.ID_Cliente  
WHERE P.ID_Pedido IS NULL;
```

◊ Nivel DIFICIL – 2.4 – MySQL

Ejercicio 1:

Total recaudado por curso, aunque no tenga inscripciones.

```
SELECT C.NombreCurso, IFNULL(SUM(Pago), 0) AS Total  
FROM Cursos C  
LEFT JOIN Inscripciones I ON C.ID_Curso = I.ID_Curso  
GROUP BY C.NombreCurso;
```

Ejercicio 2:

Mostrar empleados que no participaron en ningún proyecto.

```
SELECT E.Nombre  
FROM Empleados E  
LEFT JOIN Proyectos_Empieados PE ON E.ID_Empieado = PE.ID_Empieado  
WHERE PE.ID_Proyecto IS NULL;
```

◊ Nivel DIFICIL – 2.4 – PostgreSQL

Ejercicio 1:

Cantidad de alumnos sin inscripciones activas.

```
SELECT A.Nombre  
FROM Alumnos A  
LEFT JOIN Inscripciones I ON A.ID_Alumno = I.ID_Alumno  
WHERE I.ID_Inscripcion IS NULL;
```

Ejercicio 2:

Cursos sin evaluaciones registradas.

```
SELECT C.NombreCurso  
FROM Cursos C  
LEFT JOIN Evaluaciones E ON C.ID_Curso = E.ID_Curso  
WHERE E.ID_Evaluacion IS NULL;
```

◊ Nivel DIFICIL – 2.4 – Oracle

Ejercicio 1:

Obtener empleados y su jefe directo (autojoin).

```
SELECT E.Nombre AS Empleado, J.Nombre AS Jefe  
FROM Empleados E  
LEFT JOIN Empleados J ON E.ID_Jefe = J.ID_Empleado;
```

Ejercicio 2:

Total de ventas por vendedor incluso si no vendió nada.

```
SELECT V.Nombre, NVL(SUM(Ve.Total), 0) AS TotalVentas  
FROM Vendedores V  
LEFT JOIN Ventas Ve ON V.ID_Vendedor = Ve.ID_Vendedor  
GROUP BY V.Nombre;
```

◊ Nivel FÁCIL – 2.5 – SQL Server

Obtener los clientes cuyo ID es mayor que el ID promedio de todos los clientes.

```
SELECT *  
FROM Clientes  
WHERE ID_Cliente >  
(SELECT AVG(ID_Cliente) FROM Clientes);
```

Listar los productos cuyo precio es mayor que el precio promedio.

```
SELECT *  
FROM Productos  
WHERE Precio >  
(SELECT AVG(Precio) FROM Productos);
```

◊ Nivel FÁCIL – 2.5 – MySQL

Encontrar empleados cuyo salario es mayor que el salario promedio.

```
SELECT *  
FROM Empleados  
WHERE Salario >  
(SELECT AVG(Salario) FROM Empleados);
```

Mostrar cursos con duración mayor que la duración promedio.

```
SELECT *  
FROM Cursos  
WHERE Duracion >  
(SELECT AVG(Duracion) FROM Cursos);
```

◊ Nivel FACIL – 2.5 – PostgreSQL

Listar libros con más páginas que el promedio de páginas.

```
SELECT *
FROM Libros
WHERE Paginas >
      (SELECT AVG(Paginas) FROM Libros);
```

Mostrar ciudades con más habitantes que la ciudad promedio.

```
SELECT *
FROM Ciudades
WHERE Habitantes >
      (SELECT AVG(Habitantes) FROM Ciudades);
```

◊ Nivel FACIL – 2.5 – Oracle

Mostrar empleados que ganan más que el promedio de su departamento.

```
SELECT *
FROM Empleados E
WHERE Salario >
      (SELECT AVG(Salario) FROM Empleados WHERE ID_Dpto = E.ID_Dpto);
```

Listar departamentos con más de 5 empleados.

```
SELECT *
FROM Departamentos
WHERE ID_Dpto IN
      (SELECT ID_Dpto FROM Empleados GROUP BY ID_Dpto HAVING COUNT(*) > 5);
```

◊ Nivel INTERMEDIO – 2.5 – SQL Server

Obtener productos cuya categoría tiene un promedio de precio mayor a 100.

```
SELECT * FROM Productos
WHERE CategoriaID IN (
    SELECT CategoriaID
    FROM Productos
    GROUP BY CategoriaID
    HAVING AVG(Precio) > 100
);
```

Listar empleados cuyo salario supera al salario promedio de su área.

```
SELECT * FROM Empleados E
WHERE Salario > (
    SELECT AVG(Salario)
    FROM Empleados
    WHERE Area = E.Area
);
```

◊ Nivel INTERMEDIO – 2.5 – MySQL

Encontrar clientes que han realizado más compras que el promedio.

```
SELECT * FROM Clientes
WHERE ID_Cliente IN (
    SELECT ID_Cliente
    FROM Compras
    GROUP BY ID_Cliente
    HAVING COUNT(*) > (
        SELECT AVG(ComprasTotales)
        FROM (
            SELECT COUNT(*) AS ComprasTotales
            FROM Compras
            GROUP BY ID_Cliente
        ) AS Subconsulta
    )
);
```

Mostrar productos cuya cantidad vendida es mayor al promedio del total vendido.

```
SELECT * FROM Productos
WHERE ID_Producto IN (
    SELECT ID_Producto
    FROM Ventas
    GROUP BY ID_Producto
    HAVING SUM(Cantidad) > (
        SELECT AVG(Suma)
        FROM (
            SELECT SUM(Cantidad) AS Suma
            FROM Ventas
            GROUP BY ID_Producto
        ) AS Promedios
    )
);
```

◊ Nivel INTERMEDIO – 2.5 – PostgreSQL

Obtener empleados con antigüedad mayor que el promedio por departamento.

```
SELECT * FROM Empleados E
WHERE Antiguedad > (
    SELECT AVG(Antiguedad)
    FROM Empleados
    WHERE Departamento = E.Departamento
);
```

Listar cursos con más alumnos que el promedio general.

```
SELECT * FROM Cursos
WHERE ID_Curso IN (
    SELECT ID_Curso
    FROM Inscripciones
    GROUP BY ID_Curso
    HAVING COUNT(*) > (
        SELECT AVG(Cantidad)
        FROM (
            SELECT COUNT(*) AS Cantidad
            FROM Inscripciones
            GROUP BY ID_Curso
        ) AS Sub
    )
);
```

◊ Nivel INTERMEDIO – 2.5 – Oracle

Listar empleados cuyo salario está por encima del salario máximo de otro departamento.

```
SELECT * FROM Empleados
WHERE Salario > (
    SELECT MAX(Salario)
    FROM Empleados
    WHERE ID_Depto = 10
);
```

Mostrar productos más vendidos que el producto con menor venta.

```
SELECT * FROM Productos
WHERE CantidadVendida > (
    SELECT MIN(CantidadVendida)
```

```
    FROM Productos  
);
```

◊ Nivel DIFICIL – 2.5 – SQL Server

Listar clientes que han realizado compras solo en el último mes.

```
SELECT * FROM Clientes  
WHERE ID_Cliente IN (  
    SELECT ID_Cliente  
    FROM Compras  
    WHERE MONTH(Fecha) = MONTH(GETDATE()) AND YEAR(Fecha) = YEAR(GETDATE())  
    EXCEPT  
    SELECT ID_Cliente  
    FROM Compras  
    WHERE MONTH(Fecha) <> MONTH(GETDATE()) OR YEAR(Fecha) <> YEAR(GETDATE())  
);
```

Mostrar productos que no han sido vendidos en ninguna tienda.

```
SELECT * FROM Productos  
WHERE ID_Producto NOT IN (  
    SELECT DISTINCT ID_Producto  
    FROM Ventas  
);
```

◊ Nivel DIFICIL – 2.5 – MySQL

Obtener empleados que trabajan en más de un proyecto simultáneamente.

```
SELECT * FROM Empleados  
WHERE ID_Emppleado IN (  
    SELECT ID_Emppleado  
    FROM ProyectoEmpleado  
    GROUP BY ID_Emppleado  
    HAVING COUNT(DISTINCT ID_Proyecto) > 1  
);
```

Listar productos con ventas mayores que la suma de ventas de productos de baja rotación.

```
SELECT * FROM Productos  
WHERE ID_Producto IN (  
    SELECT ID_Producto
```

```
FROM Ventas
GROUP BY ID_Producto
HAVING SUM(Cantidad) > (
    SELECT SUM(Cantidad)
    FROM Ventas
    WHERE ID_Producto IN (
        SELECT ID_Producto FROM Productos WHERE Categoría = 'Baja Rotación'
    )
)
);
```

◊ Nivel DIFICIL – 2.5 – PostgreSQL

Obtener clientes que compraron todos los productos disponibles.

```
SELECT * FROM Clientes C
WHERE NOT EXISTS (
    SELECT * FROM Productos P
    WHERE NOT EXISTS (
        SELECT * FROM Compras R
        WHERE R.ID_Cliente = C.ID_Cliente AND R.ID_Producto = P.ID_Producto
    )
);
);
```

Listar proveedores que han entregado productos a todos los depósitos.

```
SELECT * FROM Proveedores P
WHERE NOT EXISTS (
    SELECT * FROM Depositos D
    WHERE NOT EXISTS (
        SELECT * FROM Entregas E
        WHERE E.ID_Proveedor = P.ID_Proveedor AND E.ID_Depósito = D.ID_Depósito
    )
);
);
```

◊ Nivel DIFICIL – 2.5 – Oracle

Listar empleados que ganan más que el jefe de su departamento.

```
SELECT * FROM Empleados E
WHERE Salario > (
    SELECT Salario
    FROM Empleados
```

```
    WHERE ID_Emppleado = E.ID_Jefe  
);
```

Mostrar departamentos que tienen empleados en más de una sede.

```
SELECT * FROM Departamentos  
WHERE ID_Dpto IN (  
    SELECT ID_Dpto  
    FROM Empleados  
    GROUP BY ID_Dpto  
    HAVING COUNT(DISTINCT ID_Sede) > 1  
);
```

◊ Nivel FACIL – 2.6 – SQL Server

Crear una vista que muestre el nombre y correo electrónico de todos los clientes activos.

```
CREATE VIEW VistaClientesActivos AS  
SELECT Nombre, Email FROM Clientes WHERE Estado = 'Activo';
```

Crear una vista para mostrar productos cuyo stock sea mayor a 100 unidades.

```
CREATE VIEW VistaProductosDisponibles AS  
SELECT Nombre, Stock FROM Productos WHERE Stock > 100;
```

◊ Nivel FACIL – 2.6 – MySQL

Vista con nombre y salario de empleados del área de Ventas.

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

Vista de libros con más de 300 páginas.

```
CREATE VIEW VistaLibrosLargos AS  
SELECT Titulo, Paginas FROM Libros WHERE Paginas > 300;
```

◊ Nivel FACIL – 2.6 – PostgreSQL

Vista que muestre estudiantes aprobados (nota mayor a 6).

```
CREATE VIEW VistaAprobados AS  
SELECT Nombre, Nota FROM Estudiantes WHERE Nota >= 6;
```

Vista de ciudades con más de 50000 habitantes.

```
CREATE VIEW VistaCiudadesGrandes AS  
SELECT Nombre, Habitantes FROM Ciudades WHERE Habitantes > 50000;
```

◊ Nivel FACIL – 2.6 – Oracle

Vista con los empleados del sector IT.

```
CREATE VIEW VistaIT AS
```

```
SELECT Nombre, Puesto FROM Empleados WHERE Departamento = 'IT';
```

Vista de cuentas bancarias con saldo mayor a \$10,000.

```
CREATE VIEW VistaCuentasAltas AS
```

```
SELECT NumeroCuenta, Saldo FROM Cuentas WHERE Saldo > 10000;
```

◊ Nivel INTERMEDIO – 2.6 – SQL Server

Vista con nombre del cliente, total de pedidos realizados.

```
CREATE VIEW VistaResumenClientes AS
```

```
SELECT C.Nombre, COUNT(P.ID_Pedido) AS TotalPedidos
```

```
FROM Clientes C
```

```
JOIN Pedidos P ON C.ID_Cliente = P.ID_Cliente
```

```
GROUP BY C.Nombre;
```

Vista con nombre y total gastado por cliente (usando JOIN).

```
CREATE VIEW VistaGastosClientes AS
```

```
SELECT C.Nombre, SUM(D.Cantidad * D.PrecioUnitario) AS TotalGastado
```

```
FROM Clientes C
```

```
JOIN Pedidos P ON C.ID_Cliente = P.ID_Cliente
```

```
JOIN DetallePedido D ON P.ID_Pedido = D.ID_Pedido
```

```
GROUP BY C.Nombre;
```

◊ Nivel INTERMEDIO – 2.6 – MySQL

Vista que muestre la cantidad de cursos por cada profesor.

```
CREATE VIEW VistaCursosProfesor AS
```

```
SELECT ProfesorID, COUNT(*) AS CantCursos
```

```
FROM Cursos
```

```
GROUP BY ProfesorID;
```

Vista que muestre productos, su categoría y precio, ordenados por precio.

```
CREATE VIEW VistaProductosCategorias AS
```

```
SELECT P.Nombre, C.Nombre AS Categoria, P.Precio
```

```
FROM Productos P
```

```
JOIN Categorias C ON P.ID_Categoría = C.ID_Categoría
```

```
ORDER BY P.Precio DESC;
```

◊ Nivel INTERMEDIO – 2.6 – PostgreSQL

Vista con total de ventas por producto.

```
CREATE VIEW VistaVentasPorProducto AS
SELECT ProductID, SUM(Cantidad) AS TotalVendido
FROM DetalleVentas
GROUP BY ProductID;
```

Vista con promedio de calificaciones por materia.

```
CREATE VIEW VistaPromedios AS
SELECT Materia, AVG(Nota) AS Promedio FROM Notas GROUP BY Materia;
```

◊ Nivel INTERMEDIO – 2.6 – ORACLE

Vista con empleados y la cantidad de tareas asignadas.

```
CREATE VIEW VistaTareasEmpleado AS
SELECT E.Nombre, COUNT(T.ID_Tarea) AS TotalTareas
FROM Empleados E
LEFT JOIN Tareas T ON E.ID_Empresa = T.ID_Empresa
GROUP BY E.Nombre;
```

Vista con total recaudado por evento.

```
CREATE VIEW VistaEventosRecaudacion AS
SELECT E.NombreEvento, SUM(V.Precio) AS Total
FROM Eventos E
JOIN Ventas V ON E.ID_Evento = V.ID_Evento
GROUP BY E.NombreEvento;
```

◊ Nivel DIFÍCIL – 2.6 – SQL Server

Vista que muestra clientes con más de 3 pedidos y gasto total.

```
CREATE VIEW VistaClientesFrecuentes AS
SELECT C.Nombre, COUNT(P.ID_Pedido) AS CantPedidos,
SUM(D.Cantidad * D.PrecioUnitario) AS TotalGastado
FROM Clientes C
JOIN Pedidos P ON C.ID_Cliente = P.ID_Cliente
JOIN DetallePedido D ON P.ID_Pedido = D.ID_Pedido
GROUP BY C.Nombre
HAVING COUNT(P.ID_Pedido) > 3;
```

Vista que combina empleados, sus jefes y el total de horas trabajadas.

```
CREATE VIEW VistaHorasJefes AS
SELECT E.Nombre, J.Nombre AS Jefe, SUM(H.Horas) AS TotalHoras
FROM Empleados E
JOIN Jefes J ON E.ID_Jefe = J.ID_Jefe
JOIN Horas H ON E.ID_Empresa = H.ID_Empresa
GROUP BY E.Nombre, J.Nombre;
```

```
FROM Empleados E
JOIN Jefes J ON E.ID_Jefe = J.ID_Jefe
JOIN HorasTrabajadas H ON E.ID_Emppleado = H.ID_Emppleado
GROUP BY E.Nombre, J.Nombre;
```

◊ Nivel DIFICIL – 2.6 – MySQL

Vista que muestre los productos con ventas mayores al promedio de ventas de todos los productos.

```
CREATE VIEW VistaProductosTop AS
SELECT P.Nombre, SUM(D.Cantidad) AS TotalVendido
FROM Productos P
JOIN DetalleVentas D ON P.ID_Producto = D.ID_Producto
GROUP BY P.Nombre
HAVING SUM(D.Cantidad) > (
SELECT AVG(Total) FROM (
SELECT SUM(Cantidad) AS Total FROM DetalleVentas GROUP BY ID_Producto
) AS Sub);
```

Vista con clientes y total gastado en los últimos 6 meses.

```
CREATE VIEW VistaClientesUltimos6Meses AS
SELECT C.Nombre, SUM(D.Cantidad * D.PrecioUnitario) AS Gasto
FROM Clientes C
JOIN Pedidos P ON C.ID_Cliente = P.ID_Cliente
JOIN DetallePedido D ON P.ID_Pedido = D.ID_Pedido
WHERE P.Fecha > DATE_SUB(CURDATE(), INTERVAL 6 MONTH)
GROUP BY C.Nombre;
```

◊ Nivel DIFICIL – 2.6 – PostgreSQL

Vista con promedio de ventas mensual por producto.

```
CREATE VIEW VistaPromedioMensual AS
SELECT P.Nombre, EXTRACT(MONTH FROM V.Fecha) AS Mes,
AVG(D.Cantidad * D.PrecioUnitario) AS Promedio
FROM Productos P
JOIN DetalleVentas D ON P.ID_Producto = D.ID_Producto
JOIN Ventas V ON D.ID_Venta = V.ID_Venta
GROUP BY P.Nombre, EXTRACT(MONTH FROM V.Fecha);
```

Vista con los 5 clientes que más gastaron.

```
CREATE VIEW VistaTopClientes AS
SELECT C.Nombre, SUM(D.Cantidad * D.PrecioUnitario) AS GastoTotal
```

```
FROM Clientes C
JOIN Pedidos P ON C.ID_Cliente = P.ID_Cliente
JOIN DetallePedido D ON P.ID_Pedido = D.ID_Pedido
GROUP BY C.Nombre
ORDER BY GastoTotal DESC
LIMIT 5;
```

◊ Nivel DIFICIL – 2.6 – ORACLE

Vista que muestra empleados cuyo salario es mayor al máximo de su departamento.

```
CREATE VIEW VistaEmpleadosTop AS
SELECT E.Nombre, E.Salario, E.ID_Depto
FROM Empleados E
WHERE E.Salario = (
SELECT MAX(Salario)
FROM Empleados
WHERE ID_Depto = E.ID_Depto);
```

Vista de alumnos con mejor promedio por carrera.

```
CREATE VIEW VistaMejoresPromedios AS
SELECT A.Nombre, C.NombreCarrera, AVG(N.Nota) AS Promedio
FROM Alumnos A
JOIN Notas N ON A.ID_Alumno = N.ID_Alumno
JOIN Carreras C ON A.ID_Carrera = C.ID_Carrera
GROUP BY A.Nombre, C.NombreCarrera
HAVING AVG(N.Nota) = (
SELECT MAX(Prom) FROM (
SELECT AVG(Nota) AS Prom FROM Notas N2
JOIN Alumnos A2 ON N2.ID_Alumno = A2.ID_Alumno
WHERE A2.ID_Carrera = A.ID_Carrera
GROUP BY A2.ID_Alumno
));
```

◊ Nivel FACIL – 2.7 – SQL Server

Crear una tabla con claves primarias y NOT NULL

```
CREATE TABLE Clientes (
ID_Cliente INT PRIMARY KEY,
Nombre NVARCHAR(100) NOT NULL,
```

```
Email NVARCHAR(100)
);
```

Tabla con columna con valor por defecto (DEFAULT)

```
CREATE TABLE Productos (
ID_Producto INT PRIMARY KEY,
Nombre NVARCHAR(100),
Precio DECIMAL(10,2) DEFAULT 0.00
);
```

◊ Nivel FACIL – 2.7 – MySQL

Tabla con restricción UNIQUE y NOT NULL

```
CREATE TABLE Usuarios (
ID INT PRIMARY KEY,
Usuario VARCHAR(50) NOT NULL,
Email VARCHAR(100) UNIQUE
);
```

Tabla con DEFAULT y NOT NULL

```
CREATE TABLE Categorias (
ID_Categoría INT PRIMARY KEY,
Nombre VARCHAR(100) NOT NULL,
Activa BOOLEAN DEFAULT TRUE
);
```

◊ Nivel FACIL – 2.7 – PostgreSQL

Clave primaria y campo obligatorio

```
CREATE TABLE Cursos (
ID_Curso SERIAL PRIMARY KEY,
Nombre TEXT NOT NULL,
Duracion INT
);
```

Restricción CHECK para rango válido

```
CREATE TABLE Calificaciones (
ID SERIAL PRIMARY KEY,
Nota INT CHECK (Nota >= 0 AND Nota <= 10)
);
```

- ◊ Nivel FACIL – 2.7 – Oracle

Tabla con campo obligatorio y valor por defecto

```
CREATE TABLE Proveedores (
    ID_Proveedor NUMBER PRIMARY KEY,
    Nombre VARCHAR2(100) NOT NULL,
    Estado VARCHAR2(20) DEFAULT 'Activo'
);
```

Tabla con restricción UNIQUE y CHECK

```
CREATE TABLE Empleados (
    ID_Emppleado NUMBER PRIMARY KEY,
    DNI VARCHAR2(10) UNIQUE,
    Edad NUMBER CHECK (Edad >= 18)
);
```

- ◊ Nivel INTERMEDIO – 2.7 – SQL Server

Relación entre Clientes y Pedidos con FOREIGN KEY

```
CREATE TABLE Pedidos (
    ID_Pedido INT PRIMARY KEY,
    ID_Cliente INT,
    Fecha DATE,
    FOREIGN KEY (ID_Cliente) REFERENCES Clientes(ID_Cliente)
);
```

Restricción de valor predeterminado y CHECK

```
CREATE TABLE Pagos (
    ID_Pago INT PRIMARY KEY,
    Monto DECIMAL(10,2) CHECK (Monto > 0),
    MetodoPago VARCHAR(50) DEFAULT 'Efectivo'
);
```

- ◊ Nivel INTERMEDIO – 2.7 – MySQL

Relación con clave foránea

```
CREATE TABLE Ventas (
    ID_Venta INT PRIMARY KEY,
    ClienteID INT,
```

```
FOREIGN KEY (ClienteID) REFERENCES Usuarios(ID)
);
```

Restricción múltiple con CHECK

```
CREATE TABLE Productos (
ID INT PRIMARY KEY,
Precio DECIMAL(10,2),
Stock INT CHECK (Stock >= 0 AND Precio > 0)
);
```

◊ Nivel INTERMEDIO – 2.7 – PostgreSQL

Relacionar cursos con profesores

```
CREATE TABLE Profesores (
ID_Profesor SERIAL PRIMARY KEY,
Nombre TEXT
);
```

```
CREATE TABLE Cursos (
ID_Curso SERIAL PRIMARY KEY,
Nombre TEXT,
ProfesorID INT REFERENCES Profesores(ID_Profesor)
);
```

Restricción de datos válidos para campo sexo

```
CREATE TABLE Personas (
ID SERIAL PRIMARY KEY,
Sexo CHAR(1) CHECK (Sexo IN ('M', 'F'))
);
```

◊ Nivel INTERMEDIO – 2.7 – ORACLE

Relación entre empleados y departamentos

```
CREATE TABLE Departamentos (
ID_Depto NUMBER PRIMARY KEY,
Nombre VARCHAR2(100)
);
```

```
CREATE TABLE Empleados (
ID_Emppleado NUMBER PRIMARY KEY,
```

```
Nombre VARCHAR2(100),
ID_Depto NUMBER REFERENCES Departamentos(ID_Depto)
);
```

Tabla con múltiples restricciones

```
CREATE TABLE Vehiculos (
ID_Vehiculo NUMBER PRIMARY KEY,
Patente VARCHAR2(10) UNIQUE,
Kilometros NUMBER CHECK (Kilometros >= 0),
Activo CHAR(1) DEFAULT 'S'
);
```

◊ Nivel DIFICIL – 2.7 – SQL Server

Relación con tabla asociativa y múltiples claves foráneas

```
CREATE TABLE Estudiantes (
ID_Estudiante INT PRIMARY KEY,
Nombre NVARCHAR(100)
);
```

```
CREATE TABLE Cursos (
ID_Curso INT PRIMARY KEY,
Nombre NVARCHAR(100)
);
```

```
CREATE TABLE Inscripciones (
ID_Estudiante INT,
ID_Curso INT,
FechaInscripcion DATE,
PRIMARY KEY (ID_Estudiante, ID_Curso),
FOREIGN KEY (ID_Estudiante) REFERENCES Estudiantes(ID_Estudiante),
FOREIGN KEY (ID_Curso) REFERENCES Cursos(ID_Curso)
);
```

Restricción compleja: solo adultos pueden tener tarjeta

```
CREATE TABLE Tarjetas (
ID_Tarjeta INT PRIMARY KEY,
ID_Cliente INT UNIQUE,
EdadCliente INT CHECK (EdadCliente >= 18),
FOREIGN KEY (ID_Cliente) REFERENCES Clientes(ID_Cliente)
);
```

◊ Nivel DIFICIL – 2.7 – MySQL

Relación N:M con tabla intermedia

```
CREATE TABLE Autores (
    ID INT PRIMARY KEY,
    Nombre VARCHAR(100)
);
```

```
CREATE TABLE Libros (
    ID INT PRIMARY KEY,
    Titulo VARCHAR(100)
);
```

```
CREATE TABLE AutorLibro (
    ID_Autor INT,
    ID_Libro INT,
    PRIMARY KEY (ID_Autor, ID_Libro),
    FOREIGN KEY (ID_Autor) REFERENCES Autores(ID),
    FOREIGN KEY (ID_Libro) REFERENCES Libros(ID)
);
```

Restricción con CHECK en múltiples columnas

```
CREATE TABLE Vuelos (
    ID INT PRIMARY KEY,
    Origen VARCHAR(50),
    Destino VARCHAR(50),
    CHECK (Origen <> Destino)
);
```

◊ Nivel DIFICIL – 2.7 – PostgreSQL

Tabla con clave compuesta y restricciones

```
CREATE TABLE Reservas (
    ID_Cliente INT,
    ID_Habitacion INT,
    Fecha DATE,
    PRIMARY KEY (ID_Cliente, ID_Habitacion),
    FOREIGN KEY (ID_Cliente) REFERENCES Clientes(ID_Cliente),
    FOREIGN KEY (ID_Habitacion) REFERENCES Habitaciones(ID_Habitacion)
```

);

Restricción que obliga un mínimo de unidades vendidas

```
CREATE TABLE Ventas (
ID SERIAL PRIMARY KEY,
ProductoID INT REFERENCES Productos(ID),
Cantidad INT CHECK (Cantidad >= 1)
);
```

◊ Nivel DIFICIL – 2.7 – ORACLE

Relación compleja entre vehículos, conductores y viajes

```
CREATE TABLE Conductores (
ID NUMBER PRIMARY KEY,
Nombre VARCHAR2(100)
);
```

```
CREATE TABLE Vehiculos (
ID NUMBER PRIMARY KEY,
Modelo VARCHAR2(100)
);
```

```
CREATE TABLE Viajes (
ID_Conductor NUMBER,
ID_Vehiculo NUMBER,
Fecha DATE,
PRIMARY KEY (ID_Conductor, ID_Vehiculo, Fecha),
FOREIGN KEY (ID_Conductor) REFERENCES Conductores(ID),
FOREIGN KEY (ID_Vehiculo) REFERENCES Vehiculos(ID)
);
```

Restricción en formato de DNI

```
CREATE TABLE Ciudadanos (
ID NUMBER PRIMARY KEY,
DNI VARCHAR2(9) CHECK (REGEXP_LIKE(DNI, '^[0-9]{8}[A-Z]$'))
);
```

◊ Tablas – 2.8

Tablas:

```
CREATE TABLE Productos (
```

```
    ID INT PRIMARY KEY,
```

```
    Nombre NVARCHAR(50),
```

```
    Categoría NVARCHAR(50)
```

```
);
```

```
INSERT INTO Productos VALUES
```

```
(1, 'cámara digital', 'fotografía'),
```

```
(2, 'notebook gamer', 'tecnología'),
```

```
(3, 'micrófono condensador', 'audio');
```

◊ Nivel FACIL – 2.8 – SQL Server

Ejercicio 1 – Normalización de texto y extracción

```
SELECT UPPER(Nombre) AS NombreMayuscula,
```

```
    LEN(Categoría) AS LargoCategoría
```

```
FROM Productos;
```

Ejercicio 2 – Fecha y usuario actual

```
SELECT GETDATE() AS FechaActual,
```

```
    SUSER_NAME() AS Usuario;
```

◊ Nivel FACIL – 2.8 – MySQL

Ejercicio 1 – Extracción de subcadenas

```
SELECT SUBSTRING(Nombre, 1, 5) AS PrimerasLetras,
```

```
    LENGTH(Categoría) AS Largo
```

```
FROM Productos;
```

Ejercicio 2 – Hora y base actual

```
SELECT NOW() AS FechaActual,
```

```
    CURRENT_USER() AS Usuario,
```

```
    DATABASE() AS BaseActual;
```

◊ Nivel FÁCIL – 2.8 – PostgreSQL

Ejercicio 1 – Concatenación y formato

```
SELECT CONCAT(UPPER(Nombre), ' - ', Categoria) AS Descripcion  
FROM Productos;
```

Ejercicio 2 – Funciones de sistema

```
SELECT CURRENT_DATE AS FechaHoy,  
       CURRENT_USER AS Usuario,  
       VERSION() AS Version;
```

◊ Nivel FÁCIL – 2.8 – Oracle

Ejercicio 1 – Extraer parte del nombre

```
SELECT SUBSTR(Nombre, 1, 4) AS Corte,  
       LENGTH(Categoria) AS Largo  
FROM Productos;
```

Ejercicio 2 – Fecha y usuario

```
SELECT SYSDATE AS Fecha,  
       USER AS Usuario  
FROM dual;
```

◊ Nivel INTERMEDIO – 2.8 – SQL Server

Ejercicio 1 – Fechas relativas

```
SELECT DATEADD(DAY, 10, GETDATE()) AS FechaFutura,  
       DATENAME(MONTH, GETDATE()) AS MesActual;
```

Ejercicio 2 – Manipulación de texto y funciones anidadas

```
SELECT REPLACE(UPPER(Nombre), 'A', '@') AS NombreCambiado  
FROM Productos;
```

◊ Nivel INTERMEDIO – 2.8 – MySQL

Ejercicio 1 – Fechas y redondeo

```
SELECT ROUND(AVG(ID), 2) AS PromedioID,
```

```
DATE_ADD(NOW(), INTERVAL 5 DAY) AS FechaFutura  
FROM Productos;
```

Ejercicio 2 – Conversión de tipos

```
SELECT CAST('2025-08-06' AS DATE) AS FechaConvertida;
```

- ◊ Nivel INTERMEDIO – 2.8 – PostgreSQL

Ejercicio 1 – Fecha + Intervalos

```
SELECT NOW() + INTERVAL '7 days' AS EnUnaSemana;
```

Ejercicio 2 – Extraer año de una fecha textual

```
SELECT EXTRACT(YEAR FROM DATE '2022-10-15') AS Año;
```

- ◊ Nivel INTERMEDIO – 2.8 – ORACLE

Ejercicio 1 – Operaciones con fechas

```
SELECT SYSDATE + 3 AS FechaEnTresDias,  
       ADD_MONTHS(SYSDATE, 1) AS FechaProxMes  
  FROM dual;
```

Ejercicio 2 – Convertir y formatear fecha

```
SELECT TO_DATE('06/08/2025', 'DD/MM/YYYY') AS FechaFormateada FROM dual;
```

- ◊ Nivel DIFÍCIL – 2.8 – SQL Server

Ejercicio 1 – Cálculo con datos y presentación personalizada

```
SELECT ID,  
       Nombre,  
       Categoria,  
       UPPER(Nombre) + ' pertenece a ' + UPPER(Categoría) AS Descripcion  
  FROM Productos;
```

Ejercicio 2 – Auditoría avanzada

```
SELECT GETDATE() AS FechaConsulta,  
       HOST_NAME() AS Maquina,  
       SUSER_SNAME() AS Usuario;
```

◊ Nivel DIFICIL – 2.8 – MySQL Server

Ejercicio 1 – Cálculos y concatenación

```
SELECT CONCAT(UCASE(Nombre), '[',Categoria,',']') AS Etiqueta  
FROM Productos;
```

Ejercicio 2 – Fecha futura y mod

sql

CopiarEditar

```
SELECT NOW() + INTERVAL (MOD(ID, 3)) DAY AS FechaPersonalizada  
FROM Productos;
```

◊ Nivel DIFICIL – 2.8 – PostgreSQL

Ejercicio 1 – Texto dinámico y condiciones

```
SELECT CASE  
    WHEN LENGTH(Nombre) > 10 THEN 'Nombre largo'  
    ELSE 'Nombre corto'  
END AS Clasificacion,  
Nombre  
FROM Productos;
```

Ejercicio 2 – Extraer año + concatenación

```
SELECT EXTRACT(YEAR FROM NOW()) || '-' || Nombre AS NombreAnio  
FROM Productos;
```

◊ Nivel DIFICIL – 2.8 – ORACLE

Ejercicio 1 – Combinación con funciones numéricas

```
SELECT ID,  
       POWER(ID, 2) AS Cuadrado,  
       MOD(ID, 2) AS Paridad  
FROM Productos;
```

Ejercicio 2 – Función de auditoría

```
SELECT SYSDATE AS Fecha,  
       USER AS Usuario,  
       UID AS IDUsuario
```

FROM dual;