UNIVERSIDAD SIGLO 21

Desarrollo de Aplicaciones con Bases de Datos

Licenciatura en Informática
Trabajo Práctico 1

Prof. Titular Disciplinar: Silvia Laura Castelli
Prof. Titular Experto: Ricardo Ramón Daubrowsky
Alumno: Pablo Alejandro Hamann
Legajo: VINF010782
Año: 2025

Tabla de contenido

Introducción

Este documento corresponde al desarrollo de las consignas planteadas en el Trabajo Práctico 1. Complementan al mismo, el archivo SQL que contiene todas las sentencias necesarias para crear la base de datos, su estructura, poblarla con algunos datos modelo, y realizar las consultas solicitadas.

Repositorio en GitHub

Tanto este documento, como el *script* SQL con las todas las sentencias para el borrado, creación de la estructura de tablas, funciones, *trigers*, población de datos y consultas sobre la base de datos que parcialmente se van detallando de forma comentada en este documento, se pueden ver en el repositorio en GitHub creado para el cursado de esta materia. Allí, se mantienen actualizadas tanto las actividades prácticas como los TPs y cualquier otro tipo de actividad que implique desarrollo (de documentación, programación, etc.), que se dé durante el cursado de la materia. El repositorio se puede acceder mediante el siguiente enlace:

---

https://github.com/linkstat/dabd/tree/main

Archivos principales del proyecto ***dabd*** (este proyecto):

- Este documento en formato PDF:
  - https://github.com/linkstat/dabd/raw/refs/heads/main/docs/HAMANN-PABLO-ALEJANDRO-TP1.pdf
- Este documento en formato DOCX de Word:
  - https://github.com/linkstat/dabd/raw/refs/heads/main/docs/HAMANN-PABLO-ALEJANDRO-TP1.docx
- Archivo de script SQL:
  - https://raw.githubusercontent.com/linkstat/dabd/refs/heads/main/sql/HAMANN-PABLO-ALEJANDRO-TP1.sql

También es posible ver el historial de *commits* realizado a los archivos (y a toda la estructura del proyecto), ya que se trata de un repositorio público y se actualizando de forma regular, sobre todo cuando se aplican muchos cambios.

Tareas previas a la creación de la estructura de la base de datos

Borrado y creación desde cero de la base de datos

Para asegurar que la base de datos se pudiera crear desde cero, tanto en su estructura de datos como en la población de datos iniciales (y posteriores consultas sobre éstos), lo primero que realiza el script SQL es borrar la tabla, y recrearla desde cero. Para esto se define una variable con el nombre de la tabla (que es pedidos, tal cual se indicó en el caso problemático). Esto se hace para evitar errores de tipeo: simplemente se define en una variable una sola vez el nombre, y se utiliza la variable en todo momento. Esta lógica de utilización de variables que se definen una sola vez, está presente a lo largo del desarrollo de todo el script.

```
-- Definición del nombre de la BD vía una variable (para evitar errores de tipeo,
defino una sola vez, y utilizo luego la variable)
SET @dbname = 'pedidos';

-- Borrado de la BD (si existiera)
```

```
SET @sql = CONCAT('DROP DATABASE IF EXISTS ', @dbname);
PREPARE stmt FROM @sql;
EXECUTE stmt;
DEALLOCATE PREPARE stmt;

-- Creación (o recreación, según) de la base de datos
SET @sql = CONCAT('CREATE DATABASE ', @dbname, ' CHARACTER SET utf8mb4 COLLATE
utf8mb4_unicode_ci');
PREPARE stmt FROM @sql;
EXECUTE stmt;
DEALLOCATE PREPARE stmt;

-- Ahora vamos a usar la BD creada
SET @sql = CONCAT('USE ', @dbname);
PREPARE stmt FROM @sql;
EXECUTE stmt;
DEALLOCATE PREPARE stmt;
```

Definición de funciones personalizadas

Dado que se decidió utilizar UUID y almacenarlos en formato binario, contar con una función que facilite la inserción y consulta de estos datos, nos facilitará enormemente la vida.
Las funciones propuestas[1], usan la representación estándar de UUID y no la propuesta de *MySQL 8.0+* (`UUID_TO_BIN(uuid, swap_flag=1)`). Es decir, los bytes no se reordenan.
Gracias a esto, se puede utilizar la biblioteca `java.util.UUID` para leer y manipular los UUID de la base de datos sin problemas, ya que son totalmente compatibles.

Función para convertir (y almacenar) un UUID en un binario de 16 bytes

```
-- Función para convertir (y almacenar) un UUID en un binario de 16 bytes
DELIMITER $$
CREATE FUNCTION `UUID_TO_BIN`(uuid CHAR(36))
RETURNS BINARY(16)
DETERMINISTIC
BEGIN
  RETURN UNHEX(CONCAT(
    SUBSTRING(uuid, 1, 8),      -- aaaaaaaa
    SUBSTRING(uuid, 10, 4),     -- bbbb
    SUBSTRING(uuid, 15, 4),     -- cccc
    SUBSTRING(uuid, 20, 4),     -- dddd
    SUBSTRING(uuid, 25, 12)     -- eeeeeeeeeeee
  ));
END$$
DELIMITER ;
```

[1] Rediseño en base a las propuestas en: https://mariadb.com/kb/en/uuid-data-type/

Función para recuperar y convertir de nuevo a UUID

```
-- Función para recuperar y convertir de nuevo a UUID
DELIMITER $$
CREATE FUNCTION `BIN_TO_UUID`(b BINARY(16))
RETURNS CHAR(36) CHARSET ascii
DETERMINISTIC
BEGIN
   DECLARE hexStr CHAR(32);
   SET hexStr = HEX(b);
   RETURN LOWER(CONCAT(
     SUBSTR(hexStr, 1, 8), '-',      -- aaaaaaaa
     SUBSTR(hexStr, 9, 4), '-',      -- bbbb
     SUBSTR(hexStr, 13, 4), '-',     -- cccc
     SUBSTR(hexStr, 17, 4), '-',     -- dddd
     SUBSTR(hexStr, 21, 12)          -- eeeeeeeeeeee
  ));
END$$
DELIMITER ;
```

Sentencias de creación de la estructura de la base de datos

Primero, se deshabilitan las restricciones de clave foránea (más que nada, para evitar potenciales errores que pudieran darse durante la creación e tablas), se crean las tablas y sus relaciones, y luego se habilita nuevamente las restricciones de clave foránea.

Creación de Tablas

Las tablas se crearon siguiendo la descripción dada en la presentación de la situación problemática. Durante el proceso de creación de las tablas, se realizó un muy pequeño cambio relacionado a la elección del tipo de dato para las PK de cada tabla, decidiendo utilizar UUID como claves primarias y almacenarlos como datos binarios.

Previamente a la creación de la estructura de tablas (que se detallan en los siguientes títulos), deshabilitamos las restricciones de clave foránea, y al finalizar, volvemos a habilitar las restricciones.

```
-- Deshabilitar las Restricciones de Claves Foráneas
SET FOREIGN_KEY_CHECKS = 0;

/*
 * Creación de toda la estructura de tablas y sus relaciones
 */

-- Rehabilitar las Restricciones de Claves Foráneas
SET FOREIGN_KEY_CHECKS = 1;
```

Tabla Clientes

Aquí se agregó el campo DNI (que no estaba incluido en el diseño dado en la situación problemática), ya que el mismo es solicitado en alguna de las consultas más adelante en el desarrollo de este TP.

```
CREATE TABLE Clientes (
    idcliente BINARY(16) NOT NULL PRIMARY KEY,
    DNI VARCHAR(20) NOT NULL,
    Apellido VARCHAR(100) NOT NULL,
    Nombres VARCHAR(100) NOT NULL,
    Direccion VARCHAR(255) NOT NULL,
    mail VARCHAR(100) NOT NULL
);
```

Tabla Proveedores

```
CREATE TABLE Proveedores (
    idproveedor BINARY(16) NOT NULL PRIMARY KEY,
    NombreProveedor VARCHAR(100) NOT NULL,
    Direccion VARCHAR(255) NOT NULL,
    email VARCHAR(100) NOT NULL
);
```

Tabla Vendedor

```
CREATE TABLE Vendedor (
    idvendedor BINARY(16) NOT NULL PRIMARY KEY,
    DNI VARCHAR(20) NOT NULL,
    Apellido VARCHAR(100) NOT NULL,
    Nombres VARCHAR(100) NOT NULL,
    email VARCHAR(100) NOT NULL,
    comision DECIMAL(5,2) NOT NULL
);
```

Tabla Productos

```
CREATE TABLE Productos (
    idproducto BINARY(16) NOT NULL PRIMARY KEY,
    Descripcion VARCHAR(255) NOT NULL,
    PrecioUnitario DECIMAL(10,2) NOT NULL,
    Stock INT NOT NULL,
    StockMax INT NOT NULL,
    StockMin INT NOT NULL,
    idproveedor BINARY(16) NOT NULL,
    origen ENUM('nacional', 'importado') NOT NULL,
    CONSTRAINT fk_producto_proveedor FOREIGN KEY (idproveedor) REFERENCES
Proveedores(idproveedor)
);
```

Tabla Pedidos

Aquí me tomé la libertad de modificar levemente la propuesta dada: por un lado, tengo que conservar el concepto de número de pedido (para la presentación de datos, por ejemplo), y por otra parte, quiero mantener consistencia interna al usar UUID almacenado como binario para las PK de todas las tablas de la base de datos.

```
CREATE TABLE Pedidos (
    idpedido BINARY(16) NOT NULL PRIMARY KEY,
    NumeroPedido INT NOT NULL AUTO_INCREMENT,
    idcliente BINARY(16) NOT NULL,
    idvendedor BINARY(16) NOT NULL,
    fecha DATE NOT NULL,
    Estado ENUM('pendiente', 'confirmado', 'anulado') NOT NULL DEFAULT 'pendiente',
    CONSTRAINT fk_pedido_cliente FOREIGN KEY (idcliente) REFERENCES Clientes(idcliente)
        ON DELETE RESTRICT,
    CONSTRAINT fk_pedido_vendedor FOREIGN KEY (idvendedor) REFERENCES Vendedor(idvendedor),
    UNIQUE KEY (NumeroPedido)
);
```

Tabla DetallePedidos

Al igual que con la tabla anterior, de Pedidos, nuevamente y siguiendo la misma lógica de conservar los PK como UUID almacenados como binarios, realizo una 'modificación sutil' para mantener coherencia.

```
CREATE TABLE DetallePedidos (
    idDetallePedido BINARY(16) NOT NULL PRIMARY KEY,
    NumeroPedido INT NOT NULL,
    renglon INT NOT NULL,
    idproducto BINARY(16) NOT NULL,
    cantidad INT NOT NULL,
    PrecioUnitario DECIMAL(10,2) NOT NULL,
    Total DECIMAL(10,2) AS (cantidad * PrecioUnitario) VIRTUAL,
    CONSTRAINT uniq_detalle UNIQUE (NumeroPedido, renglon),
    CONSTRAINT fk_detalle_numpedido FOREIGN KEY (NumeroPedido) REFERENCES Pedidos(NumeroPedido)
        ON DELETE CASCADE,
    CONSTRAINT fk_detalle_producto FOREIGN KEY (idproducto) REFERENCES Productos(idproducto)
);

/* Esta tabla no se explicita en la actividad propuesta, pero sin embargo, la penúltima regla
dice:
 * Todo pedido anulado debe ser auditado, grabando en la tabla de log, la información
 * del pedido anulado, indicando la fecha de anulación.
 */
CREATE TABLE LogAnulaciones (
    idLogAnulaciones BINARY(16) NOT NULL PRIMARY KEY,
```

```
    idpedido BINARY(16) NOT NULL,
    FechaAnulacion DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    Observaciones TEXT,
    CONSTRAINT fk_log_idpedido FOREIGN KEY (idpedido) REFERENCES Pedidos(idpedido)
);
```

Tabla LogAnulaciones

Esta tabla no se explicita en la actividad propuesta, pero sin embargo, la penúltima regla dice: *Todo pedido anulado debe ser auditado, grabando en la tabla de log, la información del pedido anulado, indicando la fecha de anulación.*

```
CREATE TABLE LogAnulaciones (
    idLogAnulaciones BINARY(16) NOT NULL PRIMARY KEY,
    idpedido BINARY(16) NOT NULL,
    FechaAnulacion DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    Observaciones TEXT,
    CONSTRAINT fk_log_idpedido FOREIGN KEY (idpedido) REFERENCES Pedidos(idpedido)
);
```

Creación de *Triggers*

Luego, para asegurar el cumplimiento de las reglas de negocio indicadas en la situación problemática, creamos una serie de *triggers*, que se ejecutarán sobre una tabla específica, durante una acción específica.

Trigger trg_before_insert_detalle

Este *trigger* actúa sobre la tabla `DetallePedidos` en `BEFORE INSERT` para asegurar que:

- se consulte el stock disponible y el precio unitario actual del producto (según su `idproducto`).
- se produzca un error si el stock es insuficiente para la cantidad solicitada.
- se asigne el precio unitario del producto en el campo correspondiente del detalle.

Cuando se dispara, indica producto y stock cuando genera el error.

```
DELIMITER $$
CREATE TRIGGER trg_before_insert_detalle
BEFORE INSERT ON DetallePedidos
FOR EACH ROW
BEGIN
    DECLARE v_stock INT;
    DECLARE v_precio DECIMAL(10,2);
    DECLARE v_desc VARCHAR(255);
    DECLARE v_msg VARCHAR(512);

    -- Consultamos el stock, precio y descripción del producto a insertar
    SELECT Stock, PrecioUnitario, Descripcion
      INTO v_stock, v_precio, v_desc
      FROM Productos
```

```
     WHERE idproducto = NEW.idproducto;

    -- Verificamos que haya stock suficiente; sino generamos un error informativo
    IF v_stock < NEW.cantidad THEN
        SET v_msg = CONCAT('Stock insuficiente para el producto ', v_desc,
                           '. Stock disponible: ', v_stock,
                           '. Cantidad requerida: ', NEW.cantidad);
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = v_msg;
    END IF;

    -- Asignamos automáticamente el precio unitario del producto al detalle del pedido
    SET NEW.PrecioUnitario = v_precio;
END;
$$
DELIMITER ;
```

Trigger trg_after_update_confirmado

Otra imposición de la consigna a resolver consiste en actualizar el stock del producto al confirmar el pedido. Este *trigger* actúa sobre la tabla `Pedidos` y se dispara después de una actualización:

- cuando el estado de un pedido cambia a 'confirmado' (si es que previamente no lo estaba), se actualiza el stock de cada producto restando la cantidad pedida.

```
DELIMITER $$
CREATE TRIGGER trg_after_update_confirmado
AFTER UPDATE ON Pedidos
FOR EACH ROW
BEGIN
    IF NEW.Estado = 'confirmado' AND OLD.Estado <> 'confirmado' THEN
        UPDATE Productos p
        JOIN DetallePedidos d ON p.idproducto = d.idproducto
        SET p.Stock = p.Stock - d.cantidad
        WHERE d.NumeroPedido = NEW.NumeroPedido;
    END IF;
END;
$$
DELIMITER ;
```

Trigger trg_after_insert_detalle_stock

Este *trigger* es prácticamente idéntico al anterior, solo que se ejecuta durante la inserción, y sobre la tabla `DetallePedidos` ¿Por qué? Porque el *trigger* anterior no sirve cuando se realiza la inserción directamente como confirmado.

Si la lógica de negocios es que un pedido ingresa como pendiente si o si, y luego debiera ser actualizado, este *trigger* no tendría sentido. Pero como estamos realizando inserciones con pedidos que pueden tener estado 'confirmado' al momento del `INSERT`, entonces este *trigger* es fundamental para actualizar el stock. Además, cuando hacemos este tipo de inserciones (como en

este TP), en ese momento aún no existen los detalles del pedido, entonces la actualización no se realiza (por eso la hacemos sobre `DetallePedidos`).

```
DELIMITER $$
CREATE TRIGGER trg_after_insert_detalle_stock
AFTER INSERT ON DetallePedidos
FOR EACH ROW
BEGIN
    DECLARE v_estado ENUM('pendiente','confirmado','anulado');

    -- Obtenemos el estado del pedido correspondiente al detalle
    SELECT Estado
      INTO v_estado
      FROM Pedidos
     WHERE NumeroPedido = NEW.NumeroPedido;

    IF v_estado = 'confirmado' THEN
        UPDATE Productos
        SET Stock = Stock - NEW.cantidad
        WHERE idproducto = NEW.idproducto;
    END IF;
END$$
DELIMITER ;
```

Trigger trg_after_update_anulado

Otras dos reglas de negocio indicada en la consigna, indica que:

- *Todo pedido anulado debe ser auditado, grabando en la tabla de log, la información del pedido anulado, indicando la fecha de anulación.*
- *El sistema debe recomponer el stock de cada pedido confirmado que es anulado.*

Entonces, cuando se anula un pedido (cambiando el estado a 'anulado'), este *trigger* sobre la tabla Pedidos, en su acción `AFTER UPDATE`, realizará lo siguiente:

- Registrar en `LogAnulaciones` la información del pedido anulado (incluida la fecha de anulación).
- Reponer el stock de los productos involucrados (sumando las cantidades que se restaron previamente).

```
DELIMITER $$
CREATE TRIGGER trg_after_update_anulado
AFTER UPDATE ON Pedidos
FOR EACH ROW
BEGIN
    IF NEW.Estado = 'anulado' AND OLD.Estado = 'confirmado' THEN
        -- Recomponer stock: sumar cantidades de cada detalle del pedido anulado
        UPDATE Productos p
        JOIN DetallePedidos d ON p.idproducto = d.idproducto
        SET p.Stock = p.Stock + d.cantidad
        WHERE d.NumeroPedido = NEW.NumeroPedido;
```

```
        -- Registrar en LogAnulaciones
        INSERT INTO LogAnulaciones (idLogAnulaciones, idpedido, FechaAnulacion, Observaciones)
        VALUES (UUID_TO_BIN(UUID()), NEW.idpedido, NOW(),
                CONCAT('Pedido ', NEW.NumeroPedido, ' anulado.'));
    END IF;
END;
$$
DELIMITER ;
```

Trigger trg_before_insert_productos

Este *trigger*, verifica que cuando se inserte un nuevo producto, el valor de Stock se encuentre el máximo y mínimo posible. Atento a la regla de negocio que indica:

Al ingresar un nuevo producto, se debe controlar que el stock se encuentre entre los límites de stock mínimo y máximo

```
DELIMITER $$
CREATE TRIGGER trg_before_insert_productos
BEFORE INSERT ON Productos
FOR EACH ROW
BEGIN
    DECLARE v_msg VARCHAR(512);

    IF NEW.Stock < NEW.StockMin OR NEW.Stock > NEW.StockMax THEN
        SET v_msg = CONCAT('El stock (', NEW.Stock,
                           ') debe estar entre ', NEW.StockMin,
                           ' y ', NEW.StockMax, '.');
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = v_msg;
    END IF;
END;
$$
DELIMITER ;
```

Conjunto de sentencias SQL para poblar la base de datos

Dado que los PK son UUID en formato binario, la técnica utilizada para realizar la población de los datos es generar UUIDs aleatoriamente (utilizando la función de MariaDB `UUID()` luego pasándolo a binario con la función previamente definida `UUID_TO_BIN()` y luego almacenando en la variable), de esta forma, las sentencias de población resultan más limpias, de mejor legibilidad.

Ingresar 5 clientes

```
-- Ingresar 5 clientes.
SET @uuid_cliente1 = UUID_TO_BIN(UUID());
SET @uuid_cliente2 = UUID_TO_BIN(UUID());
SET @uuid_cliente3 = UUID_TO_BIN(UUID());
```

```
SET @uuid_cliente4 = UUID_TO_BIN(UUID());
SET @uuid_cliente5 = UUID_TO_BIN(UUID());

INSERT INTO Clientes (idcliente, DNI, Apellido, Nombres, Direccion, mail)
VALUES
    (@uuid_cliente1, '18465781', 'Rojas Valdivia', 'Lucy Amanda', 'Av. Sabatini 3288',
'lucyamanda23@latinmail.com'),
    (@uuid_cliente2, '39512723', 'Alcaide', 'Santiago Agustín', 'Yrigoyen 733 5 C, La Plata,
Buenos Aires', 'santialcaide@mineral.ru'),
    (@uuid_cliente3, '22101645', 'Roqué', 'Juan Manuel', 'Avellaneda 935, La Banda, Santiago del
Estero', 'jmroque@yustech.com.ar'),
    (@uuid_cliente4, '42013728', 'Pérez', 'Carlos Enrique', 'Bedoya 724, Córdoba, Córdoba',
'carlitosperez@gmail.com'),
    (@uuid_cliente5, '12309421', 'Sánchez', 'Omar Wenceslao', 'Rivadavia, 724 3 C, Rosario, Santa
Fe', 'wen733@mail.ru');
```

Ingresar 3 proveedores

```
-- Ingresar 3 proveedores.
SET @uuid_proveedor1 = UUID_TO_BIN(UUID());
SET @uuid_proveedor2 = UUID_TO_BIN(UUID());
SET @uuid_proveedor3 = UUID_TO_BIN(UUID());

INSERT INTO Proveedores (idproveedor, NombreProveedor, Direccion, email)
VALUES
    (@uuid_proveedor1, 'Marolio', 'Corrientes 2350, Gral. Rodríguez, Buenos Aires',
'info@marolio.com.ar'),
    (@uuid_proveedor2, 'Arcor', 'Av. Chacabuco 1160, Córdoba, Córdoba', 'arcor@arcor.com'),
    (@uuid_proveedor3, 'Dos Hermanos', 'Av. Pres. Juan Domingo Perón y Scalabrini Ortiz,
Concordia, Entre Ríos','info@doshermanos.com.ar');
```

Ingresar 3 vendedores

```
-- Ingresar 3 vendedores.
SET @uuid_vendedor1 = UUID_TO_BIN(UUID());
SET @uuid_vendedor2 = UUID_TO_BIN(UUID());
SET @uuid_vendedor3 = UUID_TO_BIN(UUID());

INSERT INTO Vendedor (idvendedor, DNI, Apellido, Nombres, email, comision)
VALUES
    (@uuid_vendedor1, '36113214', 'Garay', 'Mauricio Elio', 'mgaray@msn.com', 10.15),
    (@uuid_vendedor2, '28101438', 'Cabral Perez', 'Matías', 'mcp@outlook.com', 23.2),
    (@uuid_vendedor3, '24741573', 'Castellanos', 'Matías','mcastellanos@gmail.com', 14.6);
```

Ingresar al menos 10 productos (distribuidos entre los 3 proveedores creados)

```
-- Ingresar al menos 10 productos (distribuidos entre los 3 proveedores creados).
SET @uuid_prod01 = UUID_TO_BIN(UUID());
SET @uuid_prod02 = UUID_TO_BIN(UUID());
SET @uuid_prod03 = UUID_TO_BIN(UUID());
SET @uuid_prod04 = UUID_TO_BIN(UUID());
SET @uuid_prod05 = UUID_TO_BIN(UUID());
SET @uuid_prod06 = UUID_TO_BIN(UUID());
SET @uuid_prod07 = UUID_TO_BIN(UUID());
SET @uuid_prod08 = UUID_TO_BIN(UUID());
SET @uuid_prod09 = UUID_TO_BIN(UUID());
SET @uuid_prod10 = UUID_TO_BIN(UUID());

INSERT INTO Productos (idproducto, Descripcion, PrecioUnitario, Stock, StockMax, StockMin,
idproveedor, origen)
VALUES
    (@uuid_prod01, 'Arroz Parboil 1kg Dos Hnos Libre Gluten Sin Tacc', 20865.0, 1518, 5000, 500,
@uuid_proveedor3, 'nacional'),
    (@uuid_prod02, 'Huevo de pascuas Arcor Milk unicornio chocolate 140g', 18999.0, 12497, 15000,
0, @uuid_proveedor2, 'nacional'),
    (@uuid_prod03, 'Yerba Mate Marolio Con Menta - Bolsa 500g', 1487.5, 1213, 12000, 1050,
@uuid_proveedor1, 'nacional'),
    (@uuid_prod04, 'Turron Arcor 25 Gramos Display De 50 Unidades', 11999.4, 870, 1942, 200,
@uuid_proveedor2, 'nacional'),
    (@uuid_prod05, 'Arroz Yamani 500g Dos Hermanos Integral Sin Tacc Libre Gluten', 6017.0, 1803,
7500, 780, @uuid_proveedor3, 'importado'),
    (@uuid_prod06, 'Picadillo Marolio 90g', 1648.98, 680, 3800, 230, @uuid_proveedor1,
'nacional'),
    (@uuid_prod07, 'Mermelada Marolio Damasco Frasco 454 Gr', 2240.0, 213, 1300, 25,
@uuid_proveedor1, 'nacional'),
    (@uuid_prod08, 'Mermelada Light De Ciruela Arcor X 390 Grs', 2559.0, 329, 1150, 20,
@uuid_proveedor2, 'importado'),
    (@uuid_prod09, 'Bocadito Holanda Arcor X 24 Unidades', 9799.0, 871, 900, 50,
@uuid_proveedor2, 'nacional'),
    (@uuid_prod10, 'Palmito Rodaja 800 Gramos Marolio', 7900.0, 852, 2500, 500, @uuid_proveedor1,
'importado');
```

Ingresar 10 pedidos en total con diferente cantidad de renglones (entre 1 y 3 renglones)

```
-- Ingresar 10 pedidos en total con diferente cantidad de renglones (se sugiere crear pedidos con
1, 2 o 3 renglones máximo).
-- Generamos y almacenamos los UUID para los 10 pedidos solicitados
```

```
SET @uuid_pedido01 = UUID_TO_BIN(UUID());
SET @uuid_pedido02 = UUID_TO_BIN(UUID());
SET @uuid_pedido03 = UUID_TO_BIN(UUID());
SET @uuid_pedido04 = UUID_TO_BIN(UUID());
SET @uuid_pedido05 = UUID_TO_BIN(UUID());
SET @uuid_pedido06 = UUID_TO_BIN(UUID());
SET @uuid_pedido07 = UUID_TO_BIN(UUID());
SET @uuid_pedido08 = UUID_TO_BIN(UUID());
SET @uuid_pedido09 = UUID_TO_BIN(UUID());
SET @uuid_pedido10 = UUID_TO_BIN(UUID());

-- Genero pedidos (al hacer un INSERT MULTIROW, pierdo la posibilidad de usar LAST_INSERT_ID();
para la variable @numPedido, pero sigue siendo más legible y me resulta cómodo en gral)
INSERT INTO Pedidos (idpedido, idcliente, idvendedor, fecha, Estado)
VALUES
    (@uuid_pedido01, @uuid_cliente1, @uuid_vendedor1, '2025-02-23', 'confirmado'),
    (@uuid_pedido02, @uuid_cliente5, @uuid_vendedor2, '2025-03-14', 'confirmado'),
    (@uuid_pedido03, @uuid_cliente1, @uuid_vendedor1, '2025-04-04', 'pendiente'),
    (@uuid_pedido04, @uuid_cliente4, @uuid_vendedor2, '2025-01-28', 'confirmado'),
    (@uuid_pedido05, @uuid_cliente2, @uuid_vendedor3, '2025-04-11', 'confirmado'),
    (@uuid_pedido06, @uuid_cliente2, @uuid_vendedor3, '2025-02-18', 'pendiente'),
    (@uuid_pedido07, @uuid_cliente1, @uuid_vendedor3, '2025-01-08', 'confirmado'),
    (@uuid_pedido08, @uuid_cliente3, @uuid_vendedor2, '2025-03-05', 'confirmado'),
    (@uuid_pedido09, @uuid_cliente4, @uuid_vendedor2, '2025-04-10', 'pendiente'),
    (@uuid_pedido10, @uuid_cliente3, @uuid_vendedor2, '2025-03-21', 'confirmado');
```

Pedido 01 de 10 (3 renglones)

```
-- Pedido 01 de 10 (3 renglones)
-- Nota, los números de pedido son autoincrementales (no se introducen manualmente),
-- asi que recupero el valor que necesito en cada caso, realizando una consulta (tengo/conozco el
@uuid_pedidoNN)
SELECT NumeroPedido INTO @numPedido FROM Pedidos WHERE idpedido = @uuid_pedido01;
-- Genero en cada caso los UUID para el detalle de pedido (y no antes todos juntos para no
perderme)
SET @uuid_DP01r1 = UUID_TO_BIN(UUID()); -- Pedido 1 Renglón 1
SET @uuid_DP01r2 = UUID_TO_BIN(UUID()); -- Pedido 1 Renglón 2
SET @uuid_DP01r3 = UUID_TO_BIN(UUID()); -- Pedido 1 Renglón 3
INSERT INTO DetallePedidos (idDetallePedido, NumeroPedido, renglon, idproducto, cantidad)
VALUES
    (@uuid_DP01r1, @numPedido, 1, @uuid_prod01, 58),
    (@uuid_DP01r2, @numPedido, 2, @uuid_prod02, 32),
    (@uuid_DP01r3, @numPedido, 3, @uuid_prod03, 211);
```

Pedido 02 de 10 (1 renglón)

```
-- Pedido 02 de 10 (1 renglón)
SELECT NumeroPedido INTO @numPedido FROM Pedidos WHERE idpedido = @uuid_pedido02;
SET @uuid_DP02r1 = UUID_TO_BIN(UUID());
INSERT INTO DetallePedidos (idDetallePedido, NumeroPedido, renglon, idproducto, cantidad)
VALUES (@uuid_DP02r1, @numPedido, 1, @uuid_prod05, 36);
```

Pedido 03 de 10 (2 renglones)

```
-- Pedido 03 de 10 (2 renglones)
SELECT NumeroPedido INTO @numPedido FROM Pedidos WHERE idpedido = @uuid_pedido03;
SET @uuid_DP03r1 = UUID_TO_BIN(UUID());
SET @uuid_DP03r2 = UUID_TO_BIN(UUID());
INSERT INTO DetallePedidos (idDetallePedido, NumeroPedido, renglon, idproducto, cantidad)
VALUES
    (@uuid_DP03r1, @numPedido, 1, @uuid_prod01, 9),
    (@uuid_DP03r2, @numPedido, 2, @uuid_prod04, 12);
```

Pedido 04 de 10 (3 renglones)

```
-- Pedido 04 de 10 (3 renglones)
SELECT NumeroPedido INTO @numPedido FROM Pedidos WHERE idpedido = @uuid_pedido04;
SET @uuid_DP04r1 = UUID_TO_BIN(UUID());
SET @uuid_DP04r2 = UUID_TO_BIN(UUID());
SET @uuid_DP04r3 = UUID_TO_BIN(UUID());

INSERT INTO DetallePedidos (idDetallePedido, NumeroPedido, renglon, idproducto, cantidad)
VALUES
    (@uuid_DP04r1, @numPedido, 1, @uuid_prod09, 15),
    (@uuid_DP04r2, @numPedido, 2, @uuid_prod06, 22),
    (@uuid_DP04r3, @numPedido, 3, @uuid_prod08, 10);
```

Pedido 05 de 10 (1 renglón)

```
-- Pedido 05 de 10 (1 renglón)
SELECT NumeroPedido INTO @numPedido FROM Pedidos WHERE idpedido = @uuid_pedido05;
SET @uuid_DP05r1 = UUID_TO_BIN(UUID());

INSERT INTO DetallePedidos (idDetallePedido, NumeroPedido, renglon, idproducto, cantidad)
VALUES (@uuid_DP05r1, @numPedido, 1, @uuid_prod10, 14);
```

Pedido 06 de 10 (2 renglones)

```
-- Pedido 06 de 10 (2 renglones)
SELECT NumeroPedido INTO @numPedido FROM Pedidos WHERE idpedido = @uuid_pedido06;
SET @uuid_DP06r1 = UUID_TO_BIN(UUID());
```

```
SET @uuid_DP06r2 = UUID_TO_BIN(UUID());

INSERT INTO DetallePedidos (idDetallePedido, NumeroPedido, renglon, idproducto, cantidad)
VALUES
    (@uuid_DP06r1, @numPedido, 1, @uuid_prod04, 75),
    (@uuid_DP06r2, @numPedido, 2, @uuid_prod08, 23);
```

Pedido 07 de 10 (3 renglones)

```
-- Pedido 07 de 10 (3 renglones)
SELECT NumeroPedido INTO @numPedido FROM Pedidos WHERE idpedido = @uuid_pedido07;
SET @uuid_DP07r1 = UUID_TO_BIN(UUID());
SET @uuid_DP07r2 = UUID_TO_BIN(UUID());
SET @uuid_DP07r3 = UUID_TO_BIN(UUID());

INSERT INTO DetallePedidos (idDetallePedido, NumeroPedido, renglon, idproducto, cantidad)
VALUES
    (@uuid_DP07r1, @numPedido, 1, @uuid_prod07, 38),
    (@uuid_DP07r2, @numPedido, 2, @uuid_prod04, 52),
    (@uuid_DP07r3, @numPedido, 3, @uuid_prod01, 92);
```

Pedido 08 de 10 (2 renglones)

```
-- Pedido 08 de 10 (2 renglones)
SELECT NumeroPedido INTO @numPedido FROM Pedidos WHERE idpedido = @uuid_pedido08;
SET @uuid_DP08r1 = UUID_TO_BIN(UUID());
SET @uuid_DP08r2 = UUID_TO_BIN(UUID());

INSERT INTO DetallePedidos (idDetallePedido, NumeroPedido, renglon, idproducto, cantidad)
VALUES
    (@uuid_DP08r1, @numPedido, 1, @uuid_prod08, 108),
    (@uuid_DP08r2, @numPedido, 2, @uuid_prod06, 625);
```

Pedido 09 de 10 (1 renglón)

```
-- Pedido 09 de 10 (1 renglón)
SELECT NumeroPedido INTO @numPedido FROM Pedidos WHERE idpedido = @uuid_pedido09;
SET @uuid_DP09r1 = UUID_TO_BIN(UUID());

INSERT INTO DetallePedidos (idDetallePedido, NumeroPedido, renglon, idproducto, cantidad)
VALUES (@uuid_DP09r1, @numPedido, 1, @uuid_prod02, 458);
```

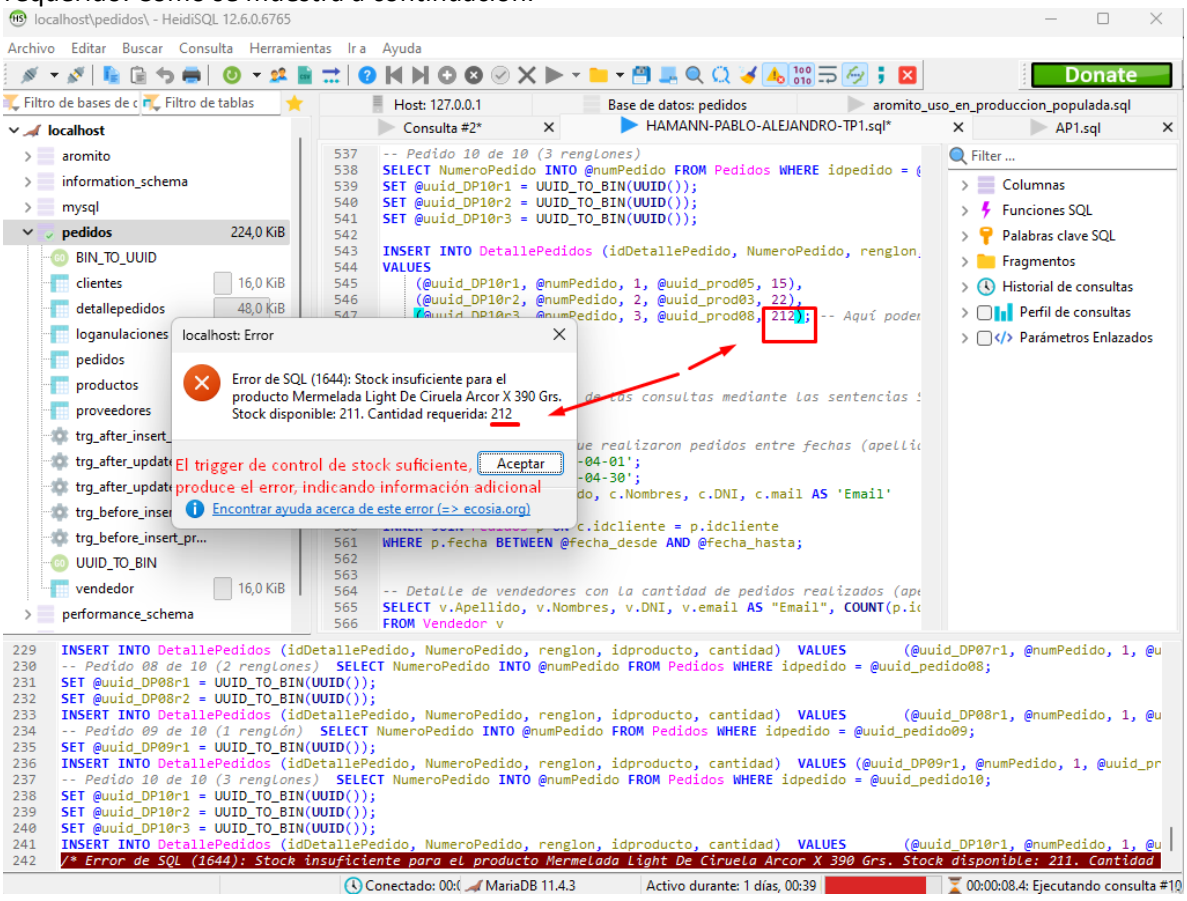
Pedido 10 de 10 (3 renglones)

```
-- Pedido 10 de 10 (3 renglones)
SELECT NumeroPedido INTO @numPedido FROM Pedidos WHERE idpedido = @uuid_pedido10;
```

```
SET @uuid_DP10r1 = UUID_TO_BIN(UUID());
SET @uuid_DP10r2 = UUID_TO_BIN(UUID());
SET @uuid_DP10r3 = UUID_TO_BIN(UUID());

INSERT INTO DetallePedidos (idDetallePedido, NumeroPedido, renglon, idproducto, cantidad)
VALUES
    (@uuid_DP10r1, @numPedido, 1, @uuid_prod05, 15),
    (@uuid_DP10r2, @numPedido, 2, @uuid_prod03, 22),
    (@uuid_DP10r3, @numPedido, 3, @uuid_prod08, 210);
```

En este punto, como la disponibilidad de stock fue mermando (por la cantidad de pedidos dados), si en el renglón 3, en vez de pedir 210 unidades del producto con PK `@DP10r3` (como se se hace en el SQWL mostrado), solicitáramos una cantidad superior a 211, el *trigger* de verificación stock correspondiente, produciría un error, indicando que no hay stock suficiente para el producto requerido. Como se muestra a continuación:



Realizar las siguientes consultas sobre la base de datos

Se desarrolla en los siguientes títulos, con los campos indicados en cada caso durante el enunciado de las actividades. También, se presentan capturas de pantalla (las consultas se realizaron con el software *Navicat Premium 17*).

1. Detalle de clientes que realizaron pedidos entre fechas

```
-- Detalle de clientes que realizaron pedidos entre fechas (apellido, nombres, DNI, correo
electrónico).
SET @fecha_desde = '2025-04-01';
SET @fecha_hasta = '2025-04-30';
SELECT DISTINCT c.Apellido, c.Nombres, c.DNI, c.mail AS 'Email'
FROM Clientes c
INNER JOIN Pedidos p ON c.idcliente = p.idcliente
WHERE p.fecha BETWEEN @fecha_desde AND @fecha_hasta;
```

Resultado de la consulta 1

Message | Summary | Result 1 | Result 2 | Result 3 | Result 4 | Result 5 | Result 6 | Result 7 | Result 8

Data | Info

Apellido	Nombres	DNI	Email
Rojas Valdivia	Lucy Amanda	18465781	lucyamanda23@latinmail.com
Alcaide	Santiago Agustín	39512723	santialcaide@mineral.ru
Pérez	Carlos Enrique	42013728	carlitosperez@gmail.com

2. Detalle de vendedores con la cantidad de pedidos realizados

```
-- Detalle de vendedores con la cantidad de pedidos realizados (apellido, nombres, DNI, correo
electrónico, CantidadPedidos).
SELECT v.Apellido, v.Nombres, v.DNI, v.email AS "Email", COUNT(p.idpedido) AS Cant_Pedidos
FROM Vendedor v
LEFT JOIN Pedidos p ON v.idvendedor = p.idvendedor
GROUP BY v.idvendedor, v.Apellido, v.Nombres, v.DNI, v.email;
```

Resultado de la consulta 2

Message | Summary | Result 1 | Result 2 | Result 3 | Result 4 | Result 5 | Result 6 | Result 7 | Result 8

Data | Info

Apellido	Nombres	DNI	Email	Cant_Pedidos
Garay	Mauricio Elio	36113214	mgaray@msn.com	2
Cabral Perez	Matías	28101438	mcp@outlook.com	5
Castellanos	Matías	24741573	mcastellanos@gmail.com	3

3. Detalle de pedidos con un total mayor a un determinado valor umbral

En este caso, el valor umbral se definió arbitrariamente en 5 millones, y se almacenó en la variable `@valorUmbral`. De forma tal que si se deseara un umbral diferente, solo se modifica la variable, sin modificar la consulta.

```
-- Detalle de pedidos con un total mayor a un determinado valor umbral (NumeroPedido, fecha,
TotalPedido).
SET @valorUmbral = 500000.00;
SELECT p.NumeroPedido, p.fecha AS Fecha, SUM(d.Total) AS TotalPedido
FROM Pedidos p
JOIN detallepedidos d ON p.NumeroPedido = d.NumeroPedido
GROUP BY p.NumeroPedido, p.fecha
HAVING TotalPedido > @valorUmbral;
```

Resultado de la consulta 3

Message | Summary | Result 1 | Result 2 | Result 3 | Result 4 | Result 5 | Result 6 | Result 7 | Result 8

Data | Info

NumeroPedido	Fecha	TotalPedido
1	2025-02-23	2132000,50
6	2025-02-18	958812,00
7	2025-01-08	2628668,80
8	2025-03-05	1306984,50
9	2025-04-10	8701542,00
10	2025-03-21	660370,00

4. Lista de productos vendidos entre fechas.

Tal cual lo indicado en la consigna, `CantidadTotal` se calcula sumando todas las cantidades vendidas del producto. Al igual que en el caso anterior, para evitar modificar la sentencia, el rango de fechas está definido mediante el uso de dos variables: `@fecha_desde` y `@fecha_hasta`.

```
-- Lista de productos vendidos entre fechas. (Descripción, CantidadTotal).
CantidadTotal se calcula sumando todas las cantidades vendidas del producto.
SET @fecha_desde = '2025-04-01';
SET @fecha_hasta = '2025-04-30';
SELECT pr.Descripcion, SUM(dp.cantidad) AS CantidadTotal
FROM Pedidos pe
JOIN DetallePedidos dp ON pe.NumeroPedido = dp.NumeroPedido
JOIN Productos pr ON dp.idproducto = pr.idproducto
WHERE pe.fecha BETWEEN @fecha_desde AND @fecha_hasta
GROUP BY pr.Descripcion;
```

Resultado de la consulta 4

Message | Summary | Result 1 | Result 2 | Result 3 | Result 4 | Result 5 | Result 6 | Result 7 | Result 8

Data | Info

Descripcion	CantidadTotal
Arroz Parboil 1kg Dos Hnos Libre Gluten Sin Tacc	9
Huevo de pascuas Arcor Milk unicornio chocolate 140g	458
Palmito Rodaja 800 Gramos Marolio	14
Turron Arcor 25 Gramos Display De 50 Unidades	12

5. ¿Cuál es el proveedor que realizó más?

Esta pregunta no pudo lelgar a ser del todo bien comprendida, ya que es un tanto ambigua, ya que podría referirse al proveedor del cual más productos se vendieron (por ejemplo, *Arcor*, *Marolio*, etc), o tal vez al proveedor del cual más productos compramos / recibimos. Aquí la tomamos como **cuál es el proveedor del cual más vendimos**.

```sql
-- ¿Cuál es el proveedor que realizó más?
SELECT prov.NombreProveedor, SUM(dp.cantidad) AS TotalProductosVendidos
FROM Proveedores prov
JOIN Productos prod ON prov.idproveedor = prod.idproveedor
JOIN DetallePedidos dp
  ON prod.idproducto = dp.idproducto
GROUP BY prov.idproveedor, prov.NombreProveedor
ORDER BY TotalProductosVendidos DESC
LIMIT 1;
```

Resultado de la consulta 5

Message | Summary | Result 1 | Result 2 | Result 3 | Result 4 | Result 5 | Result 6 | Result 7 | Result 8

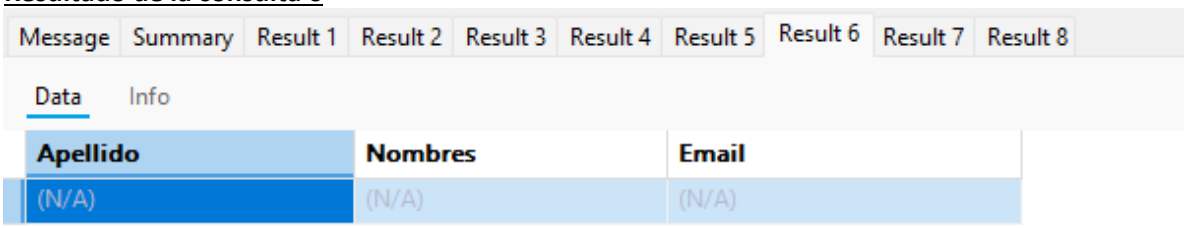
Data | Info

NombreProveedor	TotalProductosVendid
Arcor	995

6. Detalle de clientes registrados que nunca realizaron un pedido

```sql
-- Detalle de clientes registrados que nunca realizaron un pedido. (apellido, nombres, e-mail).
SELECT c.Apellido, c.Nombres, c.mail AS "Email"
FROM Clientes c
LEFT JOIN Pedidos p ON c.idcliente = p.idcliente
WHERE p.idcliente IS NULL;
```

Resultado de la consulta 6



Message | Summary | Result 1 | Result 2 | Result 3 | Result 4 | Result 5 | Result 6 | Result 7 | Result 8

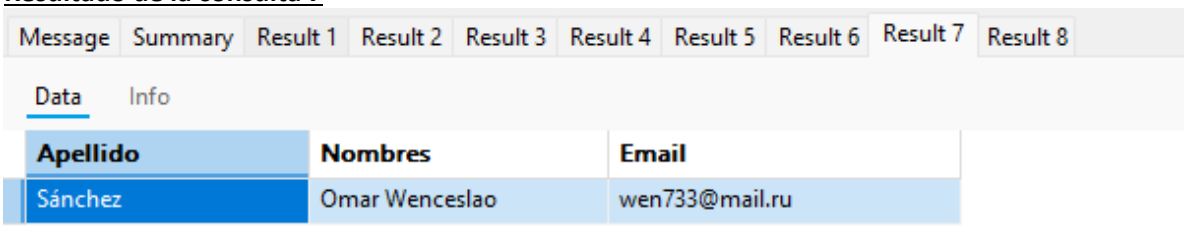
Data | Info

Apellido	Nombres	Email
(N/A)	(N/A)	(N/A)

7. Detalle de clientes que realizaron menos de dos pedidos

```sql
-- Detalle de clientes que realizaron menos de dos pedidos. (apellido, nombres, e-mail).
SET @cantPedidos = 2;
SELECT c.Apellido, c.Nombres, c.mail AS "Email"
FROM Clientes c
LEFT JOIN Pedidos p ON c.idcliente = p.idcliente
GROUP BY c.idcliente, c.Apellido, c.Nombres, c.mail
HAVING COUNT(p.idpedido) < @cantPedidos;
```

Resultado de la consulta 7



Message | Summary | Result 1 | Result 2 | Result 3 | Result 4 | Result 5 | Result 6 | Result 7 | Result 8

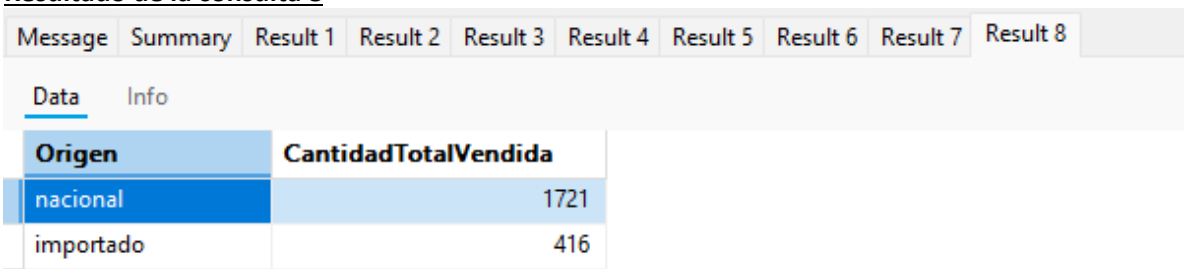
Data | Info

Apellido	Nombres	Email
Sánchez	Omar Wenceslao	wen733@mail.ru

8. Cantidad total vendida por origen de producto

```sql
-- Cantidad total vendida por origen de producto.
SELECT p.origen AS "Origen", SUM(d.cantidad) AS "CantidadTotalVendida"
FROM Productos p
JOIN DetallePedidos d ON p.idproducto = d.idproducto
GROUP BY p.origen;
```

Resultado de la consulta 8



Message | Summary | Result 1 | Result 2 | Result 3 | Result 4 | Result 5 | Result 6 | Result 7 | Result 8

Data | Info

Origen	CantidadTotalVendida
nacional	1721
importado	416