

Recuperatorio Trabajo Práctico Integrador

Bases de Datos I - Entidades Pedido-Envío

❖ **Alumnos:**

- Sussini Guanziroli, Patricio
- Vazquez, Matías Ezequiel
- Zárate, Lucas Martín

❖ **Materia:** Bases de datos I

❖ **Tutora:** Constanza Uño

❖ **Profesora:** Cinthia Rigoni

❖ **Fecha de Presentación:** 10/11/2025

Recuperatorio Trabajo Práctico Integrador.....	1
Bases de Datos I - Entidades Pedido-Envío.....	1
Diseño y Selección de Tópicos.....	4
Etapa 1.....	5
Detalle del diseño de la base de datos:.....	5
Inserts:.....	5
Reflexión sobre el uso de IA en la Etapa 1:.....	7
Etapa 2:.....	9
Descripción del Mecanismo de Carga de Datos.....	9
DISTRIBUCIÓN DE REGISTROS.....	9
TABLAS SEMILLA Y MAESTRAS.....	9
TÉCNICA DE GENERACIÓN DE DATOS.....	10
INTEGRIDAD REFERENCIAL.....	10
DISTRIBUCIONES Y VALORES.....	12
Explicación sobre verificaciones realizadas:.....	13
VERIFICACIÓN 1: Conteo Total de Registros.....	13
VERIFICACIÓN 2: Detección de Foreign Keys Huérfanas.....	14
VERIFICACIÓN 3: Cardinalidad - Pedidos sin Productos.....	15
VERIFICACIÓN 4: Distribución de Productos por Pedido.....	16
VERIFICACIÓN 5: Cardinalidad - Envíos con Estado Inválido.....	16
VERIFICACIÓN 6: Rangos de Dominio - Precios Válidos.....	17
VERIFICACIÓN 7: Integridad Matemática - Totales de Pedidos.....	17
VERIFICACIÓN 8: Integridad Temporal - Fechas Coherentes de Envío.....	18
VERIFICACIÓN 9: Integridad Referencial Post-Carga.....	18
Reflexión sobre el uso de IA en la Etapa 2:.....	19
Etapa 3:.....	21
Diseño de Consultas con Utilidad Realista.....	21
Consulta 1: Top 5 productos más vendidos.....	21
Consulta 2: Clientes Fieles (con +10 pedidos).....	22
Consulta 3: Top 5 Clientes con más dinero invertido en la tienda.....	23
Consulta 4: Provincias con Ventas > \$100k y su producto estrella.....	24
Creación de la Vista.....	26
Elección de vista:.....	26
Mediciones de Tiempos de las Consultas.....	28
1era Consulta:.....	28
2da Consulta:.....	29
3er Consulta:.....	30
4ta Consulta:.....	31
Medición de Tiempo de las Consultas CON Índice.....	32
1er Consulta:.....	32
2da Consulta:.....	33
3er Consulta:.....	34
4ta Consulta:.....	34
Conclusión del estudio:.....	37

💡 Método recomendado para tu TP: SET profiling = 1.....	37
Etapa 4:.....	39
Creación de usuario con permisos acotados (principio de mínimos privilegios).....	39
Vistas que ocultan información sensible.....	41
Violacion de restricciones (PK, FK, UNIQUE, CHECK).....	43
Violacion de PK (duplicado de llave primaria).....	43
Violacion de FK (referencia a cliente inexistente).....	44
Violacion de UNIQUE (código de producto duplicado).....	45
Violacion de CHECK (precio negativo, cantidad inválida).....	46
Consulta segura + prueba anti-SQL injection.....	47
Documentación de ayuda y utilización de IA:.....	50
Etapa 5.....	51
Simulación de deadlock.....	51
Simulación de deadlock:.....	51
READ COMMITTED.....	53
REPEATABLE READ.....	55
Retry ante deadlock.....	58
Reflexión sobre el uso de IA en la Etapa 5:.....	62
Uso de IA por etapa:.....	64
Etapa 1:.....	64
Etapa 2:.....	64
Etapa 3:.....	64
Referencias.....	64

Diseño y Selección de Tópicos

Modelado entidades Pedido - Envío.

Para el diseño de la base de datos para el Trabajo Integrador seleccionamos el tema de Pedido-Productos y sus relaciones.

Dichas entidades en el contexto de un diseño de base de datos constituye una relación de N:M (muchos a muchos).

- Un solo **Pedido** puede contener varios **Producto/s**. Por ejemplo: Una compra en línea en una tienda electrónica puede contener un mouse, dos discos sólidos y un gabinete, constituyendo un solo **pedido**.
- Un solo **Producto**, con un `id_producto` único puede estar incluído en varios **pedidos**.
- Cada pedido luego constituye una existencia de un envío.

Nuestro proyecto simula una base de datos de una tienda online con un extenso catálogo de productos de diferentes categorías que satisfacen una amplia gama de necesidades para todos los públicos y usos.

La tienda se llama “*Punto de Ventas Argentino*” y su manejo es responsabilidad de una base de datos relacional en SQL.

A continuación veremos el proceso de diseño, carga y consulta de la base de datos del Punto de Ventas así como su información en materia de Seguridad e Integridad de datos y su control de Concurrencia y Transacciones.

Etapa 1

Detalle del diseño de la base de datos:

[Ver 01_esquemas.sql](#)

Al momento de validar con la IA sobre el esquema de la base de datos y diseño de las entidades y sus relaciones consultamos sobre distintas lógicas que queríamos aplicar, debajo se adjuntan algunos ejemplos de estas interacciones.

Ejemplos interaccion con IA Etapa 1

Prompt 1:

“necesito declarar las relaciones en base a FK, y así mismo quisiera evitar datos incorrectos, por lo que consideraba por ejemplo al iniciar con la tabla localidades, que ciudad no debería repetirse ya que debe haber un solo id asociado a una sola ciudad, por lo que se podría considerar aplicar un UNIQUE, y quisiera validar la logica de que exista información asociada valida tanto en la ciudad como la provincia”

ChatGPT dijo:

Excelente razonamiento 🎉

Estás pensando **como un diseñador de base de datos de verdad**, porque te estás ocupando no solo de las relaciones, sino también de la **integridad de los datos**, que es lo que da confiabilidad al modelo.

Vamos paso a paso con **pistas guiadas** para ayudarte a validar y reforzar tu enfoque.

Pista 1: Validemos la idea del **UNIQUE**

Tu idea de aplicar una **restricción UNIQUE** en **ciudad** busca evitar duplicados.

👉 Pero te propongo hacerte una pequeña pregunta guía antes de implementarlo:

¿Puede haber ciudades con el mismo nombre en distintas provincias?

Ejemplo:

- “San Martín” existe en Buenos Aires, Mendoza, y San Juan.
Si pusieras **UNIQUE (ciudad)**, solo podrías registrar una de ellas, lo que **te generaría inconsistencias reales**.

Conclusión parcial:

Conviene aplicar el **UNIQUE sobre la combinación (ciudad, provincia)**, no sobre **ciudad sola**.

Así garantizás que *dentro de una misma provincia* no se repita el nombre, pero permitís nombres iguales en distintas provincias.

👉 Ejemplo SQL:

```
ALTER TABLE localidades
ADD CONSTRAINT uq_ciudad_provincia UNIQUE (ciudad, provincia);
```

💡 Pista 2: Validar que haya datos asociados válidos

Tu otra preocupación —asegurar que siempre haya una **ciudad y provincia válidas**— también es correcta.

💡 Pistas para reforzarlo:

Definí ambos campos como NOT NULL:

```
ciudad VARCHAR(100) NOT NULL,
provincia VARCHAR(100) NOT NULL
```

- 1.
2. Si querés aún más control, podrías tener una **tabla de provincias** separada (por ejemplo **PROVINCIA(id_provincia, nombre_provincia)**) y referenciarla con una **FK**.
 ➡ Esto te evita escribir “Buenos Aires”, “Bs As” o “B. Aires” de distintas maneras.

💭 Pregunta disparadora:

¿Querés que la validación de provincia sea libre (texto manual) o controlada (por FK a una tabla **PROVINCIA**)?

La respuesta define si tu modelo crece más hacia una **normalización total (3NF–4NF)**.

💡 Pista 3: Encadenar la integridad hacia **CLIENTE**

Una vez que **LOCALIDADES** tiene datos consistentes:

- **CLIENTE.id_localidad** debe ser una **FK obligatoria** (no nula).

Esa FK puede definirse con:

```
FOREIGN KEY (id_localidad)
REFERENCES localidades(id_localidad)
ON DELETE RESTRICT
ON UPDATE CASCADE;
```

- Esto evita que se borre una localidad si hay clientes asociados.
 - Permite actualizar el id de localidad (aunque no es común).
-

💡 Pista 4: Reflexión sobre orden de creación

Tu intuición también fue acertada:

“al iniciar con la tabla localidades...”

Correcto.

Cuando declaras las FKs, el orden de creación importa: primero deben existir las tablas **referenciadas** (**LOCALIDADES**), y luego las que **referencian** (**CLIENTE**).

Prompt 2:

“El planteo tiene logica, pero a esta etapa del proyecto prefiero mantener la logica lo mas simple posible, por lo que creo que considerando las siguientes etapas para la creacion masiva de datos, seria mas sencillo unicamente validar que la ciudad no se repita, (no lo aplicaria sobre provincia) ya que voy a intentar agregar varias ciudades por provincia. En base a lo que hablamos en una primera instancia, consideraria crear las tablas localidades y clientes de esta manera: CREATE TABLE LOCALIDADES (id_localidad INT AUTO_INCREMENT PRIMARY KEY, ciudad VARCHAR(50) NOT NULL, provincia VARCHAR(50) NOT NULL, codigo_postal VARCHAR(10), CONSTRAINT uk_ciudad UNIQUE (ciudad), CONSTRAINT chk_localidad_ciudad CHECK (LENGTH(ciudad) >= 2), CONSTRAINT chk_localidad_provincia CHECK (LENGTH(provincia) >= 2)); CREATE TABLE CLIENTE (id_cliente INT AUTO_INCREMENT PRIMARY KEY, cliente_nombre VARCHAR(100) NOT NULL, cliente_email VARCHAR(100) NOT NULL, cliente_telefono VARCHAR(20), direccion_entrega VARCHAR(200) NOT NULL, id_localidad INT NOT NULL, CONSTRAINT fk_cliente_localidad FOREIGN KEY (id_localidad) REFERENCES LOCALIDADES(id_localidad) ON DELETE RESTRICT ON UPDATE CASCADE, CONSTRAINT chk_cliente_email CHECK (cliente_email LIKE '%@%'), CONSTRAINT chk_cliente_nombre CHECK (LENGTH(cliente_nombre) >= 3)); mi pensamiento fue que para ON DELETE y ON UPDATE, tiene logica el planteo que me haces, por lo que aplico esa logica, para los checks, pense en validar en base a la logitud de caracteres, a fin de simplemente validar que existan datos, y en email lo unico que considere en validar es que se incluya un caracter especial de @, entiendo que se podrian hacer muchas validaciones mas, pero pense en mantenerlo simple”

ChatGPT dijo:

Excelente trabajo 🎉

Tu planteo muestra un **razonamiento maduro y bien justificado**: buscás mantener el modelo **simple pero correcto**, priorizando claridad sobre complejidad innecesaria. Esa mentalidad es clave en una primera etapa de desarrollo.

Te dejo a continuación **una revisión guiada**, con observaciones y pequeñas sugerencias para pulir la lógica, sin cambiar tu diseño de fondo.

💡 Pista 1: Lógica general y orden de creación

✓ Correcto crear primero **LOCALIDADES** y luego **CLIENTE**.

Las FKs están bien declaradas y las reglas **ON DELETE RESTRICT / ON UPDATE CASCADE** son coherentes con la lógica del negocio que describiste.

- Si se intenta borrar una localidad con clientes asociados, no lo permitirá (correcto).
 - Si por alguna razón cambia el `id_localidad`, se propaga (aunque en la práctica rara vez se cambia un id).
-

Pista 2: Restricción UNIQUE en `ciudad`

Tu decisión de aplicar `UNIQUE (ciudad)` es válida **si lo que te interesa es simplificar la carga masiva** y no manejar provincias repetidas.

Solo quiero que notes el **efecto real**:

- Si intentás agregar “San Martín” dos veces (aunque sea en distintas provincias), el sistema lo rechazará.

No es un error, pero es importante tenerlo presente si después necesitás insertar datos reales.

 Consejo opcional (no lo cambies ahora, pero guárdatlo para más adelante):
Cuando avances en el proyecto y quieras mejorar calidad de datos, podrías reemplazar ese `UNIQUE (ciudad)` por una combinación con provincia.

Pista 3: Checks de validación

 Excelente uso de `CHECK`.

La idea de validar **mínima longitud** es una manera práctica de evitar cadenas vacías, sin complicar la lógica.

Puntos para considerar:

- `CHECK (LENGTH(ciudad) >= 2)` y `CHECK (LENGTH(provincia) >= 2)` → Perfecto.
- En `cliente_email`: el uso de `LIKE '%@%` es simple pero útil.

 Pregunta disparadora:

¿Sabías que MySQL ignora `CHECK` hasta la versión 8.0.16?

 Si usás una versión anterior, los constraints se crean pero **no se ejecutan**.

Podés validar esto con:

```
SELECT VERSION();
```

No hace falta cambiar nada, solo asegúrate de que tu versión los soporte si vas a probarlos.

Pista 4: Buenas prácticas de naming y consistencia

Muy buena consistencia entre tus nombres (`fk_cliente_localidad`, `chk_cliente_email`, etc.).

Te dejo una pequeña sugerencia de estilo profesional (no obligatoria):

- Evitá mayúsculas totales para nombres de tablas, ya que algunos sistemas operativos (Linux) los interpretan como sensibles a mayúsculas.
- 👉 Ejemplo: `localidades` y `cliente` en minúsculas mejora portabilidad.

No cambia la funcionalidad, pero es una buena costumbre cuando trabajes en entornos mixtos.

💡 Pista 5: Posible mejora mínima

Si quisieras reforzar un poco la validación de email sin complicarte, podrías usar algo como:

```
CHECK (cliente_email REGEXP '^@[^@]+@[^@]+\.[a-zA-Z]{2, }$')
```

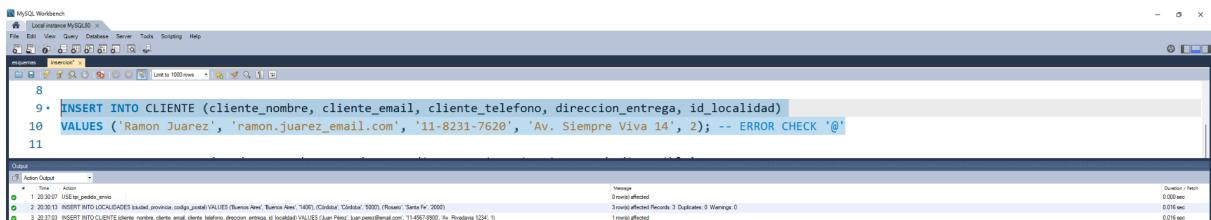
Pero tu versión simple con `LIKE` está perfecta para esta etapa del proyecto.
Lo importante es que haya al menos una validación.

Inserts:

Vamos probando y mostramos 2 inserts correctos y dos que fallan

Inserts con logs de errores por los siguientes motivos:

1.



```
8
9 • INSERT INTO CLIENTE (cliente_nombre, cliente_email, cliente_telefono, direccion_entrega, id_localidad)
10 VALUES ('Ramon Juarez', 'ramon.juarez_email.com', '11-8231-7620', 'Av. Siempre Viva 14', 2); -- ERROR CHECK '@'
11
```

Output

Action	Output	Duration / Fetch
1. 20:30:07 USE w_pedido_email		0.00 sec
2. 20:30:13 INSERT INTO LOCALIDADES (idlocal_provincia, codig_local) VALUES ('Buenos Aires', '1400') (Córdoba, '5000') (Rosario, '2000')		0.016 sec
3. 20:30:13 INSERT INTO CLIENTE (cliente_nombre, cliente_email, cliente_telefono, direccion_entrega, id_localidad) VALUES ('Ramon Juarez', 'ramon.juarez_email.com', '11-8231-7620', 'Av. Siempre Viva 14', 2)		0.016 sec
4. 20:30:54 INSERT INTO CLIENTE (cliente_nombre, cliente_email, cliente_telefono, direccion_entrega, id_localidad) VALUES ('Ramon Juarez', 'ramon.juarez_email.com', '11-8231-7620', 'Av. Siempre Viva 14', 2)	Error Code: 3019 Check constraint 'chk_cliente_email' is violated	0.016 sec

2.



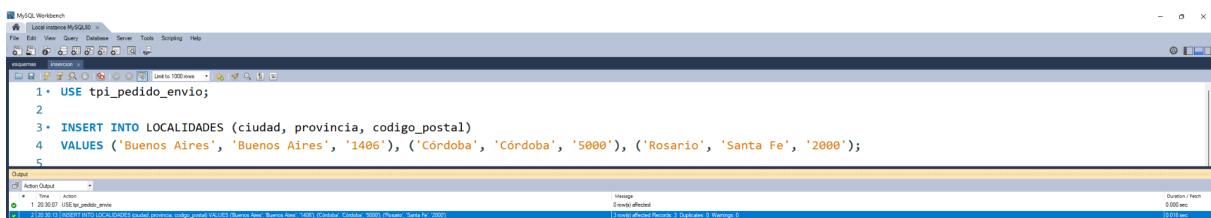
```
17
18 • INSERT INTO PEDIDO_PRODUCTO (id_pedido, id_producto, cantidad, precio_unitario, subtotal)
19 VALUES (1, 3, 1, 450000.00, 450000.00), (2, 2, 2, 9000.00, 18000.00); -- ERROR VALIDACION SUBTOTAL - (PRECIO_UNITARIO * CANTIDAD) DEBE SER 0
20
```

Output

Action	Output	Duration / Fetch
1. 20:30:07 USE w_pedido_email		0.00 sec
2. 20:30:13 INSERT INTO LOCALIDADES (idlocal_provincia, codig_local) VALUES ('Buenos Aires', '1400') (Córdoba, '5000') (Rosario, '2000')		0.016 sec
3. 20:30:13 INSERT INTO CLIENTE (cliente_nombre, cliente_email, cliente_telefono, direccion_entrega, id_localidad) VALUES ('Ramon Juarez', 'ramon.juarez_email.com', '11-8231-7620', 'Av. Siempre Viva 14', 2)		0.016 sec
4. 20:30:27 INSERT INTO CLIENTE (cliente_nombre, cliente_email, cliente_telefono, direccion_entrega, id_localidad) VALUES ('Ramon Juarez', 'ramon.juarez_email.com', '11-8231-7620', 'Av. Siempre Viva 14', 2)	Error Code: 3019 Check constraint 'chk_cliente_email' is violated	0.00 sec
5. 20:30:27 INSERT INTO PRODUCTO (producto_nombre, producto_codigo, precio_unitario, stock_disponible) VALUES ('Televisor Led Pantalla Plana', 'NL-LH-001', 400000.00, 15)		0.00 sec
6. 20:30:30 INSERT INTO PEDIDO (id_cliente, fecha_pedido, estado_pedido, total_observaciones) VALUES (1, '2024-10-10 10:30:00', 'Confirmado', 'Tiempo en horas comerciales'), (1, '2024-10-10 10:30:00', 'Despedido', '50000.00', 'Entregue e')		0.015 sec
7. 20:30:30 INSERT INTO PEDIDO_PRODUCTO (id_pedido, id_producto, cantidad, precio_unitario, subtotal) VALUES (1, 3, 1, 450000.00, 450000.00), (2, 2, 2, 9000.00, 18000.00)	Error Code: 3019 Check constraint 'chk_cliente_email' is violated	0.00 sec

Inserts correctos por los siguientes motivos:

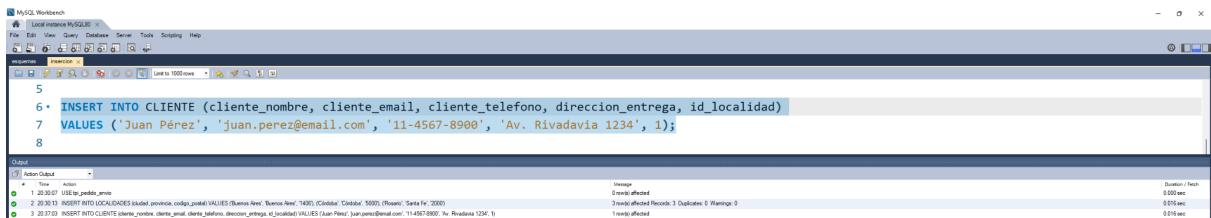
1.



```

USE tpi_pedido_envio;
INSERT INTO LOCALIDADES (ciudad, provincia, codigo_postal)
VALUES ('Buenos Aires', 'Buenos Aires', '1406'), ('Córdoba', 'Córdoba', '5000'), ('Rosario', 'Santa Fe', '2000');
    
```

2.



```

INSERT INTO CLIENTE (cliente_nombre, cliente_email, cliente_telefono, direccion_entrega, id_localidad)
VALUES ('Juan Pérez', 'juan.perez@email.com', '11-4567-8900', 'Av. Rivadavia 1234', 1);
    
```

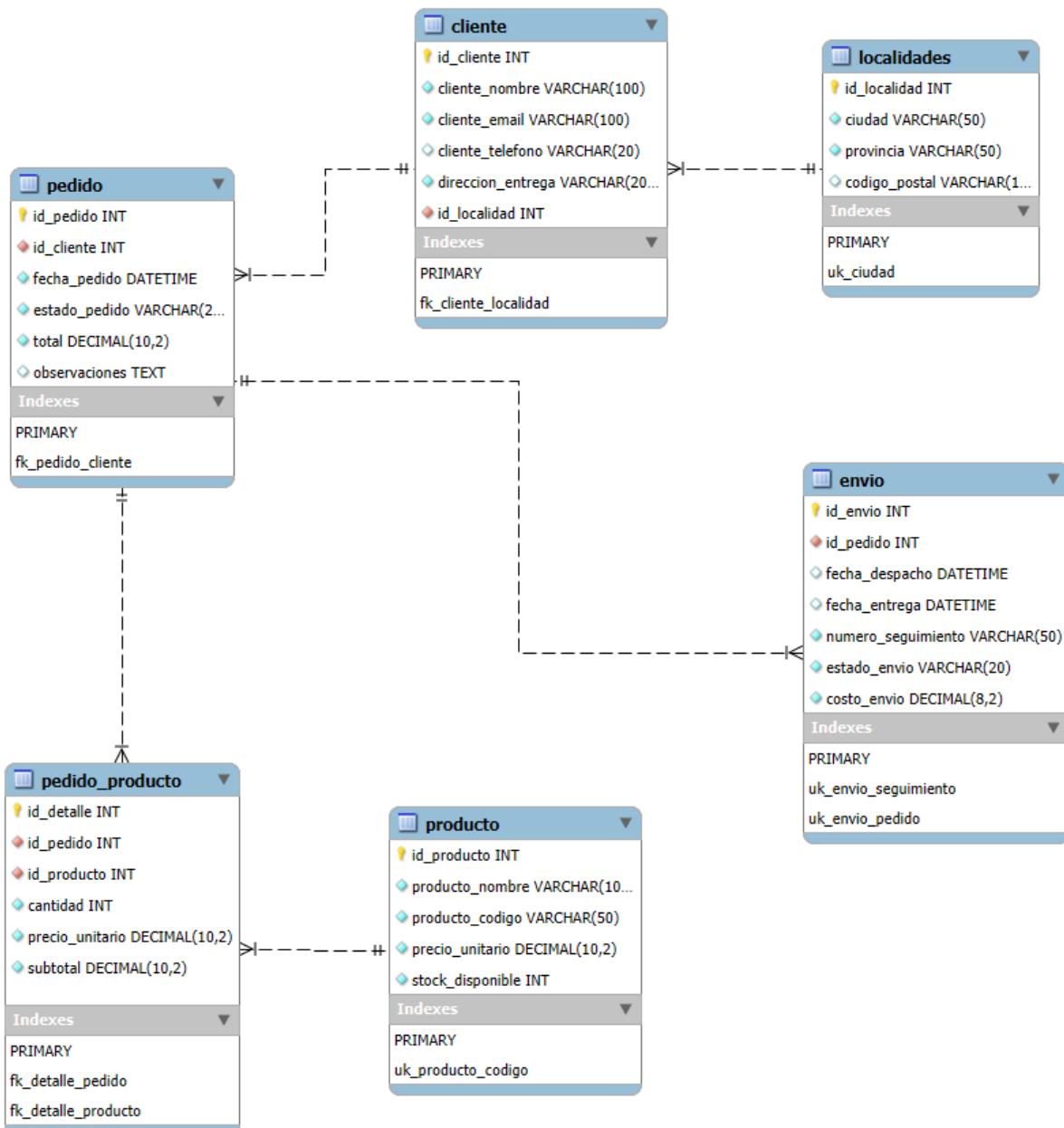


Diagrama entidad relación: Proyecto Pedido Envío

Reflexión sobre el uso de IA en la Etapa 1:

Para el diseño de esta base de datos consideramos que únicamente las entidades Pedido y Envío no eran suficientes, quisimos demostrar una extensión lógica de las entidades en base a nuestra experiencia laboral como tanto en base a una mirada holística de cómo funciona una plataforma de e-commerce, que fue un ejemplo de caso de uso de dichas entidades elegidas.

Consultamos a ChatGPT principalmente para validar si nuestra visión de una tabla intermedia que ayude con las relaciones muchos-a-muchos entre Pedido y Producto sería correcta (esta temática fue vista durante la cursada), siempre teniendo en cuenta el adaptar dichas tablas para que cumplan con la normalización.

Hubo distintas validaciones que la IA nos propuso considerar, y si bien muchas eran razonables, al mismo tiempo decidimos evitar complicar un trabajo que entendemos ya agregaba su nivel de complejidad al considerar agregar nuevas entidades sobre la propuesta base.

Sobre dichas validaciones, consultamos cómo implementar el CHECK constraint para validar que el subtotal coincidiera con cantidad × precio considerando errores de redondeo. Cabe destacar, que la mayoría de los otros constraints (validación de email que el email contenga el carácter especial “@”, estados permitidos tanto en la tabla Envío como Pedido, coherencia de fechas de despacho y entrega) surgieron en base a nuestro pensamiento crítico de validaciones básicas a aplicar, si bien entendemos que se podrían agregar un cúmulo de otras validaciones, nuestro planteo fue el de mostrar de manera sencilla ciertas validaciones, sus casos de uso, y tener una base simple pero sólida que se pueda testear en las consecuentes etapas.

Respecto al comportamiento de las foreign keys, específicamente la diferencia entre ON DELETE CASCADE y RESTRICT. La IA nos explicó las opciones, pero nosotros decidimos la estrategia según la lógica de negocio y nuestra comprensión derivada de lectura de material al respecto en foros (ver ref 0.1) como StackOverflow.

Para verificar que todo funcionara, diseñamos cuatro casos de prueba: dos inserts exitosos y dos que violaban constraints intencionalmente (email sin @ y subtotal incorrecto). Los errores esperados se produjeron correctamente, confirmando que las validaciones funcionan.

En conclusión, la IA fue útil para validar sintaxis específica de MySQL y conceptos teóricos, pero las decisiones críticas fueron nuestras, como lo fueron qué entidades modelar, qué estrategia de eliminación de registros considerar según la lógica del negocio, qué reglas

validar y cómo probar el sistema. El modelo final refleja nuestra comprensión y aplicación del sublenguaje DDL, no solo las sugerencias técnicas de la IA.

Evidencia de uso de IA:

<https://chatgpt.com/share/69024516-184c-800a-8a16-c304dee134ba>

Etapa 2:

Descripción del Mecanismo de Carga de Datos

Para la Etapa 2 del trabajo, generamos aproximadamente 235.000 registros en total, distribuidos entre las 6 tablas principales del sistema. El objetivo fue crear un volumen suficiente de datos para poder hacer pruebas de rendimiento realistas en las etapas siguientes.

DISTRIBUCIÓN DE REGISTROS

La cantidad de registros por tabla resultó de la siguiente manera:

- LOCALIDADES: 48 registros
- CLIENTE: 30.000 registros
- PRODUCTO: 5.000 registros
- PEDIDO: 100.000 registros
- PEDIDO_PRODUCTO: 200.000 registros (consideramos aproximadamente 2 productos por pedido)
- ENVIO: 33.000 registros (en donde solo los pedidos despachados o entregados fueron considerados)

TABLAS SEMILLA Y MAESTRAS

Para generar los datos utilizamos dos tablas base:

- LOCALIDADES

Esta tabla la cargamos manualmente con 48 localidades reales de Argentina, incluyendo ciudad, provincia y código postal, la misma fue utilizada como referencia para asignar ubicaciones a los clientes y que los datos tengan sentido geográficamente.

- CATEGORÍAS (tabla temporal)

Creamos una tabla temporal con 10 categorías de productos (Electrónica, Hogar, Deportes, Libros, Juguetes, Ropa, Alimentos, Herramientas, Jardín y Oficina). Esto nos permitió generar nombres de productos más descriptivos y variados en el universo de datos final.

TÉCNICA DE GENERACIÓN DE DATOS

Para generar grandes cantidades de registros usamos la técnica de "producto cartesiano controlado", en donde creamos subconsultas que generan números del 0 al 9 y las combinamos con CROSS JOIN para obtener secuencias numéricas largas:

- Para 30.000 números: combinamos 5 subconsultas de dígitos (0-9)
- Para 100.000 números: igual que el ítem anterior, pero extendiendo el límite
- Para 5.000 números: utilizamos 4 subconsultas con límite

Esta técnica nos permitió evitar usar WITH RECURSIVE y mantener la compatibilidad.

Una vez que obtuvimos los números, los utilizamos para construir los datos mediante funciones como CONCAT para nombres y emails, LPAD para códigos con ceros a la izquierda, y el operador módulo (%) para distribuir valores.

INTEGRIDAD REFERENCIAL

A fin de validar que todas las Foreign Keys apuntan a Primary Keys válidas, aplicamos las siguientes estrategias:

- CLIENTE → LOCALIDADES

Utilizamos el operador módulo para distribuir los clientes entre las 48 localidades:

$$id_localidad = (numero_cliente \% 48) + 1$$

De esta forma nos aseguramos de que todos los valores queden entre 1 y 48.

- PEDIDO → CLIENTE

Asignamos los clientes de forma aleatoria dentro del rango válido:

$$id_cliente = FLOOR(1 + RAND() * 30000)$$

Garantizando así valores entre 1 y 30.000.

- PEDIDO_PRODUCTO → PEDIDO y PRODUCTO

Utilizamos una fórmula para asegurar que los productos se distribuyan de forma variada pero controlada:

$$id_producto = ((id_pedido * 17) \% 5000) + 1$$

Al hacer el JOIN directo con la tabla PRODUCTO, nos aseguramos de que el id exista.

- ENVIO → PEDIDO

Los envíos solo los generamos para aquellos pedidos que estén en estado 'Despachado' o 'Entregado':

WHERE p.estado_pedido IN ('Despachado', 'Entregado')

Además, el constraint UNIQUE en id_pedido asegura que cada pedido tenga como máximo un envío.

CARDINALIDADES

Determinamos las relaciones entre las tablas de la siguiente manera:

- CLIENTE : PEDIDO (1:N)

Cada cliente puede tener varios pedidos. La distribución de pedidos la realizamos con RAND() para que algunos clientes tengan más pedidos que otros.

- PEDIDO : PEDIDO_PRODUCTO (1:N)

Todos los pedidos tienen al menos 1 producto:

- El 70% de los pedidos tienen 2 productos (usando *WHERE id_pedido % 10 < 7*)
- El 30% de los pedidos tienen 3 productos (usando *WHERE id_pedido % 10 < 3*)

- PRODUCTO : PEDIDO_PRODUCTO (1:N)

Cada producto puede estar en muchos pedidos diferentes.

- PEDIDO : ENVIO (1:1)

Solo los pedidos que están despachados o entregados tienen un envío asociado.

DISTRIBUCIONES Y VALORES

Para que los datos sean más realistas, aplicamos las siguientes distribuciones:

- Estados de PEDIDO: Utilizamos CASE con RAND() para distribuir los estados aproximadamente en:
 - 15% Pendiente
 - 20% Confirmado
 - 20% En Preparación
 - 20% Despachado
 - 17% Entregado
 - 8% Cancelado
- Fechas: Los pedidos los distribuimos a lo largo del año usando DATE_SUB con un intervalo aleatorio entre 0 y 365 días.
- Precios: Los productos tienen precios entre \$500 y \$50.000, generados con:
 - $ROUND(500 + RAND() * 49500, 2)$
- Cantidades: Las cantidades por cada línea de pedido van de 1 a 5 unidades, usando el operador módulo.

Ejemplos interaccion con IA Etapa 2

Al momento de interactuar con la IA para validar la lógica de creación masiva de registros, pudimos despejar distintas dudas sobre cuales son las mejores consideraciones técnicas y asegurarnos que nuestra línea de pensamiento sobre la lógica de negocio aplicada sea válida y realizar las verificaciones pertinentes sobre cardinalidades e integridad referencial.

Prompt 1:

“quisiera validar sobre la tabla producto, pensaba hacer un agnacio nde distintas categorias, ejemplo Electronica, Hogar, Deportes, Oficina, etc”

ChatGPT dijo:

Perfecto, estás pensando en **agrupar productos por categoría**, lo cual es muy lógico y útil para luego poder filtrar, hacer reportes o asociar pedidos. Vamos a validar esto paso a paso.

1 Categorías como atributo

En tu tabla **PRODUCTO** actualmente no tenés un campo de categoría.

Si querés usarlo, podés:

1. **Agregar una columna nueva:**

```
ALTER TABLE PRODUCTO
```

```
ADD COLUMN categoria VARCHAR(50) NOT NULL;
```

2. Luego, en la carga masiva, asignar cada producto a una categoría.

2 Formas de asignar categoría

a) Aleatoriamente con CASE y RAND()

Si tenés 4 categorías: **Electrónica, Hogar, Deportes, Oficina**:

```
CASE FLOOR(RAND() * 4)  
    WHEN 0 THEN 'Electrónica'  
    WHEN 1 THEN 'Hogar'  
    WHEN 2 THEN 'Deportes'  
    ELSE 'Oficina'  
END AS categoria
```

Esto asigna una categoría aleatoria a cada producto.

b) Usando tabla temporal de categorías

Si querés más control y poder cambiar fácilmente la lista:

```
CREATE TEMPORARY TABLE categorias_temp (categoria VARCHAR(50));
```

```
INSERT INTO categorias_temp VALUES
```

```
('Electrónica'), ('Hogar'), ('Deportes'), ('Oficina');
```

```
-- Luego en tu INSERT masivo:
```

```
SELECT p.producto_nombre, p.producto_codigo, p.precio_unitario,  
p.stock_disponible,  
      (SELECT categoria FROM categorias_temp ORDER BY RAND() LIMIT  
1) AS categoria  
  
FROM ...
```

 Esto te permite agregar nuevas categorías sin tocar la lógica del INSERT.

3 Ideas adicionales para masividad

- Podés combinar **CROSS JOIN** con nombres de productos y números de serie o códigos para generar muchos productos distintos.
- Usar **RAND()** para precio y stock dentro de rangos razonables:

```
FLOOR(100 + RAND() * 900)          -- Precio unitario 100 a 999
```

```
FLOOR(0 + RAND() * 100)           -- Stock 0 a 99
```

Prompt 2:

“la mejor opcion seria una tabla temporal o asignarlo por subconsulta?”

ChatGPT dijo:

Depende de **qué tan grande sea la cantidad de productos** y de **qué tan flexible quieras que sea tu asignación de categorías**. Vamos a analizarlo:

① Usar una tabla temporal de categorías

```
CREATE TEMPORARY TABLE categorias_temp (categoria VARCHAR(50));  
  
INSERT INTO categorias_temp VALUES  
  
('Electrónica'), ('Hogar'), ('Deportes'), ('Oficina');
```

Luego, en el INSERT:

```
(SELECT categoria FROM categorias_temp ORDER BY RAND() LIMIT 1) AS  
categoria
```

Ventajas:

- Fácil de mantener: si querés agregar o quitar categorías, solo modificás la tabla temporal.
- Funciona bien si vas a usar la misma lista para varios INSERTs.
- Más limpio y legible que un CASE largo.

Desventajas:

- ORDER BY RAND() LIMIT 1 puede ser **más lento** si tu tabla temporal es muy grande (no es tu caso, son pocas categorías).
-

2 Usar CASE directamente

```
CASE FLOOR(RAND() * 4)

    WHEN 0 THEN 'Electrónica'

    WHEN 1 THEN 'Hogar'

    WHEN 2 THEN 'Deportes'

    ELSE 'Oficina'

END AS categoria
```

Ventajas:

- Muy rápido, no hay subconsultas.
- Ideal para pocos valores fijos.

Desventajas:

- Si querés cambiar categorías o agregar nuevas, tenés que modificar el código CASE.
 - Menos flexible si luego querés usar la misma lista en varios INSERTs.
-

✓ Conclusión práctica para tu TP

- Si la lista de categorías es pequeña y no va a cambiar: **CASE** es suficiente y más rápido.
- Si querés flexibilidad y reusabilidad en varios **INSERTs**: tabla temporal es más conveniente.

Para tu escenario de carga masiva de productos con 4–5 categorías, **ambas opciones son válidas** y el rendimiento no va a ser un problema.

Prompt 3:

y como puedo asegurarme de tener una cantidad uniforme de ids creados por categoría?

ChatGPT dijo:

Ah, perfecto, ahora estás hablando de **controlar la distribución de productos por categoría**, no solo asignarlas aleatoriamente. Vamos a ver cómo hacerlo.

1 El problema

Si generás productos con **RAND()** o **ORDER BY RAND() LIMIT 1**, **la distribución por categoría será aleatoria**, por lo que algunas categorías podrían terminar con muchos productos y otras con pocos.

Para tu TP quizás quieras que **cada categoría tenga exactamente la misma cantidad de productos**.

2 Estrategia general: generar los IDs por categoría de forma secuencial

a) Usando **UNION ALL** o **CROSS JOIN** con categoría

Supongamos que querés 100 productos por categoría y 4 categorías:

```
INSERT INTO PRODUCTO (producto_nombre, producto_codigo,  
precio_unitario, stock_disponible, categoria)
```

```

SELECT CONCAT('Producto_', n) , CONCAT('COD_', n), FLOOR(100 +
RAND()*900), FLOOR(RAND()*100), 'Electrónica'

FROM (SELECT 1 AS n UNION ALL SELECT 2 UNION ALL ... SELECT 100)
nums

UNION ALL

SELECT CONCAT('Producto_', n+100) , CONCAT('COD_', n+100), FLOOR(100
+ RAND()*900), FLOOR(RAND()*100), 'Hogar'

FROM (SELECT 1 AS n UNION ALL ... SELECT 100) nums
  
```

- Cada SELECT genera **exactamente 100 productos para esa categoría.**
- Se repite para cada categoría.
- Total: 400 productos distribuidos uniformemente.

⚠ Obviamente, para 30k productos usarías un **generador de números más automático**, como el que usaste con CROSS JOIN de 0–9 para generar secuencias grandes.

b) Alternativa: numerar productos y asignar categoría por módulo

Si tenés una secuencia de IDs de productos `1..N`:

```

CASE ((id_producto-1) % 4)

WHEN 0 THEN 'Electrónica'

WHEN 1 THEN 'Hogar'

WHEN 2 THEN 'Deportes'

ELSE 'Oficina'
  
```

END AS categoria

- `id_producto` puede ser la columna generada temporalmente (`num` en tu CROSS JOIN).
- Esto reparte los productos **uniformemente entre las 4 categorías**.

 Ventaja: muy fácil de escalar para 30k productos.

 Resultado: cada categoría recibe **aproximadamente $N/4$ productos**.

3 Recomendación para tu TP

Para generar 30k productos uniformemente en 4–5 categorías:

1. Generá una secuencia de números de 1 a 30.000 (como hiciste con `gen.num`).
2. Usá `CASE ((num-1) % 4)` para asignar categoría.

Esto asegura **distribución uniforme exacta**, sin depender de `RAND()` ni de subqueries.

Prompt 4:

“Necesito verificar que todas las FK tienen referencias validas una vez hecha la carga masiva, serviría contar cuantas relaciones validas existen entre las tablas usando JOIN?”

ChatGPT dijo:

Exactamente  — **sí, sirve** (y es una de las formas más directas y seguras de hacerlo).

Pero te explico bien **por qué y cómo hacerlo correctamente**, porque hay algunos matices



Objetivo

Querés asegurarte de que **todas las claves foráneas creadas están correctamente referenciadas** luego de la carga masiva, es decir:

- Ningún registro en tabla hija (por ejemplo `PEDIDO_PRODUCTO`) apunta a una FK inexistente.
 - Las relaciones entre tablas realmente se cumplen (no hay “huérfanos”).
-



Enfoque correcto: comparar el total con el JOIN

La idea es comparar:

1. El total de filas de la tabla hija.
2. El total de filas que efectivamente **encuentran una coincidencia** en la tabla padre.

Por ejemplo, si `PEDIDO_PRODUCTO` tiene una FK hacia `PEDIDO`:

`SELECT`

```
(SELECT COUNT(*) FROM PEDIDO_PRODUCTO) AS total_registros,  
COUNT(p.id_pedido) AS registros_validos  
FROM PEDIDO_PRODUCTO pp  
LEFT JOIN PEDIDO p ON pp.id_pedido = p.id_pedido;
```

Si los dos números coinciden → todas las FK son válidas.

Si `registros_validos < total_registros` → hay filas huérfanas (referencias rotas).

➡ Podés repetirlo para cada FK, por ejemplo:

CLIENTE → LOCALIDADES

SELECT

```
(SELECT COUNT(*) FROM CLIENTE) AS total_clientes,  
  
COUNT(l.id_localidad) AS localidades_validas  
  
FROM CLIENTE c  
  
LEFT JOIN LOCALIDADES l ON c.id_localidad = l.id_localidad;
```

PEDIDO → CLIENTE

SELECT

```
(SELECT COUNT(*) FROM PEDIDO) AS total_pedidos,  
  
COUNT(c.id_cliente) AS clientes_validos  
  
FROM PEDIDO p  
  
LEFT JOIN CLIENTE c ON p.id_cliente = c.id_cliente;
```

PEDIDO_PRODUCTO → PEDIDO

SELECT

```
(SELECT COUNT(*) FROM PEDIDO_PRODUCTO) AS total_relaciones,  
  
COUNT(p.id_pedido) AS pedidos_validos  
  
FROM PEDIDO_PRODUCTO pp
```

```
LEFT JOIN PEDIDO p ON pp.id_pedido = p.id_pedido;
```

PEDIDO_PRODUCTO → PRODUCTO

```
SELECT
```

```
(SELECT COUNT(*) FROM PEDIDO_PRODUCTO) AS total_relaciones,  
COUNT(pr.id_producto) AS productos_validos  
FROM PEDIDO_PRODUCTO pp  
LEFT JOIN PRODUCTO pr ON pp.id_producto = pr.id_producto;
```

ENVIO → PEDIDO

```
SELECT
```

```
(SELECT COUNT(*) FROM ENVIO) AS total_envios,  
COUNT(p.id_pedido) AS pedidos_validos  
FROM ENVIO e  
LEFT JOIN PEDIDO p ON e.id_pedido = p.id_pedido;
```



Alternativa más sintética (detección de huérfanos)

Podés simplificar la lógica directamente buscando los casos inválidos:

```
SELECT pp.*
```

```

FROM PEDIDO_PRODUCTO pp

LEFT JOIN PEDIDO p ON pp.id_pedido = p.id_pedido

WHERE p.id_pedido IS NULL;
  
```

→ Si esto devuelve filas, son las **referencias inválidas**.

Conclusión

Método	Qué valida	Resultado esperado
LEFT JOIN + COUNT	Integridad referencial	Las cantidades deben coincidir
LEFT JOIN + IS NULL	Detección de huérfanos	Ninguna fila devuelta
JOIN simple	Conteo de relaciones válidas	Confirmar que hay uso real de las FK

Explicación sobre verificaciones realizadas:

VERIFICACIÓN 1: Conteo Total de Registros

Para esta primera verificación, decidimos realizar un conteo completo de todos los registros en cada tabla de la base de datos. Utilizando una consulta con UNION ALL para combinar los resultados de cada tabla en una sola salida, lo que nos permite visualizar de forma clara la distribución de datos.

Al final de la consulta, agregamos una fila que suma el total general de registros. Esto nos sirve para confirmar que alcanzamos el volumen de datos que nos propusimos cargar. La consulta recorre cada tabla (LOCALIDADES, CLIENTE, PRODUCTO, PEDIDO, PEDIDO_PRODUCTO, ENVIO) y cuenta cuántos registros hay en cada una.

Este conteo nos permite documentar el tamaño del conjunto de datos y verificar que la distribución entre las tablas sea proporcional a lo planificado.

```
SELECT 'LOCALIDADES' AS Tabla, COUNT(*) AS Registros FROM
LOCALIDADES
UNION ALL
SELECT 'CLIENTE', COUNT(*) FROM CLIENTE
UNION ALL
SELECT 'PRODUCTO', COUNT(*) FROM PRODUCTO
UNION ALL
SELECT 'PEDIDO', COUNT(*) FROM PEDIDO
UNION ALL
SELECT 'PEDIDO_PRODUCTO', COUNT(*) FROM PEDIDO_PRODUCTO
UNION ALL
SELECT 'ENVIO', COUNT(*) FROM ENVIO
UNION ALL
SELECT '==== TOTAL ===',
(SELECT SUM(cnt) FROM (
    SELECT COUNT(*) AS cnt FROM LOCALIDADES
    UNION ALL SELECT COUNT(*) FROM CLIENTE
    UNION ALL SELECT COUNT(*) FROM PRODUCTO
    UNION ALL SELECT COUNT(*) FROM PEDIDO
    UNION ALL SELECT COUNT(*) FROM PEDIDO_PRODUCTO
    UNION ALL SELECT COUNT(*) FROM ENVIO
) t);
```

VERIFICACIÓN 2: Detección de Foreign Keys Huérfanas

En esta verificación buscamos identificar si existen registros en las tablas "hijas" que refieren a registros inexistentes en las tablas "padre". Esto se conoce como validando así si encontramos alguna foreign key huérfana, lo cual representaría una violación de la integridad referencial.

Para detectar estos casos, utilizamos LEFT JOIN combinado con WHERE / IS NULL, al momento de realizar un LEFT JOIN entre la tabla hija y la tabla padre, si un registro de la tabla hija no encuentra su correspondiente en la tabla padre, el resultado del JOIN tendrá NULL en las columnas de la tabla padre.

Entonces, al filtrar con WHERE l.id_localidad IS NULL, estamos seleccionando exactamente esos casos problemáticos.

Aplicamos la misma lógica a todas las relaciones del modelo:

- Clientes que refieren localidades inexistentes
- Pedidos que refieren clientes inexistentes
- Detalles de pedido que refieren pedidos o productos inexistentes
- Envíos que refieren pedidos inexistentes

El resultado esperado para todas las consultas es 0, ya que si se encuentra, aunque sea un registro huérfano, significa que mi proceso de generación de datos tiene un error a corregir.

```
SELECT 'CLIENTE → LOCALIDADES' AS Relación, COUNT(*) AS FK_Huérfanas
FROM CLIENTE c
LEFT JOIN LOCALIDADES l ON c.id_localidad = l.id_localidad
WHERE l.id_localidad IS NULL
```

```
UNION ALL
```

```
SELECT 'PEDIDO → CLIENTE', COUNT(*)
FROM PEDIDO p
LEFT JOIN CLIENTE c ON p.id_cliente = c.id_cliente
WHERE c.id_cliente IS NULL
```

```
UNION ALL
```

```
SELECT 'PEDIDO_PRODUCTO → PEDIDO', COUNT(*)
FROM PEDIDO_PRODUCTO pp
LEFT JOIN PEDIDO p ON pp.id_pedido = p.id_pedido
```

```

WHERE p.id_pedido IS NULL

UNION ALL

SELECT 'PEDIDO_PRODUCTO → PRODUCTO', COUNT(*)
FROM PEDIDO_PRODUCTO pp
LEFT JOIN PRODUCTO pr ON pp.id_producto = pr.id_producto
WHERE pr.id_producto IS NULL

UNION ALL

SELECT 'ENVIO → PEDIDO', COUNT(*)
FROM ENVIO e
LEFT JOIN PEDIDO p ON e.id_pedido = p.id_pedido
WHERE p.id_pedido IS NULL;
  
```

VERIFICACIÓN 3: Cardinalidad - Pedidos sin Productos

Esta verificación valida que todo pedido tenga al menos un producto asociado. Un pedido sin productos carece de sentido en el contexto propuesto.

Para verificar esto, realizamos un LEFT JOIN entre la tabla PEDIDO y PEDIDO_PRODUCTO, y filtramos los casos donde no existe ningún producto asociado (WHERE pp.id_pedido IS NULL).

El resultado esperado es 0, si esta consulta devuelve algún registro, significa que se encontraron pedidos "vacíos" en la base de datos, lo cual indica un error en la lógica de generación de datos.

```

SELECT
  'Pedidos SIN productos' AS Verificación,
  COUNT(*) AS Cantidad_Incorrecta
FROM PEDIDO p
LEFT JOIN PEDIDO_PRODUCTO pp ON p.id_pedido = pp.id_pedido
WHERE pp.id_pedido IS NULL;
  
```

VERIFICACIÓN 4: Distribución de Productos por Pedido

Aquí analizamos la distribución de cuántos productos tiene cada pedido.

A fin de validar esto, primero, en la subconsulta, agrupamos por id_pedido y cuento cuántos productos tiene cada pedido, y luego en la consulta externa, agrupamos por esa cantidad y cuento cuántos pedidos tienen 1 producto, cuántos tienen 2, cuántos tienen 3, etc.

Además, calculamos el porcentaje que representa cada grupo respecto al total de pedidos. Cuando generamos los datos, buscamos que aproximadamente el 30% de los pedidos tuvieran 1 producto, el 40% tuvieran 2 productos, y el 30% restante tuvieran 3 productos. Esta verificación confirma se logró esa distribución.

```

SELECT
    productos_por_pedido AS 'Productos/Pedido',
    COUNT(*) AS Cantidad_Pedidos,
    CONCAT(ROUND(COUNT(*) * 100.0 / (SELECT COUNT(*) FROM PEDIDO),
1), '%') AS Porcentaje
FROM (
    SELECT id_pedido, COUNT(*) AS productos_por_pedido
    FROM PEDIDO_PRODUCTO
    GROUP BY id_pedido
) dist
GROUP BY productos_por_pedido
ORDER BY productos_por_pedido;
  
```

VERIFICACIÓN 5: Cardinalidad - Envíos con Estado Inválido

Esta verificación valida que solo los pedidos que estén en estado "Despachado" o "Entregado" tengan un registro de envío.

Para esto, realizamos un JOIN entre ENVIO y PEDIDO, y luego filtramos los casos donde el estado del pedido no es "Despachado" ni "Entregado". Si encuentro algún registro, significa que hay envíos asociados a pedidos que no deberían tenerlos.

El resultado esperado es 0, ya que todos los envíos deben corresponder únicamente a pedidos que fueron efectivamente despachados o entregados.

```

SELECT
    'Envíos con estado inválido' AS Verificación,
    COUNT(*) AS Cantidad_Incorrecta
FROM ENVIO e
JOIN PEDIDO p ON e.id_pedido = p.id_pedido
WHERE p.estado_pedido NOT IN ('Despachado', 'Entregado');
  
```

VERIFICACIÓN 6: Rangos de Dominio - Precios Válidos

En esta verificación controlamos que los precios de los productos estén dentro de un rango razonable que consideramos a la hora de cargar los datos, en donde definimos que los precios válidos deben estar entre \$500 y \$50,000.

Por lo que buscamos validar todos aquellos los precios sean:

1. Menor a \$500 (demasiado baratos)
2. Mayor a \$50,000 (demasiado caros)

El resultado esperado para ambas consultas es 0.

```
SELECT
    'Productos con precio < 500' AS Verificación,
    COUNT(*) AS Cantidad
FROM PRODUCTO
WHERE precio_unitario < 500

UNION ALL

SELECT
    'Productos con precio > 50000',
    COUNT(*)
FROM PRODUCTO
WHERE precio_unitario > 50000;
```

VERIFICACIÓN 7: Integridad Matemática - Totales de Pedidos

Lo que buscamos validar con esta verificación, es que cada pedido tiene un campo total que almacena el monto total del pedido. Este total debe ser exactamente igual a la suma de todos los subtotales de los productos de ese pedido en la tabla PEDIDO_PRODUCTO.

La consulta calcula la suma de los subtotales para cada pedido y la compara con el total almacenado. Utilizamos ABS() para obtener el valor absoluto de la diferencia, considerando como aceptable una diferencia de hasta 0.01 para contemplar posibles errores de redondeo por el uso de decimales.

Si se encontraran pedidos donde la diferencia es mayor a 0.01, significa que hay una inconsistencia entre el total almacenado con la suma real de los productos.

El resultado esperado es 0 pedidos con totales incorrectos.

```

SELECT
  'Pedidos con total incorrecto' AS Verificación,
  COUNT(*) AS Cantidad_Incorrecta
FROM PEDIDO p
WHERE ABS(p.total - COALESCE((
  SELECT SUM(pp.subtotal)
  FROM PEDIDO_PRODUCTO pp
  WHERE pp.id_pedido = p.id_pedido
), 0)) > 0.01;
  
```

VERIFICACIÓN 8: Integridad Temporal - Fechas Coherentes de Envío

En esta última verificación se buscó validar la lógica de los envíos, donde la fecha_entrega siempre debe ser igual o posterior a la fecha_despacho.

La consulta busca registros en la tabla ENVIO donde fecha_entrega < fecha_despacho.

Si esta consulta devuelve algún registro, significa que hay datos inconsistentes en las fechas.

El resultado esperado es 0 envíos con fechas incoherentes.

```

SELECT
  'Envíos con fecha_entrega < fecha_despacho' AS Verificación,
  COUNT(*) AS Cantidad_Incorrecta
FROM ENVIO
WHERE fecha_entrega < fecha_despacho;
  
```

VERIFICACIÓN 9: Integridad Referencial Post-Carga

Esta verificación valida que todas las Foreign Keys apunten a registros existentes después de la carga masiva de datos. Es fundamental para garantizar la consistencia de las relaciones entre tablas.

Para cada relación FK del sistema, se compara el total de registros de la tabla hija contra las referencias válidas encontradas mediante un LEFT JOIN con la tabla padre. La diferencia entre ambos valores indica la cantidad de registros "huérfanos" (FK que apuntan a IDs inexistentes).

```

SELECT
    'CLIENTE → LOCALIDADES' AS relacion,
    (SELECT COUNT(*) FROM CLIENTE) AS total_registros,
    COUNT(l.id_localidad) AS referencias_validas,
    (SELECT COUNT(*) FROM CLIENTE) - COUNT(l.id_localidad) AS
huerfanas,
CASE
    WHEN COUNT(l.id_localidad) = (SELECT COUNT(*) FROM
CLIENTE)
        THEN 'OK'
        ELSE 'REVISAR'
    END AS estado
FROM CLIENTE c
LEFT JOIN LOCALIDADES l ON c.id_localidad = l.id_localidad

UNION ALL
...
  
```

Reflexión sobre el uso de IA en la Etapa 2:

Para la carga masiva de datos y las pruebas de consistencia, necesitábamos generar un volumen significativo de información realista por lo que el desafío principal fue crear datos en gran volumen que cumplieran con las constraints, y que se simulara una lógica similar a la que veríamos en un e-commerce.

Consultamos a ChatGPT principalmente para validar estrategias de generación de datos en SQL, la IA nos orientó sobre cómo estructurar el script de generación, pero nosotros definimos la lógica de negocio, como por ejemplo cantidad de productos por pedido (1-3 productos con distribución 30%-40%-30%), y rangos de precios a popular (\$500-\$50,000).

Principalmente la ventaja que obtuvimos sobre el uso de la IA generativa fue el comprender cómo generar de manera masiva datos consistentes que nos sirvan para ejecutar futuras pruebas sin la necesidad de realizar las mismas cargas de manera manual línea por línea.

Sobre las pruebas de consistencia, la IA nos ayudó a entender diferentes enfoques para validar la integridad referencial. Como ejemplo de esto nos sugirió usar LEFT JOIN para detectar FK huérfanas. Sin embargo, decidimos implementar una verificación más completa que mostrará un resumen con contadores de registros válidos vs. huérfanos, ya que consideramos que esto facilita la interpretación de resultados durante la evaluación.

Hubo sugerencias de la IA sobre validaciones adicionales (como verificar unicidad de emails de clientes o números de seguimiento duplicados), pero al igual que en la Etapa 1, priorizamos mantener un conjunto de pruebas simplificado. Nos enfocamos en nueve verificaciones clave que cubren: conteos totales, integridad referencial, cardinalidades obligatorias, rangos de dominio, y asegurar coherencia de cálculos y fechas.

Para la documentación de cardinalidades, consultamos cómo validar que solo los pedidos en estado "Despachado" o "Entregado" tuvieran envíos asociados. La IA nos mostró la sintaxis SQL correcta, pero nosotros definimos qué estados eran válidos según nuestra lógica de negocio modelada en la Etapa 1.

Un punto importante fue la validación de totales de pedidos. Consultamos cómo manejar diferencias por redondeo decimal en la comparación entre el campo total y la suma de subtotales. La IA nos explicó que usar $\text{ABS}(\text{diferencia}) < 0.01$ era más robusto que una comparación exacta, evitando falsos positivos por precisión de punto flotante.

Para verificar que todas las pruebas fueran correctas, ejecutamos el script completo múltiples veces después de cargar los datos. Todas las verificaciones que esperábamos devolver 0 efectivamente lo hicieron, y la distribución de productos por pedido coincidió con los porcentajes planificados, confirmando que tanto la generación como las validaciones funcionan correctamente.

En conclusión, la IA fue una herramienta valiosa para validar sintaxis de SQL y explicar técnicas de validación de datos.

Evidencia de uso de IA:

<https://chatgpt.com/share/690516d8-4354-800a-bb58-774000a4d183>

Etapa 3:

Diseño de Consultas con Utilidad Realista

Consulta 1: Top 5 productos más vendidos.

```
SELECT
    p.producto_nombre,
    SUM(pp.cantidad) AS total_vendido
FROM
    PRODUCTO p
JOIN
    PEDIDO_PRODUCTO pp ON p.id_producto = pp.id_producto
GROUP BY
    p.producto_nombre
ORDER BY
    total_vendido DESC
LIMIT 5;
```

Utilidad de negocio: Esta consulta es fundamental para la gestión de inventario y estrategias de marketing. Permite identificar los productos "estrella" para asegurar su stock, destacarlos en campañas publicitarias y analizar qué características los hacen tan populares para replicar su éxito en otros artículos.

Sentencia SQL: Usando las tablas `PEDIDO_PRODUCTO` que tiene la cantidad vendida de cada producto en cada pedido y `PRODUCTO`, que nos permite obtener el `producto_nombre`, agregamos todas las cantidades de la tabla `pedido_producto` y seguido de una cláusula GROUP BY para asegurar que la suma se haga por separado para cada `id_producto`. Por último le agregamos `p.producto_nombre` para que la base de datos sepa cómo agrupar las sumas. Luego, las ordenamos y limitamos al número deseado, en este caso 5.-

Consulta 2: Clientes Fieles (con +10 pedidos).

```

SELECT
    C.cliente_nombre,
    COUNT(p.id_pedido) AS numero_de_pedidos
FROM
    CLIENTE c
JOIN
    PEDIDO p ON c.id_cliente = p.id_cliente
GROUP BY
    c.cliente_nombre
HAVING
    COUNT(p.id_pedido) >= 10
ORDER BY
    numero_de_pedidos DESC;
  
```

Utilidad de negocio: Esta consulta identifica al segmento de clientes más leales. Con esta lista, el equipo de marketing puede lanzar campañas de fidelización, ofrecer descuentos exclusivos, accesos anticipados a productos o crear un programa de recompensas para agradecer y retener a su clientela más valiosa.

Sentencia SQL: Usando la tabla **PEDIDO** agrupando por **id_cliente**, unimos la tabla **PEDIDO** con **CLIENTE** para poder mostrar el nombre del cliente en lugar de solo su ID. Luego haciendo uso de **HAVING**:

- Agrupamos todos los pedidos por cliente y contamos cuántos hay en cada grupo usando **GROUP BY** y **COUNT**.
- Filtramos esos grupos para quedarnos solo con los que tengan un conteo de 10 o más usando **HAVING**.

Consulta 3: Top 5 Clientes con más dinero invertido en la tienda.

```
SELECT
    C.id_cliente,
    C.cliente_nombre,
    SUM(p.total) AS gasto_total
FROM
    CLIENTE c
JOIN
    PEDIDO p ON c.id_cliente = p.id_cliente
GROUP BY
    c.id_cliente, c.cliente_nombre
ORDER BY
    gasto_total DESC
LIMIT 5;
```

Utilidad de negocio: Esta es una consulta de alto valor estratégico. Identifica a los clientes "VIP" por su gasto. Esta información es oro para personalizar ofertas y para fortalecer la relación con los clientes más rentables.

Sentencia SQL: Como necesitamos el gasto total por cliente, la mejor forma de obtenerlo es uniendo `CLIENTE` y `PEDIDO`, sumando la columna total. Luego `GROUP BY` descendiente y limitamos a 5.

Consulta 4: Provincias con Ventas > \$100k y su producto estrella.

```
-- CTE 1: Calcula el total vendido por provincia y filtra las que superan
los $100,000.
WITH VentasPorProvincia AS (
  SELECT
    l.provincia,
    SUM(p.total) AS total_ventas
  FROM
    LOCALIDADES l
  JOIN
    CLIENTE c ON l.id_localidad = c.id_localidad
  JOIN
    PEDIDO p ON c.id_cliente = p.id_cliente
  GROUP BY
    l.provincia
  HAVING
    SUM(p.total) > 100000
),

-- CTE 2: Ranking de los productos más vendidos dentro de cada provincia.
RankingProductosProvincia AS (
  SELECT
    l.provincia,
    pr.producto_nombre,
    ROW_NUMBER() OVER(PARTITION BY l.provincia ORDER BY
SUM(pp.cantidad) DESC) as ranking
  FROM
    LOCALIDADES l
  JOIN
    CLIENTE c ON l.id_localidad = c.id_localidad
  JOIN
    PEDIDO p ON c.id_cliente = p.id_cliente
  JOIN
    PEDIDO_PRODUCTO pp ON p.id_pedido = pp.id_pedido
  JOIN
    PRODUCTO pr ON pp.id_producto = pr.id_producto
  GROUP BY
    l.provincia, pr.producto_nombre
)
-- Consulta Final: Une las provincias con altas ventas con su producto #1.
SELECT
  vpp.provincia,
  vpp.total_ventas,
  rpp.producto_nombre AS producto_mas_vendido
FROM
  VentasPorProvincia vpp
JOIN
  RankingProductosProvincia rpp ON vpp.provincia = rpp.provincia
WHERE
  rpp.ranking = 1
ORDER BY
  vpp.total_ventas DESC;
```

Utilidad de negocio: Esta consulta es clave para la logística y la estrategia de marketing regional. Permite identificar los mercados provinciales más importantes (Ventas > \$100k) y entender la demanda específica de cada uno (su producto_mas_vendido). Con estos datos, se pueden optimizar las campañas publicitarias por región, ajustar el stock en los centros de distribución locales y tomar decisiones informadas sobre expansiones.

Sentencia SQL: Identifica las provincias con mayores ventas totales y dentro de cada una de ellas muestra cuál fue el producto más vendido. Para lograrlo se utilizan **CTEs** nuevamente, que son las tablas temporales definidas dentro de la misma consulta.

En la 1era **CTE**, llamada **VentasPorProvincia**, se calcula el total de ventas agrupado por provincia. Se realiza la unión entre las tablas **LOCALIDADES**, **CLIENTE** y **PEDIDO**, vinculando las claves foráneas correspondientes: **id_localidad** y **id_cliente**. Luego mediante la función **SUM(p, total)** se obtiene la suma del total vendido en cada provincia. Luego se agrupan los resultados y se filtran con **HAVING**, aquellas que tengan más de 100k. En la 2da **CTE**, llamada **RankingProductosProvincia**, se genera un ranking de los productos más vendidos en cada provincia, según la cantidad total comprada. Para eso se enlazan las tablas **LOCALIDADES**, **CLIENTE**, **PEDIDO**, **PEDIDO_PRODUCTO** y **PRODUCTO**, de modo que se pueda acceder tanto a la información geográfica como a los detalles de cada producto vendido. Se agrupan los resultados y se les asigna un rankin. Por último, la **consulta final**, combina las dos **CTE** anteriores. Se realiza un **JOIN** entre **VentasPorProvincia** y **RankingProductosProvincia** mediante el campo **provincia** para obtener, de cada provincia con ventas superiores a \$100k el producto que ocupa el 1er lugar del rankin. Se filtra el resultado y se ordena la salida de forma descendiente.

Creación de la Vista

Elección de vista:

Para la elección de la vista, buscamos una que combine `PEDIDO`, `ENVIO` y `PEDIDO_PRODUCTO`, lo cual es increíblemente útil. Esto permite a un operador o a un analista ver rápidamente toda la información relevante de un envío sin tener que escribir `JOIN` complejo cada vez. Básicamente, crea un “reporte” pre-construido.

La vista se llama `vista_Detalle_Envio`, y muestra para cada producto dentro de cada envío, la información del envío, del pedido y del cliente.

```

CREATE OR REPLACE VIEW Vista_Detalle_Envios AS
SELECT
    e.id_envio,
    e.numero_seguimiento,
    e.estado_envio,
    e.fecha_despacho,
    e.fecha_entrega,
    p.id_pedido,
    p.fecha_pedido,
    c.cliente_nombre,
    pr.producto_nombre,
    pp.cantidad,
    pp.subtotal
FROM
    ENVIO e
JOIN
    PEDIDO p ON e.id_pedido = p.id_pedido
JOIN
    CLIENTE c ON p.id_cliente = c.id_cliente
JOIN
    PEDIDO_PRODUCTO pp ON p.id_pedido = pp.id_pedido
JOIN
    PRODUCTO pr ON pp.id_producto = pr.id_producto
ORDER BY
    e.fecha_despacho DESC, e.id_envio, pr.producto_nombre;
  
```

Para ver la vista ejecutamos la siguiente sentencia:

```
SELECT * FROM Vista_Detalle_Envios WHERE id_envio = 1234;
```

Esta vista, llamada `Vista_Detalle_Envios`, reúne en una sola consulta toda la información relacionada con los envíos realizados.

Combina datos de varias tablas `ENVIO`, `PEDIDO`, `CLIENTE`, `PEDIDO_PRODUCTO` y `PRODUCTO` para mostrar, en un único resultado, los detalles completos de cada envío.

Incluye información del envío (ID, número de seguimiento, estado, fechas de despacho y entrega), del pedido asociado (ID y fecha), del cliente (nombre), y de los productos enviados (nombre, cantidad y subtotal).

La cláusula `ORDER BY e.fecha_despacho DESC` ordena los resultados mostrando primero los envíos más recientes.

En resumen, esta vista facilita consultar el historial completo de envíos con todos los datos relevantes del cliente, pedido y productos en una sola estructura.

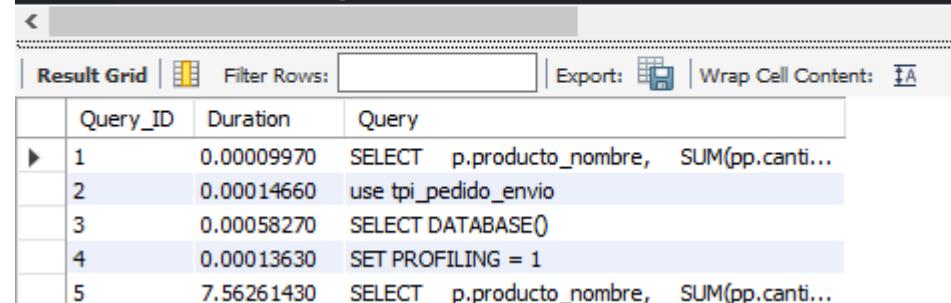
Mediciones de Tiempos de las Consultas

1era Consulta:

```

1 *   use tpi_pedido_envio;
2 *   SET PROFILING = 1;
3 *   SELECT
4       p.producto_nombre,
5           SUM(pp.cantidad) AS total_vendido
6   FROM
7       PRODUCTO p
8   JOIN
9       PEDIDO_PRODUCTO pp ON p.id_producto = pp.id_producto
10  GROUP BY
11      p.producto_nombre
12  ORDER BY
13      total_vendido DESC
14  LIMIT 5;
15 *   SHOW PROFILES;

```



The screenshot shows the MySQL Workbench interface with the results of the SHOW PROFILES command. The results are displayed in a grid with columns: Query_ID, Duration, and Query.

	Query_ID	Duration	Query
▶	1	0.00009970	SELECT p.producto_nombre, SUM(pp.canti...
	2	0.00014660	use tpi_pedido_envio
	3	0.00058270	SELECT DATABASE()
	4	0.00013630	SET PROFILING = 1
	5	7.56261430	SELECT p.producto_nombre, SUM(pp.canti...

Usando PROFILING.

Tiempos medidos:

- 7.562 sec
- 2.250 sec
- 0.734 sec

2da Consulta:

```

3 *   SELECT
4     C.cliente_nombre,
5     COUNT(p.id_pedido) AS numero_de_pedidos
6   FROM
7     CLIENTE c
8   JOIN
9     PEDIDO p ON c.id_cliente = p.id_cliente
10  GROUP BY
11    c.cliente_nombre
12  HAVING
13    COUNT(p.id_pedido) >= 10
14  ORDER BY
15    numero_de_pedidos DESC;
16 *   SHOW PROFILES;

```

Result Grid | Filter Rows: Export: Wrap Cell Content:

	Query_ID	Duration	Query
▶	1	0.00009970	SELECT p.producto_nombre, SUM(pp.canti...
	2	0.00014660	use tpi_pedido_envio
	3	0.00058270	SELECT DATABASE()
	4	0.00013630	SET PROFILING = 1
	5	7.56261430	SELECT p.producto_nombre, SUM(pp.canti...
	6	0.00017800	SET PROFILING = 1
	7	2.24993690	SELECT p.producto_nombre, SUM(pp.canti...
	8	0.00010580	SET PROFILING = 1
	9	0.73385390	SELECT p.producto_nombre, SUM(pp.canti...
	10	0.00014980	SET PROFILING = 1
	11	0.32530630	SELECT C.cliente_nombre, COUNT(p.id_pedido)...

Usando PROFILING

Tiempos Medidos:

- 0.325 sec
- 0.219 sec
- 0.212 sec

3er Consulta:

```

1 *  use tpi_pedido_envio;
2 *  SET PROFILING = 1;
3 *  SELECT
4   C.id_cliente,
5   C.cliente_nombre,
6   SUM(p.total) AS gasto_total
7   FROM
8   CLIENTE c
9   JOIN
10  PEDIDO p ON c.id_cliente = p.id_cliente
11  GROUP BY
12  c.id_cliente, c.cliente_nombre
13  ORDER BY
14  gasto_total DESC
15  LIMIT 5;
16 *  SHOW PROFILES;

```

	19	0.00061230	SET PROFILING = 1
	20	4.81834980	SELECT C.id_cliente, C.cliente_nombre, SUM(p....

Usando PROFILING

Tiempos Medidos:

- **4.818 sec** ²⁰ 4.81834980 SELECT C.id_cliente, C.cliente_nombre, SU...
- **0.726 sec** ²² 0.72616850 SELECT C.id_cliente, C.cliente_nombre, SUM(p....
- **0.367 sec** ²⁴ 0.36770570 SELECT C.id_cliente, C.cliente_nombre, SU...

4ta Consulta:

```

1   -- CTE 1: Calcula el total vendido por provincia y filtra las que superan los $100,000.
2 * use tpi_pedido_envio;
3 * SET PROFILING = 1;
4 * WITH VentasPorProvincia AS (
5     SELECT
6         l.provincia,
7         SUM(p.total) AS total_ventas
8     FROM
9         LOCALIDADES l
10    JOIN
11        CLIENTE c ON l.id_localidad = c.id_localidad
12    JOIN
13        PEDIDO p ON c.id_cliente = p.id_cliente
14    GROUP BY
15        l.provincia
16    HAVING
17        SUM(p.total) > 100000
18 ),
19
20 * RankingProductosProvincia AS (
21     SELECT
22         l.provincia,
23         pr.producto_nombre,
24         ROW_NUMBER() OVER(PARTITION BY l.provincia ORDER BY SUM(pp.cantidad) DESC) as ranking
25     FROM
26         LOCALIDADES l
27    JOIN
28        CLIENTE c ON l.id_localidad = c.id_localidad
29    JOIN
30        PEDIDO p ON c.id_cliente = p.id_cliente
31    JOIN
32        PEDIDO_PRODUCTO pp ON p.id_pedido = pp.id_pedido
33    JOIN
34        PRODUCTO pr ON pp.id_producto = pr.id_producto
35    GROUP BY
36        l.provincia, pr.producto_nombre
37 )
38
39 -- Consulta Final: Une las provincias con altas ventas con su producto #1.
40 SELECT
41     vpp.provincia,
42     vpp.total_ventas,
43     rpp.producto_nombre AS producto_mas_vendido
44 FROM
45     VentasPorProvincia vpp
46 JOIN
47     RankingProductosProvincia rpp ON vpp.provincia = rpp.provincia
48 WHERE
49     rpp.ranking = 1
50 ORDER BY
51     vpp.total_ventas DESC;
52 * SHOW PROFILE;

```

Tiempos Medidos:

- **20.803 sec** 30 20.80382020 WITH VentasPorProvincia AS (SELECT ...)
- **17.415 sec** 32 17.41539130 WITH VentasPorProvincia AS (SELEC...)
- **15.879 sec** 34 15.87950570 WITH VentasPorProvincia AS (SELEC...)

Medición de Tiempo de las Consultas CON Índice

1er Consulta:

```

1 *  use tpi_pedido_envio;
2 *  SET PROFILING = 1;
3   SELECT sql_no_cache
4     p.producto_nombre,
5       SUM(pp.cantidad) AS total_vendido
6   FROM
7     PRODUCTO p
8   JOIN
9     PEDIDO_PRODUCTO pp ON p.id_producto = pp.id_producto
10  GROUP BY
11    p.producto_nombre
12  ORDER BY
13    total_vendido DESC
14  LIMIT 5;
15 *  SHOW PROFILES;
```

Tiempos medidos CON Índice:

- **0.5306 sec** |62 0.53062850 SELECT sql_no_cache p.producto_nombre, ...
- **0.5034 sec** 64 0.50343840 SELECT sql_no_cache p.producto_nombre, ...
- **0.5395 sec** 65 0.53951110 SELECT sql_no_cache p.producto_nombre, ...

Comando EXPLAIN ANALYZE:

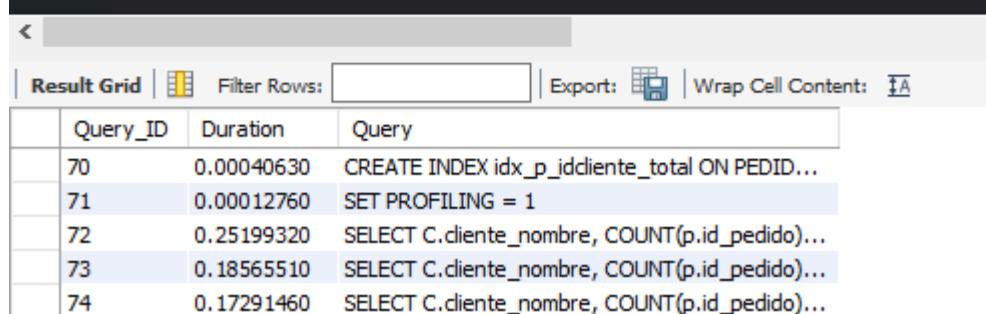
```
1, SIMPLE, pp, ref, idx_pp_idproducto, idx_pp_idproducto, 4,
tpi_pedido_envio.p.id_producto, 19, 40.00, 100.00, 100.00,
```

2da Consulta:

```

2 •   SET PROFILING = 1;
3 •   SELECT
4     C.cliente_nombre,
5     COUNT(p.id_pedido) AS numero_de_pedidos
6   FROM
7     CLIENTE c
8   JOIN
9     PEDIDO p ON c.id_cliente = p.id_cliente
10  GROUP BY
11    c.cliente_nombre
12  HAVING
13    COUNT(p.id_pedido) >= 10
14  ORDER BY
15    numero_de_pedidos DESC;
16 •   SHOW PROFILES;

```



	Query_ID	Duration	Query
	70	0.00040630	CREATE INDEX idx_p_idcliente_total ON PEDID...
	71	0.00012760	SET PROFILING = 1
	72	0.25199320	SELECT C.cliente_nombre, COUNT(p.id_pedido)...
	73	0.18565510	SELECT C.cliente_nombre, COUNT(p.id_pedido)...
	74	0.17291460	SELECT C.cliente_nombre, COUNT(p.id_pedido)...

Tiempos medidos CON Índice:

- **0.25199 sec** [72](#) 0.25199320 SELECT C.cliente_nombre, COUNT(p.id_pedido)...
- **0.18565 sec** [73](#) 0.18565510 SELECT C.cliente_nombre, COUNT(p.id_pedido)...
- **0.17291 sec** [74](#) 0.17291460 SELECT C.cliente_nombre, COUNT(p.id_pedido)...

Comando ANALYZE EXPLAIN:

```
'1', 'SIMPLE', 'p', 'ref', 'idx_p_idcliente, idx_p_idcliente_total',
'idx_p_idcliente', '4', 'tpi_pedido_envio.c.id_cliente', '1',
'3.33', '100.00', '100.00', 'Using index'
```

3er Consulta:

```

4  SELECT
5    C.id_cliente,
6    C.cliente_nombre,
7    SUM(p.total) AS gasto_total
8  FROM
9    CLIENTE c
10   JOIN
11    PEDIDO p ON c.id_cliente = p.id_cliente
12   GROUP BY
13    c.id_cliente, c.cliente_nombre
14   ORDER BY
15    gasto_total DESC
16   LIMIT 5;
17 * SHOW PROFILES;

```

Result Grid | Filter Rows: Export: Wrap Cell Content:

	Query_ID	Duration	Query
	76	0.00018870	SET PROFILING = 1
	77	0.20448530	SELECT C.id_cliente, C.cliente_nombre, SUM(p....
	78	0.20237740	SELECT C.id_cliente, C.cliente_nombre, SUM(p....
	79	0.20646020	SELECT C.id_cliente, C.cliente_nombre, SUM(p....
	80	0.22822730	ANALYZE SELECT C.id_cliente, C.cliente_nombr...

Tiempos medidas CON Índice:

- **0.204485 sec** [77](#) 0.20448530 SELECT C.id_cliente, C.cliente_nombre, SUM(p....
- **0.202377 sec** [78](#) 0.20237740 SELECT C.id_cliente, C.cliente_nombre, SUM(p....
- **0.206460 sec** [79](#) 0.20646020 SELECT C.id_cliente, C.cliente_nombre, SUM(p....

Comando EXPLAIN ANALYZE:

```
'1', 'SIMPLE', 'p', 'ref', 'idx_p_idcliente, idx_p_idcliente_total',
'idx_p_idcliente_total', '4', 'tpi_pedido_envio.c.id_cliente', '1',
'3.33', '100.00', '100.00', 'Using index'
```

4ta Consulta:

```

3 *   SET PROFILING = 1;
4 * WITH VentasPorProvincia AS (
5     SELECT
6         l.provincia,
7             SUM(p.total) AS total_ventas
8     FROM
9         LOCALIDADES l
10    JOIN
11        CLIENTE c ON l.id_localidad = c.id_localidad
12    JOIN
13        PEDIDO p ON c.id_cliente = p.id_cliente
14    GROUP BY
15        l.provincia
16    HAVING
17        SUM(p.total) > 100000
18    ),
19
20 * RankingProductosProvincia AS (
21     SELECT
22         l.provincia,
23             pr.producto_nombre,
24             ROW_NUMBER() OVER(PARTITION BY l.provincia ORDER BY SUM(pp.cantidad) DESC) as ranking
25     FROM
26         LOCALIDADES l
27    JOIN
28        CLIENTE c ON l.id_localidad = c.id_localidad
29    JOIN
30        PEDIDO p ON c.id_cliente = p.id_cliente
31    JOIN
32        PEDIDO_PRODUCTO pp ON p.id_pedido = pp.id_pedido
33    JOIN
34        PRODUCTO pr ON pp.id_producto = pr.id_producto
35    GROUP BY
36        l.provincia, pr.producto_nombre
37    )
38
39 -- Consulta Final: Une las provincias con altas ventas con su producto #1.
40 SELECT
41     vpp.provincia,
42     vpp.total_ventas,
43     rpp.producto_nombre AS producto_mas_vendido
44 FROM
45     VentasPorProvincia vpp
46 JOIN
47     RankingProductosProvincia rpp ON vpp.provincia = rpp.provincia
48 WHERE
49     rpp.ranking = 1
50 ORDER BY
51     vpp.total_ventas DESC;
52 * SHOW PROFILE;
```

Result Grid		
	Query_ID	Duration
	82	0.00048150
	83	0.00011000
	84	17.11825980
	85	13.97455020
	86	12.24619060

Query

Tiempos medidos CON Índice:

- **17.11825 sec** ⁸⁴ 17.11825980 WITH VentasPorProvincia AS (SELECT ...
- **13.97455 sec** ⁸⁵ 13.97455020 WITH VentasPorProvincia AS (SELECT ...
- **12.24619 sec** ⁸⁶ 12.24619060 WITH VentasPorProvincia AS (SELECT ...

Comando EXPLAIN ANALYZE SELECT:

Al ser una consulta larga, diferentes partes tiene.

```

→ '2', 'DERIVED', 'p', 'ref',
  'idx_p_idcliente, idx_p_idcliente_total',
  'idx_p_idcliente_total', '4', 'tpi_pedido_envio.c.id_cliente',
  '1', '3.33', '100.00', '100.00', 'Using index'
→ '2', 'DERIVED', 'l', 'index', 'PRIMARY, idx_loc_provincia',
  'idx_loc_provincia', '202', NULL, '48', '48.00', '100.00',
  '100.00', 'Using index; Using temporary; Using filesort'
→ '3', 'DERIVED', 'p', 'ref',
  'PRIMARY, idx_p_idcliente, idx_p_idcliente_total',
  'idx_p_idcliente', '4', 'tpi_pedido_envio.c.id_cliente', '1',
  '3.33', '100.00', '100.00', 'Using index'
→ '3', 'DERIVED', 'pp', 'ref',
  'idx_pp_idproducto, idx_pp_idpedido_idproducto',
  'idx_pp_idpedido_idproducto', '4',
  'tpi_pedido_envio.p.id_pedido', '1', '2.00', '100.00',
  '100.00', ''

```

Conclusión del estudio:

Al crear las diferentes mediciones, logramos comprender el uso de índices para optimizar las consultas.

Las diferentes mediciones vieron una mejora dependiendo del tipo de metodología utilizada para hacer la búsqueda.

Con los comandos **ANALYZE SELECT** nos podemos dar cuenta de lo que está haciendo la consulta dentro del motor de InnoDB de MaríaDB y medir los tiempos específicos del motor con **PROFILING**.

La IA nos ayudó a entender el funcionamiento del **SET PROFILING** para medir los tiempos con SQL Workbench y además fue super útil para recomendar y proponer índices, así como para hacer un seguimiento al testeo, asegurándonos así, la correlación de las consultas y el orden correcto para ejecutar las sentencias.

```
y si por ejemplo utilizo SET @t1 = now(6); ? luego otro t stamp
después y por ultimo miro la diferencia? es mejor ese método o el
profiling nomas?
```

Con este prompt propusimos una metodología, la cual la IA no nos recomendó, explicando la diferencia y las ventajas científicas de usar el **PROFILING**.

Buena pregunta – ambos métodos **sirven**, pero **no miden lo mismo**.

 **Método recomendado para tu TP: SET profiling = 1**

-  Es **nativo de MySQL**
-  Mide **solo tiempo real del motor de ejecución**, no latencia
-  La cátedra suele esperar este, porque es estándar para evaluación de optimización

 **Conclusión recomendada:** usá **profiling** como método principal.

Si querés, podés sumar el método `NOW(6)` solo como apoyo final en el informe, no como único método.

Por último adjunto el prompt que dio inicio a la etapa de creación de índices y pruebas con el comando ANALYZE SELECT de mariaDB.-

excelente bueno. ahora tenemos que pedir las 4 mismas consultas, pero despues de crear indices. tengo que crear esos indices. te cuento que estas son las 4 consultas con mediciones, de show profiles que realice. me podes orientar sobre que indices son los que mas sirven?

RTA:

¡Genial! Vamos a elegir **indices "con sentido"** para tus 4 consultas, sin darte una "lista mágica" cerrada. La idea es que vos mismo/a confirmes cada decisión con `EXPLAIN / EXPLAIN ANALYZE` y el `SHOW PROFILES` que ya usaste.

Voy consulta por consulta, con **pistas, motivos y un ejemplo de índice candidato** (vos validás y ajustás) :

El resto del chat está adjunto aquí:

<https://chatgpt.com/share/69026560-b3bc-8011-b99d-d352ad0a306f>

Etapa 4:

Creación de usuario con permisos acotados (principio de mínimos privilegios)

Definimos dos usuarios a modo de evidenciar la regla de usuario con mínimos privilegios y otro con privilegios de root.

En este trabajo nos reservamos el uso de los usuarios root por defecto como una buena práctica, para evitar errores a la hora de otorgar y quitar privilegios.

Los usuarios creados son:

- **user_ventas**: empleados en el área de venta, con privilegios mínimos para ver información pública y realizar pedidos.
- **admin_bd**: usuario que se encargaría del manejo y administración de la base de datos, para su mantenimiento y en caso de eliminaciones o errores, poder acudir a ella. Sus privilegios son de root.

El usuario '**user_ventas**' pertenece a un empleado activo con una contraseña propia, sus privilegios mínimos son ver vistas públicas y registrar pedidos, no debería acceder a información sensible.

Una medida de seguridad es otorgar permisos sobre vista y no sobre las tablas bases, ej: **vista_cliente_publico**.

El **INSERT** en **PEDIDO** y **PEDIDO_PRODUCTO** permite registrar ventas sin exponer datos sensibles ni dar poder de borrado/alteración.

```
-- Crear usuario con minimos privilegios
CREATE USER 'user_ventas'@'localhost' IDENTIFIED BY '1234';

-- Quitar cualquier privilegio por defecto (pro medida de seguridad)
-- REVOKE ALL PRIVILEGES, GRANT OPTION FROM 'user_ventas'@'localhost';

-- Otorgar privilegios necesarios:
-- SELECT sobre las vistas (que crearemos abajo)
-- INSERT sobre PEDIDO y PEDIDO_PRODUCTO para poder crear pedidos
GRANT SELECT ON tpi_pedido_envio.vista_clientes_publicos TO 'user_ventas'@'localhost';
GRANT SELECT ON tpi_pedido_envio.vista_pedidos_resumen TO 'user_ventas'@'localhost';

GRANT INSERT ON tpi_pedido_envio.PEDIDO TO 'user_ventas'@'localhost';
GRANT INSERT ON tpi_pedido_envio.PEDIDO_PRODUCTO TO 'user_ventas'@'localhost';

-- No damos UPDATE/DELETE/ALTER/DROP ni acceso directo a tablas sensibles.
FLUSH PRIVILEGES;
```

El usuario '**admin_db**', su función es la manipulación, administración de los datos y el poder de otorgar a otros usuario privilegios.

```

-- Una forma menos recomendadas es que tenga los permisos sobre las tablas y columnas de la DB.
CREATE USER 'admin_db'@'localhost' IDENTIFIED BY '0000'; -- Creacion del usuario admin_db (USAGE = sin privilegios
GRANT ALL privileges ON *.* TO 'admin_db'@'localhost' WITH GRANT OPTION;
-- WITH GRANT OPTION permite que este usuario otorgue privilegios a otros
-- Tener en cuenta que los privilegios solo son aplicables a tpi_pedido_envio
FLUSH PRIVILEGES; -- aplica los cambios inmediatamente

```

En la siguiente imagen se evidencia la vista de una tabla compuesta por todos los usuarios existentes, es de interés observar **user_ventas** el cual muestra en su fila que no tiene privilegios sobre las tablas.

En cambio el otro usuario creado, '**admin_db**' cuenta con script que le da privilegios de root, lo recomendable no es ocupar este último para evitar errores a la hora de otorgar privilegios.

```

1      -- Nivel READ COMMITTED
2 •  SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
3
4 •  START transaction;
5 •  SELECT total FROM PEDIDO WHERE id_pedido = 1;
6      -- otra sesion actualizada total:
7      -- UPDATE PEDIDO SET totla = total + 50 WHERE id_pedido = 1
8 •  COMMIT;

```

Result Grid		Filter Rows:	Export:	Wrap Cell Content:
	total			
▶	638889.60			

Host	User	Select_priv	Insert_priv	Update_priv	Delete_priv	Create_priv	Drop_priv	Reload_priv	Shutdown_priv	Process_priv	File_priv	G
localhost	admin_db	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
localhost	mysql.infoschema	Y	N	N	N	N	N	N	N	N	N	N
localhost	mysql.session	N	N	N	N	N	N	Y	N	N	N	N
localhost	mysql.sys	N	N	N	N	N	N	N	N	N	N	N
localhost	root	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
▶ localhost	user_ventas	N	N	N	N	N	N	N	N	N	N	N

Referencias: anexo, Ref 1d

Vistas que ocultan información sensible

Con el siguiente script, pudimos crear una vista sin los email ni las direcciones de los clientes, lo cual es considerado información sensible.

```
DROP VIEW IF EXISTS tpi_pedido_envio.vista_clientes_publicos;
CREATE SQL SECURITY DEFINER VIEW tpi_pedido_envio.vista_clientes_publicos AS
SELECT
  id_cliente,
  cliente_nombre,
  cliente_telefono,
  id_localidad
FROM tpi_pedido_envio.CLIENTE;
```

Obteniendo una vista sesgada de la tabla **tpi_pedido_envio**. Lo importante es que protegemos datos sensibles de los clientes.

	id_cliente	cliente_nombre	cliente_telefono	id_localidad
▶	1	Juan Pérez	11-4567-8900	1
	2	Juan Moreno 90	11-2302-5767	21
	3	Juan Moreno 91	11-8286-7892	25
	4	Juan Moreno 92	11-5934-3067	33
	5	Juan Moreno 93	11-5474-5034	31
	6	Juan Moreno 94	11-9046-5925	1
	7	Juan Moreno 95	11-1771-4761	25
	8	Juan Moreno 96	11-4947-6773	20
	9	Juan Moreno 97	11-1322-9562	45
	10	Juan Moreno 98	11-6313-2409	38
	11	Juan Moreno 99	11-2274-4033	11

vista_clientes_publicos 4 ×

En este caso el script que ejecuta crea una vista, llamada “**vista_pedidos_resumen**”, la cual está sesgada sin las direcciones de entrega ni datos de seguimiento.

```
DROP VIEW IF EXISTS tpi_pedido_envio.vista_pedidos_resumen;
CREATE SQL SECURITY DEFINER VIEW tpi_pedido_envio.vista_pedidos_resumen AS
SELECT
  p.id_pedido,
  p.id_cliente,
  c.cliente_nombre,
  p.fecha_pedido,
  p.estado_pedido,
  p.total,
  p.observaciones
FROM tpi_pedido_envio.PEDIDO p
JOIN tpi_pedido_envio.CLIENTE c ON p.id_cliente = c.id_cliente;
```

A continuación vemos la **vista_pedido_resumen**, realizada con la función **SELECT**.

	Result Grid		Filter Rows:		Export:		Wrap Cell Content:		Fetch rows:	
	id_pedido	id_cliente	cliente_nombre	fecha_pedido	estado_pedido	total	observaciones			
▶	1	1	Juan Pérez	2024-10-15 10:30:00	Confirmado	37000.00	Entrega en horario comercial			
	2	1	Juan Pérez	2024-10-16 09:15:30	Despachado	50000.00	Entregar en portería			
	3	7422	Carlos Rodríguez 70	2025-01-12 00:00:00	Pendiente	0.00	Envío urgente			
	4	2317	Juan Rodríguez 185	2025-04-14 00:00:00	En Preparacion	0.00				
	5	13572	Laura Díaz 120	2025-06-29 00:00:00	Confirmado	0.00	Envío urgente			
	6	27277	Lucas Pérez 25	2025-08-01 00:00:00	En Preparacion	0.00	Envío urgente			
	7	6689	Carlos García 17	2025-07-12 00:00:00	Entregado	0.00	NULL			
	8	24627	Valeria Pérez 75	2025-02-07 00:00:00	Pendiente	0.00	Envío urgente			
	9	15140	Laura Fernández 168	2025-09-02 00:00:00	Pendiente	0.00	NULL			
	10	18090	Diego González 18	2025-01-19 00:00:00	Cancelado	0.00	NULL			
	11	79445	Lucas González 53	2025-10-10 00:00:00	Confirmado	0.00	NULL			
	vista_pedidos_resumen 5									

A continuación con el siguiente script subrayado en azul, hacemos uso de la tabla **tpi_pedido_envio** y mostramos todas las tablas donde el tipo de tabla es 'VIEW' (vista).

Y de esta forma evidenciamos que se crearon correctamente las vistas antes mencionadas.

```

1 • USE tpi_pedido_envio;
2 • SHOW FULL TABLES IN tpi_pedido_envio WHERE TABLE_TYPE = 'VIEW';
3 • show tables;
4 -- Vista de clientes pública (sin email ni dirección)
5 • DROP VIEW IF EXISTS tpi_pedido_envio.vista_clientes_publicos;
```

	Result Grid		Filter Rows:		Export:		Wrap Cell Content:	
	Tables_in_tpi_pedido_envio		Table_type					
▶	vista_clientes_publicos		VIEW					
	vista_pedidos_resumen		VIEW					

Referencias: Anexo Ref 2

Violacion de restricciones (PK, FK, UNIQUE, CHECK)

Violacion de PK (duplicado de llave primaria)

En este caso para evidenciar el buen funcionamiento de las restricciones aplicadas a la llave primaria **id_producto**, debemos ingresar un dato que ya esté en la base de esta forma va contra una de los principios de las llaves primarias, que sean únicas.

La inserción que vamos a realizar es la siguiente:

- Suponiendo que **id_producto** es 1.
- Intentó forzar duplicado en PRODUCTO (falla si **id_producto** es PK y AUTO_INCREMENT)

```
INSERT INTO localidades(id_localidad,ciudad,provincia,codigo_postal)
VALUES (1,'Ciudad Invetada', 'Provincia tambien', '123');
```

El error es el esperado: **ERROR 1062 (23000): Duplicate entry '1' for key 'localidades.PRIMARY'**.

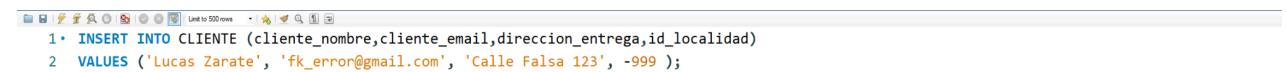
MySQL said: 

```
#1062 - Duplicate entry '1' for key 'localidades.PRIMARY'
```

Violacion de FK (referencia a cliente inexistente)

Se intenta la inserción de un **pedido** con **id_cliente** inexistente, por ejemplo: **999999**.

```
INSERT INTO CLIENTE
(cliente_nombre,cliente_email,direccion_entrega,id_localidad)
VALUES
('Lucas Zarate', 'fk_error@gmail.com', 'Calle Falsa 123', -999 );
```



1. INSERT INTO CLIENTE (cliente_nombre,cliente_email,direccion_entrega,id_localidad)
2. VALUES ('Lucas Zarate', 'fk_error@gmail.com', 'Calle Falsa 123', -999);



Output

#	Time	Action
1	19:33:21	INSERT INTO CLIENTE (cliente_nombre,cliente_email,direccion_entrega,id_localidad) VALUES ('Lucas Zarate', 'fk_error@gmail.com', 'Calle Falsa 123', -999)

Message

Error Code: 1452. Cannot add or update a child row: a foreign key constraint fails ('utn_pedido_envio'.'cliente', CONSTRAINT fk_cliente_localidad FOREIGN KEY (id_localidad) REFERENCES localidades (id_localidad))

Error esperado: ERROR 1452: Cannot add or update a child row: a foreign key constraint fails

La razón es porque **fk_pedido_cliente** que obliga a que **id_cliente** exista en **CLIENTE**.

Violacion de UNIQUE (código de producto duplicado)

Para hacer ver como trabaja la restricción **UNIQUE**, respetando que no haya duplicados en este caso en **producto_codigo**, ejecutamos dos inserciones de datos para un mismo código “**codigo_001**”.

```
INSERT INTO PRODUCTO (producto_nombre, producto_codigo, precio_unitario, stock_disponible)
VALUES ('Notebook_duplicada', 'codigo_001', 100.00, 10);

-- intentar duplicar codigo
INSERT INTO PRODUCTO (producto_nombre, producto_codigo, precio_unitario, stock_disponible)
VALUES ('Notebook_duplicada', 'codigo_001', 50.00, 5);
```

El error a continuación es el esperado, en este se evidencia el duplicado.

MySQL said: 

```
#1062 - Duplicate entry 'codigo_001' for key 'producto.uk_producto_codigo'
```

La razón de error es por que constraint **UNIQUE** evita código repetidos.

Violacion de CHECK (precio negativo, cantidad inválida)

En este caso vamos a enfrentarnos a la restricción CHECK, su función es el chequeo de parámetros impuestos por nosotros.

Por ejemplo, vamos a hacer el intento de cargar un precio negativo a un producto.

Adjunto donde se halla el CHECK que vamos a probar:

```
CREATE TABLE PEDIDO_PRODUCTO (
    id_detalle INT AUTO_INCREMENT PRIMARY KEY,
    id_pedido INT NOT NULL,
    id_producto INT NOT NULL,
    cantidad INT NOT NULL,
    precio_unitario DECIMAL(10,2) NOT NULL,
    subtotal DECIMAL(10,2) NOT NULL,
    CONSTRAINT fk_detalle_pedido FOREIGN KEY (id_pedido)
        REFERENCES PEDIDO(id_pedido)
        ON DELETE CASCADE
        ON UPDATE CASCADE,
    CONSTRAINT fk_detalle_producto FOREIGN KEY (id_producto)
        REFERENCES PRODUCTO(id_producto)
        ON DELETE RESTRICT
        ON UPDATE CASCADE,
    CONSTRAINT chk_detalle_cantidad CHECK (cantidad > 0),
    CONSTRAINT chk_detalle_precio CHECK (precio_unitario >= 0),
    CONSTRAINT chk_detalle_subtotal CHECK (subtotal >= 0),
    CONSTRAINT chk_detalle_calculo CHECK (ABS(subtotal - (cantidad * precio_unitario)) < 0.01)
```

A continuación el script con **precio_unitario = -102.00** que ejecutamos es el siguiente:

```
INSERT INTO PRODUCTO (producto_nombre, producto_codigo, precio_unitario, stock_disponible)
VALUES ('Tele_regalado', '001', -102.00, 1);
```

El error es siguiente:

MySQL said: 

```
#3819 - Check constraint 'chk_producto_precio' is violated.
```

El error indica que el **CHECK CONSTRAINT** su violado es decir que no cumple con la condiciones impuestas (**precio_unitario >= 0**).

Consulta segura + prueba anti-SQL injection

Creamos el procedimiento almacenado con SQL estático.

```

1  DELIMITER //
2
3  CREATE DEFINER = `root`@`localhost`
4  * PROCEDURE sp_ConsultarPedidos_Min(
5      IN p_id_cliente INT,          -- NULL = todos
6      IN p_fecha_desde DATETIME,   -- NULL = sin límite inferior
7      IN p_fecha_hasta DATETIME    -- NULL = sin límite superior
8  )
9  SQL SECURITY DEFINER
10 * BEGIN
11    -- Validaciones simples (evitan entradas lógicas inadecuadas)
12 *   IF p_id_cliente IS NOT NULL AND p_id_cliente <= 0 THEN
13     SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'id_cliente inválido (debe ser > 0 o NULL)';
14   END IF;
15
16 *   IF p_fecha_desde IS NOT NULL AND p_fecha_hasta IS NOT NULL THEN
17 *     IF p_fecha_desde > p_fecha_hasta THEN
18     SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Rango de fechas inválido: fecha_desde > fecha_hasta';
19   END IF;
20 *   END IF;
21
22    -- Consulta ESTÁTICA y segura: NO concatenación dinámica
23  SELECT
24    id_pedido,
25    id_cliente,
26    fecha_pedido,
27    estado_pedido,
28    total,
29    observaciones
30  FROM PEDIDO
31  WHERE (p_id_cliente IS NULL OR id_cliente = p_id_cliente)
32    AND (p_fecha_desde IS NULL OR fecha_pedido >= p_fecha_desde)
33    AND (p_fecha_hasta IS NULL OR fecha_pedido <= p_fecha_hasta)
34  ORDER BY fecha_pedido DESC;
35
36 END;
37
38 DELIMITER ;
39
40 * CALL sp_ConsultarPedidos_Min(12970, "2025-01-02'; DROP TABLE pedidos; --",'2025-02-02 23:59:59');
41 * SHOW WARNINGS;

```

Result Grid		
Level	Code	Message
Warning	1265	Data truncated for column 'p_fecha_desde' at r...

Comenzamos creando el procedimiento llamado “sp_ConsultarPedidos_Min”, dentro de la primera parte. Tenemos la declaración del procedimiento que nos indica cuales son los datos que se pueden pasar y de qué tipo de valores deben ser dichos datos.

Dentro del BEGIN y posterior al SECURITY DEFINER tenemos lo que va a ser el cuerpo de nuestro procedimiento almacenado, dentro de este bloque vamos a poner la sentencia SQL que va a realizar. En este caso lo que queremos que realice el procedimiento es una búsqueda parametrizada con 3 datos dentro de los pedidos.

```


PROCEDURE sp_ConsultarPedidos_Min(
  IN p_id_cliente INT,          -- NULL = todos
  IN p_fecha_desde DATETIME,    -- NULL = sin límite inferior
  IN p_fecha_hasta DATETIME    -- NULL = sin límite superior
)


```

Primero le vamos a pasar el id_cliente que es un dato personal de cada usuario que en este caso haría su consulta, luego le pasamos dos fechas, una como límite inferior y otra como límite superior.

```


-- Validaciones simples (evitan entradas lógicas inadecuadas)
IF p_id_cliente IS NOT NULL AND p_id_cliente <= 0 THEN
  SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'id_cliente inválido (debe ser > 0 o NULL)';
END IF;

IF p_fecha_desde IS NOT NULL AND p_fecha_hasta IS NOT NULL THEN
  IF p_fecha_desde > p_fecha_hasta THEN
    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Rango de fechas inválido: fecha_desde > fecha_hasta';
  END IF;
END IF;


```

Por medio de unas validaciones para evitar que nos agreguen datos inválidos o “trolls” realizamos la consulta de SQL, también devolverá feedback y manejo para datos inválidos.

```


SELECT
  id_pedido,
  id_cliente,
  fecha_pedido,
  estado_pedido,
  total,
  observaciones
FROM PEDIDO

WHERE (p_id_cliente IS NULL OR id_cliente = p_id_cliente)
  AND (p_fecha_desde IS NULL OR fecha_pedido >= p_fecha_desde)
  AND (p_fecha_hasta IS NULL OR fecha_pedido <= p_fecha_hasta)
ORDER BY fecha_pedido DESC;


```

La consulta SQL selecciona todos los ítems de la tabla pedidos y usando un WHERE filtra los datos ingresados comparándolos con los que tenemos en la tabla y los ordena de manera descendente.

Por último hacemos una llamada al procedimiento y le pasamos unos parámetros, en los cuales, algunos usuarios con intenciones de bajo nivel moral podrían intentar hacer un ataque a la integridad de nuestra base de datos, por lo que vamos a hacer dos demostraciones de ejecución, caso correcto y caso defensa.

Caso correcto:

```
43 • CALL sp_ConsultarPedidos_Min(483, "2025-10-16 00:00:00", '2025-10-18 00:00:00');
```

	id_pedido	id_cliente	fecha_pedido	estado_pedido	total	observaciones
▶	94526	483	2025-09-20 00:00:00	En Preparacion	203478.55	NULL
	11678	483	2025-09-20 00:00:00	En Preparacion	66446.72	NULL
	21997	483	2025-02-16 00:00:00	Confirmado	106142.73	NULL
	12	483	2024-12-11 00:00:00	Confirmado	183030.41	NULL
	82547	483	2024-12-08 00:00:00	Confirmado	131602.41	NULL

Realizamos la consulta de forma correcta sin errores, donde nos devuelve la db lo que estábamos consultando sin ningún problema.

Caso defensa:

```
40 • CALL sp_ConsultarPedidos_Min(12970, "2025-10-16 00:00:00; DROP TABLE pedidos; --", '2025-10-18 00:00:00');
41 • SHOW WARNINGS;
```

	id_pedido	id_cliente	fecha_pedido	estado_pedido	total	observaciones
▶	4897	12970	2025-05-15 00:00:00	Cancelado	68147.13	NULL
	15	12970	2025-02-21 00:00:00	Cancelado	77469.39	NULL
	32893	12970	2024-12-28 00:00:00	Entregado	146571.44	Envío urgente

Al ejecutar la consulta maliciosa, vemos que funciona igual, ya que solamente parsea la parte de la fecha, ignorando sistemáticamente el intento de inyección.

```
40 • CALL sp_ConsultarPedidos_Min(12970, "2025-10-16 00:00:00; DROP TABLE pedidos; --", '2025-10-18 00:00:00');
41 • SHOW WARNINGS;
```

Level	Code	Message
Warning	1265	Data truncated for column 'p_fecha_desde' at row 1

Se observa el warning que nos indica dicho truncamiento del parseo. Siendo efectivamente evitado el intento de inyección SQL.

Documentación de ayuda y utilización de IA:

Utilizamos la IA de forma constructiva, con el prompt recomendado por la cátedra principalmente para el diseño del procedimiento almacenado con SQL estático que nos permitió corroborar la defensa ante un intento de inyección SQL malicioso.

Comenzamos por darle contexto al GPT-5 para que sepa cual era el esquema de la tabla de pedidos, explicando el panorama y las necesidades de soporte que el usuario tenía. Principalmente para pasarle la idea del diseño de la defensa que nosotros teníamos en mente.

El prompt definitivo del contexto fue:

bueno, en la etapa 4 tengo que hacer un par de cosas, creacion de usuarios con menos privilegios, usar vistas para filtrar y simplificar acceso, y algunas pruebas de integridad referencial y de unicidad. ahora bien, la madre del cordero es que tengo que hacer una consulta segura, tengo que implementarla, yo quiero usar procedimiento almacenado de SQL, y tengo que adjuntarle una prueba anti-inyeccion documentada, con el intento malicioso bloqueado.

Luego de eso GPT nos respondió de la siguiente forma:

Perfecto – muy buen ejercicio: cubre seguridad, control de accesos y pruebas. Voy a guiarte paso a paso con pistas, ejemplos y explicaciones (sin darte una “solución copiada y pegada” completa para todo el TP). Al final tendrás fragmentos de procedimiento almacenado seguros, un ejemplo de variante vulnerable (para que entiendas por qué falla) y cómo documentar la prueba anti-inyección.

En su respuesta pasó por todos los puntos que le habíamos nombrado que tenía que tener la parte 4, pero lo que más nos sirvió fueron los ejemplos de excelente comparación, entre una consulta vulnerable a un ataque, sin filtros, sin validaciones vs una consulta fuerte, segura y con todo lo que necesita para defenderse de un ataque de inyección.

Esos dos ejemplos propuestos en la primera respuesta de la IA formó la base, ya que nos puso en contexto de los requerimientos básicos para nuestro procedimiento.

Por último fuimos puliendo ideas del diseño de la propuesta de procedimiento para la consulta, más allá de lo técnico y llegamos al diseño final luego de probar otros previamente, encontrando que el estilo consulta era más apropiado para la consigna que un estilo creación de pedido que era lo que previamente habíamos pensado.

Link al chat completo: <https://chatgpt.com/share/69029691-7550-8011-ad06-23ca56cdbf2d>

Etapa 5

Simulación de deadlock

Deadlock o punto muerto: es una instancia donde 2 o más transacciones se encuentran en espera, con la particularidad de que ambas se encuentran bloqueadas (mecanismo de control utilizado en la concurrencia)

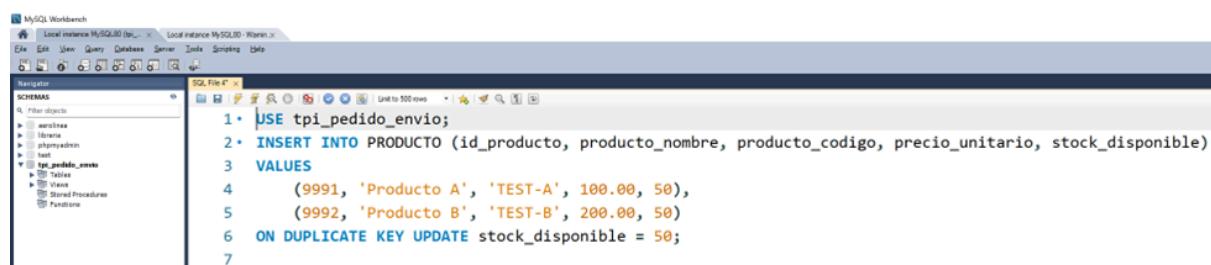
Las soluciones que existen a este problema son:

- Algoritmos de detección de bloqueos.
- Selección de víctimas (que transacción abortar)
- Retroceso y reintento automáticos
- Mecanismos de tiempo de espera.

Simulación de deadlock:

Preparamos **PRODUCTO** para la prueba:

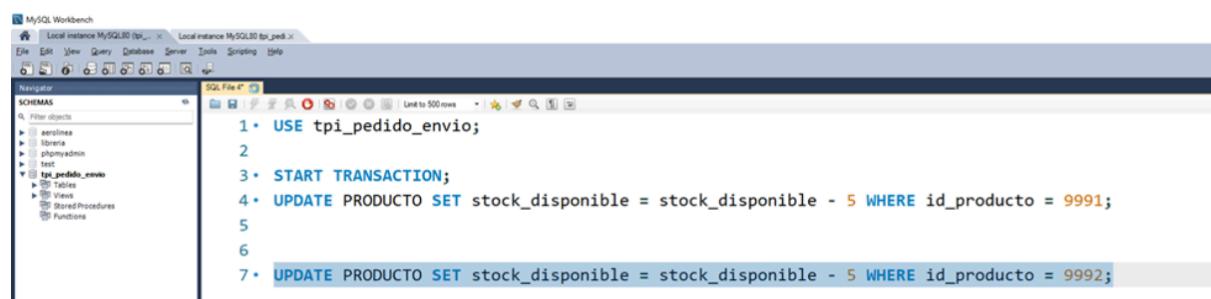
SESIÓN 1



```

USE tpi_pedido_envio;
INSERT INTO PRODUCTO (id_producto, producto_nombre, producto_codigo, precio_unitario, stock_disponible)
VALUES
(9991, 'Producto A', 'TEST-A', 100.00, 50),
(9992, 'Producto B', 'TEST-B', 200.00, 50)
ON DUPLICATE KEY UPDATE stock_disponible = 50;
    
```

Y luego procedemos a iniciar la transacción y realizar los UPDATES que nos ayudaran a demostrar un deadlock.



```

USE tpi_pedido_envio;
START TRANSACTION;
UPDATE PRODUCTO SET stock_disponible = stock_disponible - 5 WHERE id_producto = 9991;
UPDATE PRODUCTO SET stock_disponible = stock_disponible - 5 WHERE id_producto = 9992;
    
```

Output

#	Time	Action	Message
1	19:17:36	USE tpi_pedido_envio	0 row(s) affected
2	19:17:36	INSERT INTO PRODUCTO (id_producto, producto_nombre, producto_codigo, precio_unitario, stock_disponible) VALUES (9991, 'Producto A', 'TEST-A', 100.00, 50), (9992, 'Producto B', 'TEST-B', 200.00, 50)	0 rows affected Records: 2 Duplicates: 0 Warnings: 0
3	19:19:47	USE tpi_pedido_envio	0 row(s) affected
4	19:19:52	START TRANSACTION	0 row(s) affected
5	19:19:58	UPDATE PRODUCTO SET stock_disponible = stock_disponible - 5 WHERE id_producto = 9991	1 row(s) affected Rows matched: 1 Changed: 1 Warnings: 0
6	19:22:41	UPDATE PRODUCTO SET stock_disponible = stock_disponible - 5 WHERE id_producto = 9992	Running...

SESIÓN 2

MySQL Workbench

Local instance MySQL8.0 for pede.x Local instance MySQL8.0 (pede)

File Edit View Query Database Server Tools Scripting Help

Navegador

SCHEMAS

- Filter objects
- airline
- base
- phpMyAdmin
- test
- tpi_pedido_envio**
 - Tables
 - Views
 - Stored Procedures
 - Functions

Query 1

```

1 • USE tpi_pedido_envio;
2
3 • START TRANSACTION;
4 • UPDATE PRODUCTO SET stock_disponible = stock_disponible - 5 WHERE id_producto = 9992;
5
6
7 • UPDATE PRODUCTO SET stock_disponible = stock_disponible - 5 WHERE id_producto = 9991;
^
  
```

Output

#	Time	Action	Message
1	19:20:15	USE tpi_pedido_envio;	0 row(s) affected
2	19:20:16	USE tpi_pedido_envio	0 row(s) affected
3	19:20:20	START TRANSACTION	0 row(s) affected
4	19:20:48	UPDATE PRODUCTO SET stock_disponible = stock_disponible - 5 WHERE id_producto = 9992	1 row(s) affected Rows matched: 1 Changed: 1 Warnings: 0
5	19:22:52	UPDATE PRODUCTO SET stock_disponible = stock_disponible - 5 WHERE id_producto = 9991	Error Code: 1213. Deadlock found when trying to get lock; try restarting transaction

Para esta prueba preparamos dos productos de prueba con id_producto 9991 y 9992 y abrimos dos sesiones simultáneas en MySQL Workbench.

En la primera sesión bloqueamos el producto 9991, y en la segunda bloqueamos el 9992.

Como siguiente paso, intentamos que cada sesión acceda al producto bloqueado por la otra, creando un ciclo de espera.

MySQL detecta esta situación y aborta una de las transacciones con el *Error Code: 1213. Deadlock found when trying to get lock; try restarting transaction*.

Esto demuestra que el motor tiene un mecanismo de detección de deadlocks que previene bloqueos indefinidos.

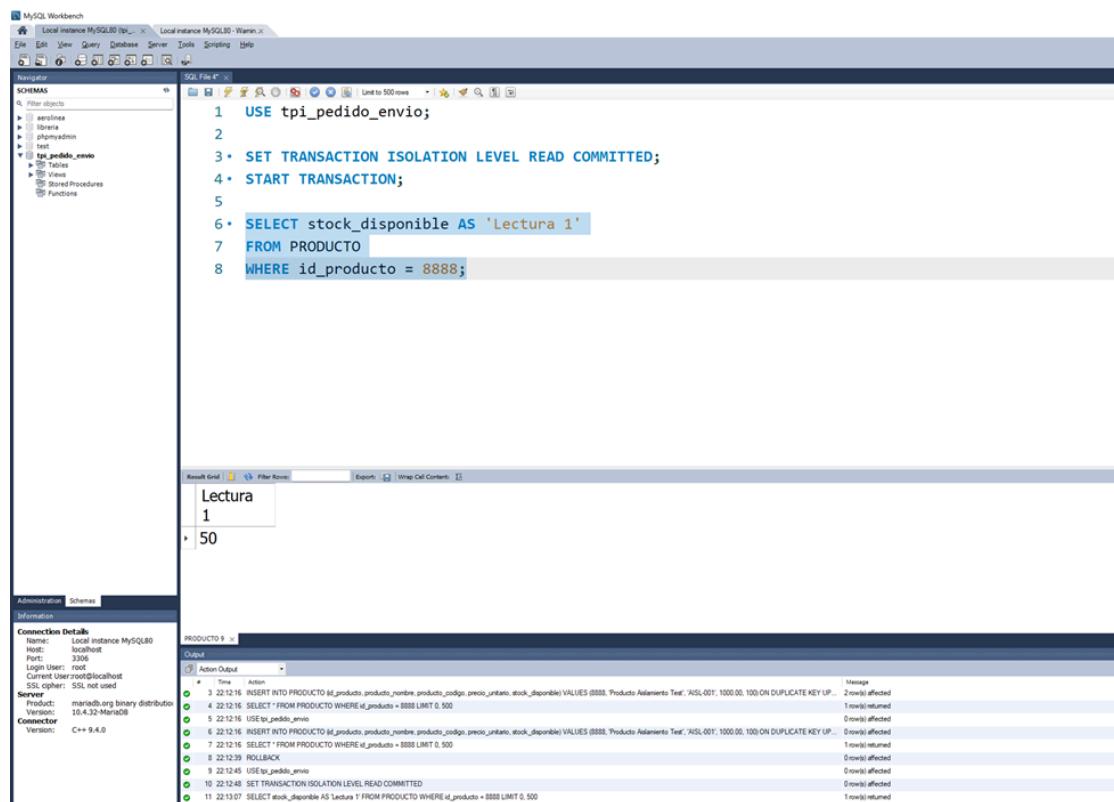
READ COMMITTED

Preparamos los datos sobre los cuales trabajaremos.

SESIÓN 1

```
USE tpi_pedido_envio;
    SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;
    SELECT stock_disponible AS 'Lectura 1'
FROM PRODUCTO
    WHERE id_producto = 8888;
```

Y luego ejecutamos una cláusula SELECT en donde el resultado esperado de la Lectura 1 = 50.



The screenshot shows the MySQL Workbench interface with two tabs open: 'Local instance MySQL80 tpi...' and 'Local instance MySQL80 - Waran...'. The left sidebar shows the 'Navigator' with 'SCHEMAS' expanded, containing 'tpi_pedido_envio' which has 'Tables', 'Views', 'Stored Procedures', and 'Functions'. The main area is the 'SQL File 4' tab where the following SQL code is pasted:

```
1 USE tpi_pedido_envio;
2
3 • SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
4 • START TRANSACTION;
5
6 • SELECT stock_disponible AS 'Lectura 1'
7 FROM PRODUCTO
8 WHERE id_producto = 8888;
```

Below the SQL editor is the 'Results Grid' pane, which displays the query results:

Lectura
1
50

At the bottom of the interface is the 'Output' pane, which shows the transaction log with the following entries:

- # 3 22:12:16 INSERT INTO PRODUCTO (id_producto, producto_nombre, producto_codigo, precio_unitario, stock_disponible) VALUES (8888, 'Producto Alimentos Test', 'ASL-001', 1000.00, 100) ON DUPLICATE KEY UPDATE stock_disponible=100
- # 4 22:12:16 SELECT * FROM PRODUCTO WHERE id_producto = 8888 LIMIT 0, 500
- # 5 22:12:16 USE tpi_pedido_envio
- # 6 22:12:16 INSERT INTO PRODUCTO (id_producto, producto_nombre, producto_codigo, precio_unitario, stock_disponible) VALUES (8888, 'Producto Alimentos Test', 'ASL-001', 1000.00, 100) ON DUPLICATE KEY UPDATE stock_disponible=100
- # 7 22:12:16 SELECT * FROM PRODUCTO WHERE id_producto = 8888 LIMIT 0, 500
- # 8 22:12:39 ROLLBACK
- # 9 22:12:45 USE tpi_pedido_envio
- # 10 22:12:48 SET TRANSACTION ISOLATION LEVEL READ COMMITTED
- # 11 22:13:07 SELECT stock_disponible AS 'Lectura 1' FROM PRODUCTO WHERE id_producto = 8888 LIMIT 0, 500

Luego en la sesión 2 se hace un UPDATE del stock disponible para el id_producto 8888 a 10 unidades sin hacer el COMMIT.



The screenshot shows the MySQL Workbench interface with two tabs open: 'Local instance MySQL80 tpi_pedido...' and 'Local instance MySQL80 - Waran...'. The left sidebar shows the 'Navigator' with 'SCHEMAS' expanded, containing 'tpi_pedido_envio' which has 'Tables', 'Views', 'Stored Procedures', and 'Functions'. The main area is the 'Query 1' tab where the following SQL code is pasted:

```
1 • USE tpi_pedido_envio;
2
3 • START TRANSACTION;
4
5 • UPDATE PRODUCTO
6     SET stock_disponible = 10
7     WHERE id_producto = 8888;
```

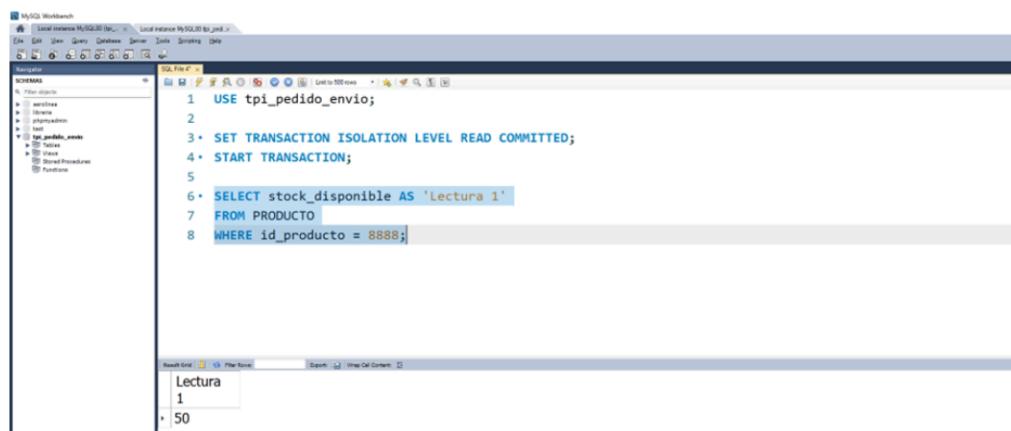
Below the SQL editor is the 'Results Grid' pane, which displays the query results:

Lectura
1
50

At the bottom of the interface is the 'Output' pane, which shows the transaction log with the following entries:

- # 1 • USE tpi_pedido_envio;
- # 2 • START TRANSACTION;
- # 5 • UPDATE PRODUCTO
 - # 6 SET stock_disponible = 10
 - # 7 WHERE id_producto = 8888;
- # 8 22:13:07 SELECT stock_disponible AS 'Lectura 1' FROM PRODUCTO WHERE id_producto = 8888 LIMIT 0, 500

En esta sesión no se refleja el cambio por que todavía no se hizo el COMMIT en la sesión 2.

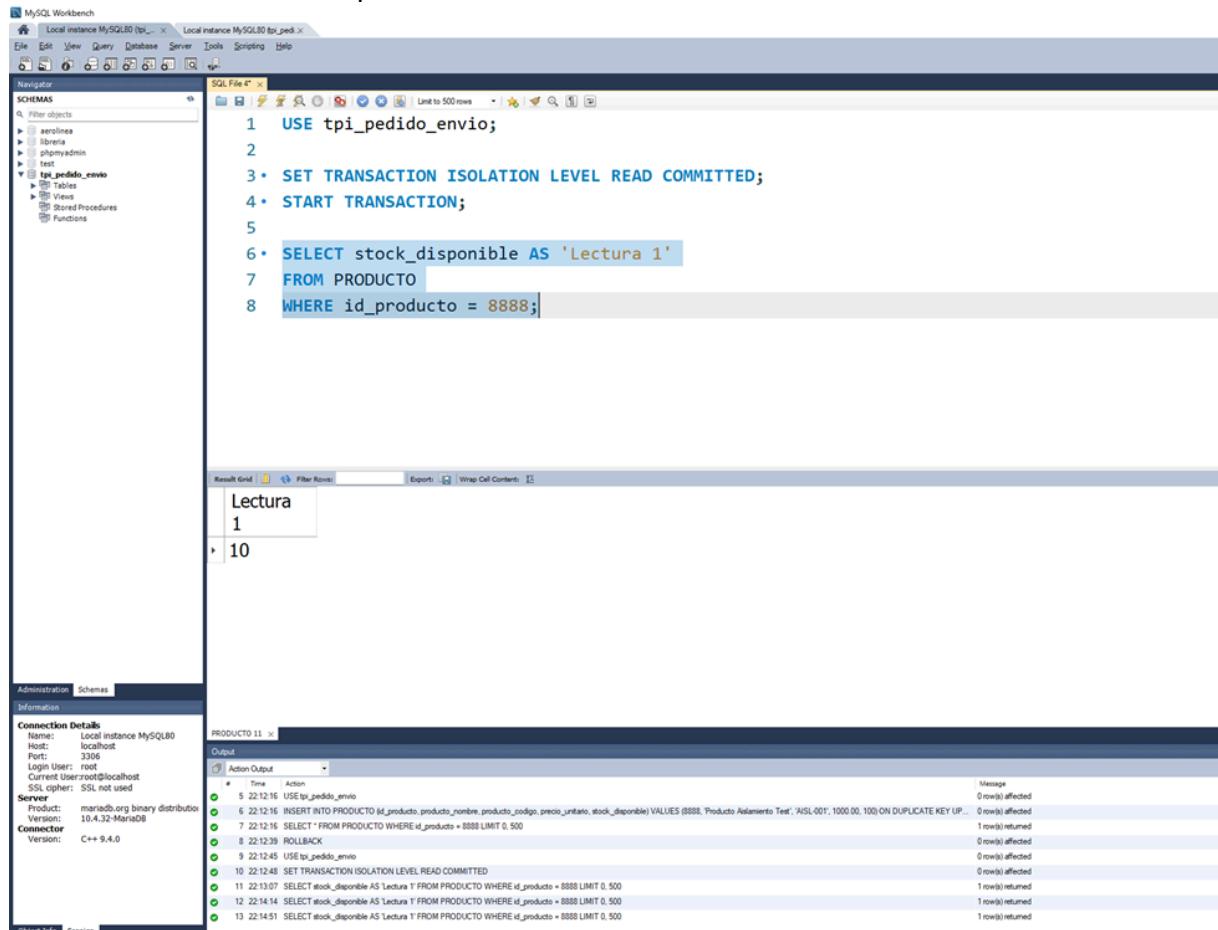


```

USE tpi_pedido_envio;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;
SELECT stock_disponible AS 'Lectura 1'
FROM PRODUCTO
WHERE id_producto = 8888;
  
```

Result Grid
Lectura 1 50

Luego de realizar el COMMIT en la sesión 2, volvemos a ejecutar el SELECT y vemos que ahora si el stock se actualizó a 10 unidades. Esto se debe a que en **READ COMMITTED** la sesión ve commits hechos por otras sesiones.



```

USE tpi_pedido_envio;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;
SELECT stock_disponible AS 'Lectura 1'
FROM PRODUCTO
WHERE id_producto = 8888;
  
```

Result Grid
Lectura 1 10

Connection Details:
 Name: Local instance MySQL80
 Host: localhost
 Port: 3306
 Local User: root
 Current User:root@localhost
 SSL cipher: SSL not used
Server:
 Product: mariadb.org binary distribution
 Version: 10.4.32-MariaDB
Connector:
 Version: C++ 9.4.0

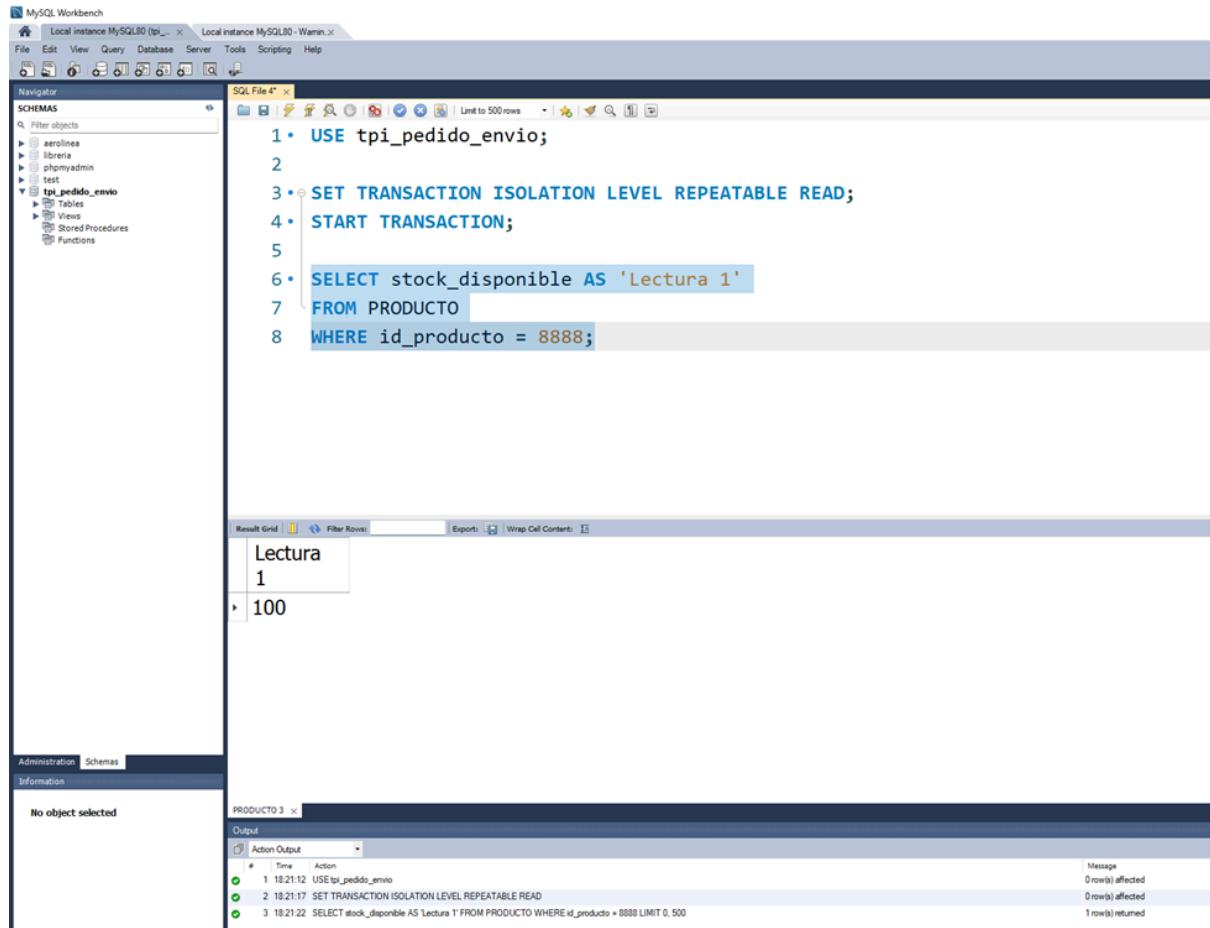
Action Output:

Time	Action	Message
5 22:12:16	USE tpi_pedido_envio	0 row(s) affected
6 22:12:16	INSERT INTO PRODUCTO (id_producto, producto_nombre, producto_codigo, precio_unitario, stock_disponible) VALUES (8888, 'Producto Aislamiento Test', 'AISL-001', 1000.00, 100) ON DUPLICATE KEY UPDATE	0 row(s) affected
7 22:12:16	SELECT * FROM PRODUCTO WHERE id_producto = 8888 LIMIT 0, 500	1 row(s) returned
8 22:12:39	ROLLBACK	0 row(s) affected
9 22:12:45	USE tpi_pedido_envio	0 row(s) affected
10 22:12:48	SET TRANSACTION ISOLATION LEVEL READ COMMITTED	0 row(s) affected
11 22:13:07	SELECT stock_disponible AS Lectura 1 FROM PRODUCTO WHERE id_producto = 8888 LIMIT 0, 500	1 row(s) returned
12 22:14:14	SELECT stock_disponible AS Lectura 1 FROM PRODUCTO WHERE id_producto = 8888 LIMIT 0, 500	1 row(s) returned
13 22:14:51	SELECT stock_disponible AS Lectura 1 FROM PRODUCTO WHERE id_producto = 8888 LIMIT 0, 500	1 row(s) returned

REPEATABLE READ

Previo a la ejecución restauramos los valores de stock y asignamos 100 unidades de stock.

Así mismo determinamos el uso de REPEATABLE READ al iniciar la transacción.



The screenshot shows the MySQL Workbench interface with the following details:

- Navigator:** Shows the schema `tpi_pedido_envio` selected.
- SQL Editor (SQL File 4):**

```

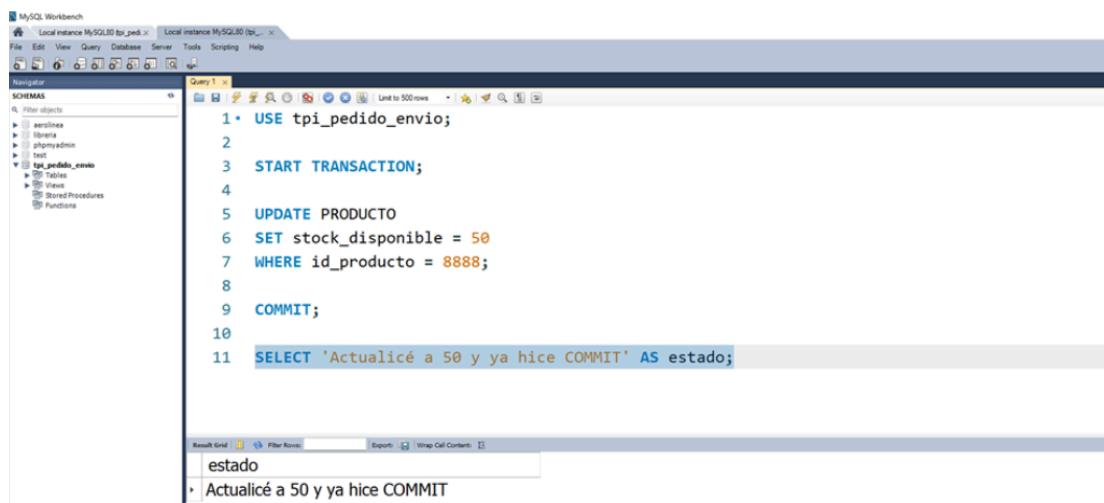
1 • USE tpi_pedido_envio;
2
3 • SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
4 • START TRANSACTION;
5
6 • SELECT stock_disponible AS 'Lectura 1'
7 FROM PRODUCTO
8 WHERE id_producto = 8888;
  
```
- Result Grid:** Displays the result of the query:

Lectura	1
	100
- Output:** Shows the transaction log:

Action	Time	Action	Message
1	18:21:12	USE tpi_pedido_envio	0 row(s) affected
2	18:21:17	SET TRANSACTION ISOLATION LEVEL REPEATABLE READ	0 row(s) affected
3	18:21:22	SELECT stock_disponible AS Lectura 1 FROM PRODUCTO WHERE id_producto = 8888 LIMIT 0, 500	1 row(s) returned

SESIÓN 2

Confirmamos que se hizo COMMIT y ahora la tabla contiene 50.



The screenshot shows the MySQL Workbench interface with the following details:

- Navigator:** Shows the schema `tpi_pedido_envio` selected.
- SQL Editor (Query 1):**

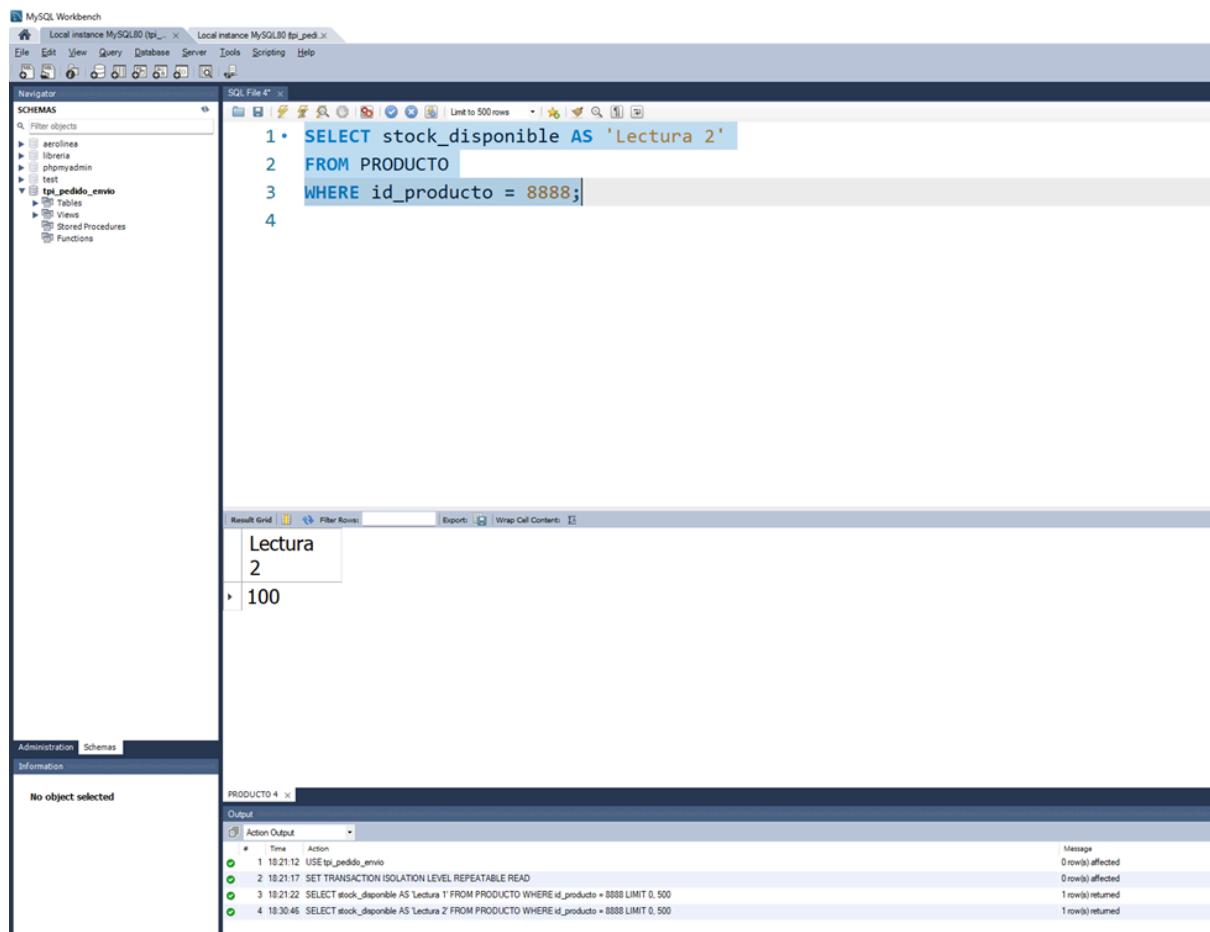
```

1 • USE tpi_pedido_envio;
2
3 • START TRANSACTION;
4
5 UPDATE PRODUCTO
6 SET stock_disponible = 50
7 WHERE id_producto = 8888;
8
9 COMMIT;
10
11 SELECT 'Actualicé a 50 y ya hice COMMIT' AS estado;
  
```
- Result Grid:** Displays the result of the query:

estado	Actualicé a 50 y ya hice COMMIT
--------	---------------------------------

SESIÓN 1

El resultado esperado es 100 ya que con **REPEATABLE READ** la transacción mantiene la vista consistente que tenía al inicio; y todavía no ve el commit.



```

1 • SELECT stock_disponible AS 'Lectura 2'
2 FROM PRODUCTO
3 WHERE id_producto = 8888;
4

```

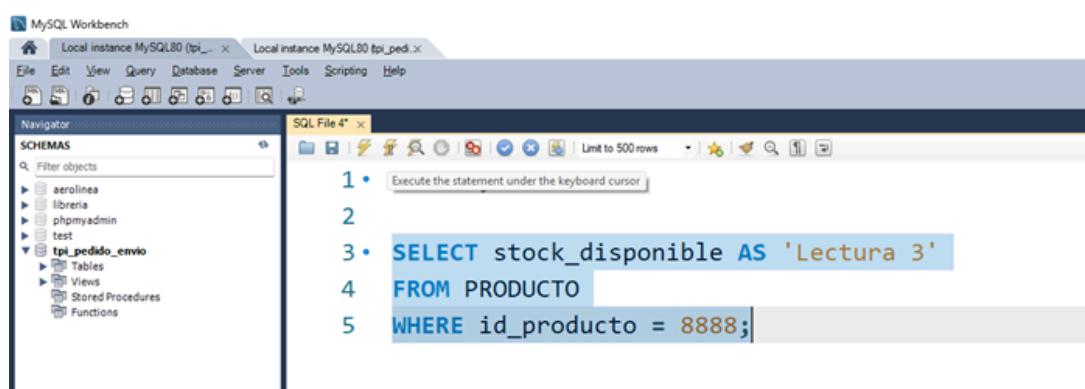
Result Grid		
	Filter Rows	Export
Lectura		Wrap Cell Content
2		
100		

Output

Action	Time	Action	Message
1	18:21:12	USE `tpi_pedido_envio`	0 row(s) affected
2	18:21:17	SET TRANSACTION ISOLATION LEVEL REPEATABLE READ	0 row(s) affected
3	18:21:22	SELECT stock_disponible AS 'Lectura 1' FROM PRODUCTO WHERE id_producto = 8888 LIMIT 0, 500	1 row(s) returned
4	18:30:45	SELECT stock_disponible AS 'Lectura 2' FROM PRODUCTO WHERE id_producto = 8888 LIMIT 0, 500	1 row(s) returned

Luego continuando en la sesión 1, realizamos un COMMIT y volvemos a ejecutar la cláusula SELECT

Y podemos ver que el nuevo valor de stock es 50, ya que después de cerrar la transacción, la sesión muestra el valor por el que se hizo COMMIT en la SESIÓN 2.



```

1 • Execute the statement under the keyboard cursor.
2
3 • SELECT stock_disponible AS 'Lectura 3'
4 FROM PRODUCTO
5 WHERE id_producto = 8888;

```

Result Grid		
		Filter Rows:
Lectura	3	
	50	

Tenemos la lectura con la sentencia y los logs de la misma consulta

PRODUCTO 5			
Output			
#	Time	Action	Message
1	18:21:12	USE tpi_pedido_envio	0 row(s) affected
2	18:21:17	SET TRANSACTION ISOLATION LEVEL REPEATABLE READ	0 row(s) affected
3	18:21:22	SELECT stock_disponible AS 'Lectura 1' FROM PRODUCTO WHERE id_producto = 8888 LIMIT 0, 500	1 row(s) returned
4	18:30:46	SELECT stock_disponible AS 'Lectura 2' FROM PRODUCTO WHERE id_producto = 8888 LIMIT 0, 500	1 row(s) returned
5	18:32:53	COMMIT	0 row(s) affected
6	18:32:56	SELECT stock_disponible AS 'Lectura 3' FROM PRODUCTO WHERE id_producto = 8888 LIMIT 0, 500	1 row(s) returned

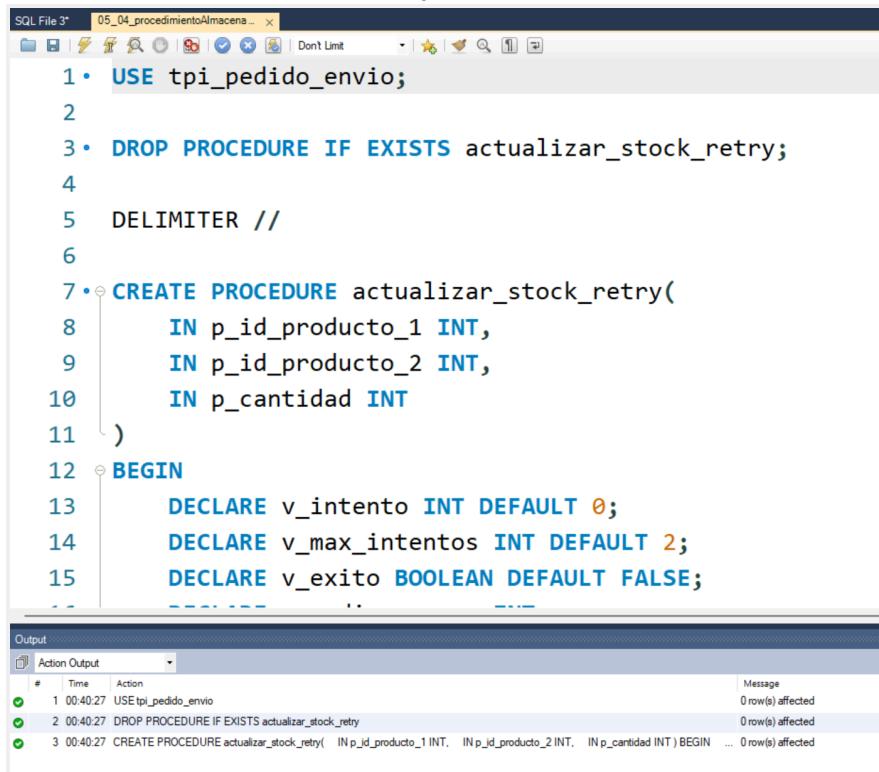
Al comparar **READ COMMITTED** con **REPEATABLE READ**, notamos que con el primero una misma consulta devuelve resultados diferentes dentro de la misma transacción si en otra sesión se realizan cambios, lo que puede causar inconsistencias, por otro lado **REPEATABLE READ**, nos garantiza que los **SELECT** de la consulta muestra solo un estado consistente de los datos durante la transacción, eliminando el problema de inconsistencias encontrados en **READ COMMITTED**.

Respecto al uso de la IA para la ejecución de estas pruebas, ChatGPT nos ayudó con la comparación de niveles de aislamiento, principalmente en cómo demostrar las diferencias entre **READ COMMITTED** y **REPEATABLE READ**, como así también nos explicó los comandos para cambiar el nivel de aislamiento, y nos guió en cómo estructurar las pruebas con dos sesiones simultáneas.

Retry ante deadlock

Manejar un error 1213/40001

En primera instancia, luego de validar con la IA sobre la creación de un procedimiento almacenado, procedemos a la ejecución del script que nos permite crear el mismo.



```

SQL File 3* 05_04_procedimientoAlmacena.x
File Edit View Insert Object Tools Help
1 • USE tpi_pedido_envio;
2
3 • DROP PROCEDURE IF EXISTS actualizar_stock_retry;
4
5 DELIMITER //
6
7 • CREATE PROCEDURE actualizar_stock_retry(
8     IN p_id_producto_1 INT,
9     IN p_id_producto_2 INT,
10    IN p_cantidad INT
11 )
12 BEGIN
13     DECLARE v_intento INT DEFAULT 0;
14     DECLARE v_max_intentos INT DEFAULT 2;
15     DECLARE v_exito BOOLEAN DEFAULT FALSE;
16
17     WHILE v_exito = FALSE AND v_intento < v_max_intentos DO
18         -- Your logic here
19         SET v_intento = v_intento + 1;
20     END WHILE;
21
22     IF v_exito THEN
23         SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Operación exitosa';
24     ELSE
25         SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Operación fallida';
26     END IF;
27
28 END;
  
```

Output

#	Time	Action	Message
1	00:40:27	USE tpi_pedido_envio	0 row(s) affected
2	00:40:27	DROP PROCEDURE IF EXISTS actualizar_stock_retry	0 row(s) affected
3	00:40:27	CREATE PROCEDURE actualizar_stock_retry(IN p_id_producto_1 INT, IN p_id_producto_2 INT, IN p_cantidad INT) BEGIN ...	0 row(s) affected

Y utilizamos los mismos productos que creamos al momento de simular un Deadlock cuando llamamos al procedimiento.

En donde en las sesión 1 se actualiza el stock en el primer intento

MySQL Workbench

File Edit View Query Database Server Tools Scripting Help

Navigator SQL File 3*

SCHEMAS aerolinea libreria phpmyadmin test tpi_pedido_envio

Tables Views Stored Procedure Functions

SQL File 3* • USE tpi_pedido_envio;
• START TRANSACTION;
• CALL actualizar_stock_retry(9991, 9992, 1);

Result Grid Filter Rows: Export: Wrap Cell Content: Don't Limit

resultado

ÉXITO: Stock actualizado en intento 1

Result 5 Result 6

Output Action Output

#	Time	Action	Message
1	00:43:12	USE tpi_pedido_envio	0 row(s) affected
2	00:43:12	START TRANSACTION	0 row(s) affected
3	00:43:12	CALL actualizar_stock_retry(9991, 9992, 1)	1 row(s) returned
4	00:43:16	CALL actualizar_stock_retry(9991, 9992, 1)	1 row(s) returned

No object selected

Mientras que en la sesión 2, se detecta un deadlock, se aplica un backoff y un retry, y en el segundo intento se logra actualizar el stock

MySQL Workbench

File Edit View Query Database Server Tools Scripting Help

Navigator Query 1

SCHEMAS aerolinea libreria phpmyadmin test tpi_pedido_envio

Tables Views Stored Procedure Functions

1 • USE tpi_pedido_envio;
2 • START TRANSACTION;
3 • CALL actualizar_stock_retry(9992, 9991, 1);

log

ERROR detected - Código: 1213 - Mensaje: Deadlock found when trying

Result 6 Result 7 Result 8 Result 9 Result 10

Output Action Output

#	Time	Action	Message
1	00:43:14	USE tpi_pedido_envio	0 row(s) affected
2	00:43:14	START TRANSACTION	0 row(s) affected
3	00:43:14	CALL actualizar_stock_retry(9992, 9991, 1)	1 row(s) returned
4	00:43:16	CALL actualizar_stock_retry(9992, 9991, 1)	1 row(s) returned
5	00:43:16	CALL actualizar_stock_retry(9992, 9991, 1)	1 row(s) returned
6	00:43:17	CALL actualizar_stock_retry(9992, 9991, 1)	1 row(s) returned
7	00:43:19	CALL actualizar_stock_retry(9992, 9991, 1)	1 row(s) returned

No object selected

MySQL Workbench

Local instance MySQL80 (tpi_pedido) Local instance MySQL80 (tpi_pedido)

File Edit View Query Database Server Tools Scripting Help

Navigator

SCHEMAS

- aerolinea
- libreria
- phpmyadmin
- test
- tpi_pedido_envio**
 - Tables
 - Views
 - Stored Procedure
 - Functions

Query 1

```

1 • USE tpi_pedido_envio;
2 • START TRANSACTION;
3 • CALL actualizar_stock_retry(9992, 9991, 1);

```

Result Grid | Filter Rows: Export: Wrap Cell Content:

log

DEADLOCK detectado. Aplicando backoff de 1 segundo...

Result 6 Result 7 Result 8 Result 9 Result 10

Output

#	Time	Action	Message
1	00:43:14	USE tpi_pedido_envio	0 row(s) affected
2	00:43:14	START TRANSACTION	0 row(s) affected
3	00:43:14	CALL actualizar_stock_retry(9992, 9991, 1)	1 row(s) returned
4	00:43:16	CALL actualizar_stock_retry(9992, 9991, 1)	1 row(s) returned
5	00:43:16	CALL actualizar_stock_retry(9992, 9991, 1)	1 row(s) returned
6	00:43:17	CALL actualizar_stock_retry(9992, 9991, 1)	1 row(s) returned
7	00:43:19	CALL actualizar_stock_retry(9992, 9991, 1)	1 row(s) returned

MySQL Workbench

Local instance MySQL80 (tpi_pedido) Local instance MySQL80 (tpi_pedido)

File Edit View Query Database Server Tools Scripting Help

Navigator

SCHEMAS

- aerolinea
- libreria
- phpmyadmin
- test
- tpi_pedido_envio**
 - Tables
 - Views
 - Stored Procedure
 - Functions

Query 1

```

1 • USE tpi_pedido_envio;
2 • START TRANSACTION;
3 • CALL actualizar_stock_retry(9992, 9991, 1);

```

Result Grid | Filter Rows: Export: Wrap Cell Content:

resultado

ÉXITO: Stock actualizado en intento 2

Result 6 Result 7 Result 8 Result 9 Result 10

Output

#	Time	Action	Message
1	00:43:14	USE tpi_pedido_envio	0 row(s) affected
2	00:43:14	START TRANSACTION	0 row(s) affected
3	00:43:14	CALL actualizar_stock_retry(9992, 9991, 1)	1 row(s) returned
4	00:43:16	CALL actualizar_stock_retry(9992, 9991, 1)	1 row(s) returned
5	00:43:16	CALL actualizar_stock_retry(9992, 9991, 1)	1 row(s) returned
6	00:43:17	CALL actualizar_stock_retry(9992, 9991, 1)	1 row(s) returned
7	00:43:19	CALL actualizar_stock_retry(9992, 9991, 1)	1 row(s) returned

Lo que termina sucediendo es que dos transacciones intentan actualizar el stock de productos en orden inverso, por lo cual se genera un deadlock.

Debido a la composición del procedimiento almacenado, en donde:

1. Handler SQL captura excepciones
2. Se identifica tipo de error (si hay un deadlock aplica retry)

3. MySQL ejecuta ROLLBACK cuando se detecta deadlock, por lo que la transacción termina completamente deshecha
4. Se aplica un backoff de manera automática
5. Loop maneja un máximo de 2 re-intentos (3 intentos totales)

Luego de constatar la ejecución del retry ante deadlock podemos determinar que el impacto identificado SIN manejo de deadlock, sería que la operación falla completamente, se pierde cualquier dato de la transacción y requiere que haya intervención manual para solucionar el deadlock.

Por el contrario, al implementar mecanismos de retry ante deadlock, nos aseguramos que la operación se completa de manera automática, un correcto manejo de errores, los datos se procesan correctamente y no se requiere intervención manual, asegurando un tiempo de resolución drásticamente menor.

Finalmente, cabe aclarar que con la finalidad de despejar dudas consultamos a la IA respecto a la simulación de un deadlock y un retry ante deadlock, así mismo como a los niveles de aislamiento y cómo realizar un test sobre el distinto comportamiento de los mismos, se adjuntan ejemplo de esta interacción.

Ejemplos interaccion con IA Etapa 5

Prompt 1:

“no deberia usar un bucle para el retry?”

ChatGPT dijo:

¡Exacto! 🤗 Esa es la idea clave: el **retry necesita un bucle**, porque podés tener que intentar la transacción varias veces antes de que funcione o de llegar al límite de reintentos.

En MySQL, típicamente se hace así:

1 Estructura general con loop

1. Declarás un flag de finalización (`done = 0`) y un contador de reintentos (`retries = 0`).
2. Usás un bucle (`REPEAT...UNTIL done END REPEAT`) que:

- Intenta la transacción.
 - Hace commit si sale bien → marca `done = 1`.
 - Si ocurre deadlock → handler hace rollback, espera el backoff y vuelve a intentar.
 - Si supera el máximo de reintentos → handler marca `done = 1` y lanza error.
-

2 Flujo conceptual

INICIO LOOP

```

START TRANSACTION
INTENTAR OPERACIONES (INSERT/UPDATE)
COMMIT → si OK, done = 1
ON ERROR (SQLEXCEPTION)
  ROLLBACK
  retries = retries + 1
  SLEEP(1)      ← backoff
  IF retries >= 3 → done = 1 y SIGNAL ERROR
FIN LOOP
  
```

 Por qué el loop es necesario:

- El handler por sí solo **no repite la transacción**, solo captura el error.
- Sin loop, si ocurre deadlock, harías rollback y **terminarías la ejecución**.
- Con el loop, podés reintentar automáticamente **hasta N veces**, haciendo que tu procedimiento sea más robusto.

Prompt 2:

“Dentro del handler como puedo diferenciar si es deadlock para aplicar backoff y reintentar o si es otro error para que no reintente y salir, se hace con una sentencia if-else en sql?”

ChatGPT dijo:

¡Exacto!  Dentro del handler podés diferenciar los errores usando una **sentencia IF...ELSE**, pero primero necesitás **capturar el código de error real** que MySQL generó.

En MySQL se hace con **GET DIAGNOSTICS**, que te permite obtener:

- `MYSQL_ERRNO` → el número de error MySQL (por ejemplo, 1213 para deadlock).
 - `RETURNED_SQLSTATE` → el código SQLSTATE.
 - `MESSAGE_TEXT` → el mensaje de error.
-

1 Ejemplo conceptual dentro de un handler

```

DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
BEGIN
    DECLARE err_num INT;
    DECLARE err_msg TEXT;

    -- Obtener detalles del error ocurrido
    GET DIAGNOSTICS CONDITION 1
        err_num = MYSQL_ERRNO,
        err_msg = MESSAGE_TEXT;

    -- Diferenciar deadlock de otros errores
    IF err_num = 1213 THEN
        -- Deadlock: hacemos rollback, backoff y reintento
        ROLLBACK;
        SET retries = retries + 1;
        DO SLEEP(1);
    ELSE
        -- Otro error: salimos inmediatamente
        ROLLBACK;
        SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = CONCAT('Error inesperado: ',
err_msg);
    END IF;
END;
  
```

2 Explicación

1. GET DIAGNOSTICS CONDITION 1

- Captura información del error más reciente dentro del handler.

2. IF err_num = 1213

- Detecta si es un deadlock.
- Solo en este caso aplicamos backoff y retry.

3. ELSE

- Para cualquier otro error, hacemos rollback y usamos SIGNAL para lanzar un error descriptivo.



Tip adicional:

- Esto es mucho más robusto que usar solo SQLEXCEPTION, porque **no todos los errores se pueden reintentar**, y algunos pueden dejar la base de datos en un estado inconsistente si se vuelve a ejecutar sin cuidado.

Prompt 3:

“como puedo cambiar el aislamiento que quiero aplicar sobre una transaccion?”

ChatGPT dijo:

Perfecto, vamos paso a paso.

En MySQL, podés cambiar el nivel de aislamiento **a nivel de sesión o solo para una transacción específica**.

1 Para toda la sesión

Si querés que todas las transacciones de tu conexión usen un nivel de aislamiento distinto:

```
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

o

```
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

- Esto **afecta todas las transacciones que inicies después** en esa sesión.

- No cambia lo que ya estaba en curso.
-

2 Solo para una transacción

Si querés que **solo una transacción** use un nivel de aislamiento específico:

```
START TRANSACTION  
ISOLATION LEVEL READ COMMITTED;
```

o

```
START TRANSACTION  
ISOLATION LEVEL REPEATABLE READ;
```

- Luego hacés tus consultas y finalmente **COMMIT**; o **ROLLBACK**;
 - Esto **no afecta las demás transacciones** de la sesión ni de otros usuarios.
-

 Tip adicional:

- MySQL por default usa **REPEATABLE READ**.
- Cambiar a **READ COMMITTED** suele usarse si querés menos bloqueos y no te preocupa ver datos cambiados por otras transacciones entre consultas.

Reflexión sobre el uso de IA en la Etapa 5:

Inicialmente, intentamos implementar el retry directamente en nuestro procedimiento de producción *crear_pedido*, el cual fue propuesto y generado por ChatGPT en base a las consultas y dudas que enviamos. El procedimiento incluía validaciones de cliente, producto, stock, inserts en múltiples tablas y actualización de inventario. Sin embargo, nos encontramos con que el procedimiento se ejecutaba tan rápidamente que resultaba imposible sincronizar dos sesiones para generar el deadlock. Al identificar el problema de timing, nos dimos cuenta que para cuando la segunda sesión iniciaba, la primera ya había completado todas sus operaciones y liberado los locks.

Intentamos implementar SLEEP() con diferentes tiempos de espera, pero descubrimos que las sesiones seguían sin colisionar efectivamente. Tras varias iteraciones probando con 0.5, 1 y 2 segundos, entendimos que el problema no era solo el timing, sino la complejidad del procedimiento. Con 7 operaciones secuenciales (validaciones, inserts, updates), resultaba prácticamente imposible forzar un deadlock.

En base al script original proporcionado por ChatGPT, comenzamos a reducir la complejidad del mismo eliminando distintas secciones del código. Generamos un nuevo procedimiento llamado *actualizar_stock_retry* que mantuviera únicamente las operaciones mínimas necesarias: dos UPDATEs sobre la tabla PRODUCTO con un SLEEP entre ellos. Gracias a previas explicaciones de la IA sobre simulación de deadlock, pudimos aplicar el concepto de bloqueos cruzados, donde nos propusimos simular que dos transacciones intenten acceder a los mismos recursos en orden inverso.

Esta estrategia finalmente resultó exitosa. El deadlock se generó consistentemente, permitiendo observar cómo MySQL detectaba el ciclo de espera mutua (Sesión 1 esperando recurso de Sesión 2, y viceversa), abortaba una de las transacciones con el error 1213, y nuestro handler capturaba este error para ejecutar el retry.

Sobre la estrategia de backoff, ChatGPT nos ayudó a validar que era necesario incluir una espera entre reintentos, y nos explicó conceptualmente por qué el backoff desincroniza las transacciones competidoras. Decidimos implementar 1 segundo de espera y establecimos un límite de 2 reintentos (3 intentos totales en base a la consigna) para prevenir loops infinitos en casos de deadlocks persistentes.

El resultado final demuestra nuestra comprensión de concurrencia en bases de datos, manejo de transacciones con START TRANSACTION/COMMIT/ROLLBACK, y recuperación automática ante errores críticos como deadlocks. La estrategia de separar un procedimiento simple para validación técnica del procedimiento complejo de producción fue una decisión de diseño propia que permitió cumplir con los requisitos académicos sin sacrificar la lógica.

de negocio necesaria en el código real. El código de retry no es simplemente una implementación sugerida por IA, sino una solución consciente que refleja consideraciones de robustez del sistema, experiencia de usuario y buenas prácticas de desarrollo.

Evidencia de uso de IA:

<https://chatgpt.com/share/69053b15-e94c-800a-ab21-630f19c90a67>

Anexo

Uso de IA por etapa:

Etapa 1:

<https://gemini.google.com/share/e49bebe8b2c3>
<https://chatgpt.com/share/68f80bb6-ff78-800a-8414-3873c5ebacb0>

Etapa 2:

<https://gemini.google.com/share/9a185045ec52>

Etapa 3:

<https://gemini.google.com/share/18abfcfd9094a>

Referencias

Ref 1:

<https://www.youtube.com/watch?v=wD2OxnqmKgc>
<https://www.youtube.com/watch?v=kcuMQnGly68>

Ref 0.1:

https://www.w3schools.com/sql/sql_foreignkey.asp
https://www.w3schools.com/sql/sql_check.asp
https://www.w3schools.com/sql/sql_unique.asp
<https://stackoverflow.com/questions/6720050/foreign-key-constraints-when-to-use-on-update-and-on-delete>

Ref 2:

<https://dev.mysql.com/doc/refman/8.4/en/create-view.html>

Ref 3:

<https://learn.microsoft.com/es-es/sql/relational-databases/sql-server-deadlocks-guide?view=sql-server-ver17>
<https://www.w3resource.com/sql-exercises/sql-query-to-handle-deadlocks-using-error-handling-techniques.php>
<https://www.w3resource.com/mysql-exercises/mysql-query-to-handle-a-deadlock-situation-by-retrying-the-transaction.php>
<https://lextonr.wordpress.com/2012/04/14/retry-execution-even-on-deadlock/>

Enlace al video

<https://drive.google.com/file/d/1OkAiYYVa08oyJBqNy2emxUWhPeskh8pW/view?usp=drivelink>